VueJS

Labs

2024-07-23#c8284f6

Lab 1 : The Application Instance

In this lab, we will use what we've learned about the Vue Application Instance so far.

1. Inside of the `app.js` file, create the Application Instance that will be the entry point of your application. Mount it on the `div` with the id `app`.

We want to store our app's title as a `data` attribute of the Application Instance.

2. Print the Application Instance in your browser console. What do you see?

3. Add a `title` data attribute.

4. Print this title through an interpolation in the `h1` tag of your `index.html` file using the mustache syntax `{{ title }}`.

Lab 2 : Tooling

By the end of this lab, we will have transitioned our current project to its `Vite` counterpart. Meaning we will take advantage of a cleaner architecture, single-file components, and all the enhancements to the development experience it has to offer.

Create a project

1. Shutdown your previous server launch in the first lab by using `Ctrl+C` command (the one you used to see the `lab1` and `lab2`).

2. To get started with Vite + Vue, you could simply run:

```
npm init vue@latest
```

and select the following options:

- Eslint

- Prettier

For the sake of the example, **we've already created** a project using `npm init vue@latest` command. You will find it in the `workspaces/lab3` folder.

For now, we have a root `App.vue` component that renders the same HTML file we had earlier but using Single File Component (SFC) syntax.

1. Start your lab 3 development server and navigate to the link displayed in your console.

   ```
   # Go to the `workspaces/lab3` folder
   $ cd lab3

   # Start the dev server
   $ npm install && npm run dev
   ```

   Your view will now be automatically updated whenever you modify a component thanks to the **Hot reload** provided by **Vite**.

2. Open the `src/main.ts` file and see how the Vue instance is created.

3. What do you think each configuration file in the root of your project does ?

Install Vue Devtools

For Vue 3 support, you will install vue-devtools.

- Chrome extension

- Firefox extension

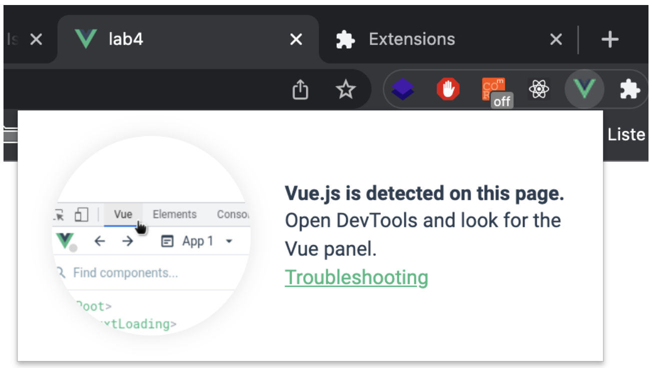
To install or update the Firefox beta version of the devtools, go to one of repository releases and download the xpi file.

- Repository releases

1. Start your development server and navigate to the link displayed in your console.

   ```
   # In your lab3 project directory
   $ npm run dev
   ```

   In your browser, check that your application is detected by the extension.

   Vue.js is detected on this page. Open DevTools and look for the Vue panel. Troubleshooting

2. Open your browser devtools (f12) and click on the Vue tab.

   You must see one component here: `App` .

(Bonus) Use a env variable to display the title

1. Create a .env file
2. Add a specific Vite env variable
3. Display it from `App.vue`

Lab 3 : Options API

In this exercise we'll create a `CardShow` component.

1. Create a `CardShow.vue` component in the directory `src/components`. This component will declare a `show` prop. The prop must be typed as an `Object`.

2. Add a `template` for the `CardShow.vue` component. To do so, move the html template from `App.vue` of the element having the class `.card-result`:

```
<template>
    <div class="card card-result">...</div>
</template>
```

3. Bind the `show` prop from `App.vue` to `CardShow.vue`

4. In `CardShow.vue`, based on the value of the `show.user.favorited` property, create a **computed** property `{{ title }} is your favorite!` or `{{ title }} is NOT your favorite!`. Display this property instead of the show title:

```
  <p class="card-header-title">{{ titleFavorite }}</p>
```

We will now toggle the favorite status. In order to respect the props immutability and the data ownership, we need to propagate the event to the parent that owns the data.

5. Emit a `toggle-favorite` event when we click on the star icon having the class `.card-header-icon`. From `App.vue`, intercept the `toggle-favorite` event by using the `v-on` directive and update the favorite property accordingly.

We will now use the input to filter the shows.

6. Bind a `v-model` directive to a `searchTerm` data to filter shows with the existing input field. You can use the `filter()` function on the `data` imported. Filter only on the `title` field of each show. Finally, on the `v-model`, apply the `.lazy` modifier to trigger the search only when the user has finished writing.

```
<script>
  import shows from '../../../resources/server-formation-vue/shows.js';

  export default {
    // ...
    data: () => ({
      shows,
    }),
    computed: {
      filteredShow () {
        return this.shows.filter(show => show.title.includes(this.searchTerm))
      }
    }
  }

</script>
```

Bonus: use v-model on a custom component

Extract the search input and make it a separate component `SearchForm.vue`.

```
  <div class="field">
     <label class="label">Search</label>
     <div class="control">
       <!-- etc... -->
     </div>
   </div>
```

From the `App.vue`, import `SearchForm.vue`. Use it instead of the current text input.

```
<!-- src/App.vue -->
<SearchForm v-model="searchTerm" />
```

Refactor `SearchForm.vue` so that the `v-model` directive used by `App.vue` works properly.

Hint: Read Component v-model for more info

Lab 4 : Composition API

The main goal of the 2 following labs is to familiarize yourself with the Composition API.

TP 1: Options API => Composition API (setup lifecycle hook)

Open the `workspaces/lab6-1` folder.

Convert `src/App.vue` and `src/components/CardShow.vue` files to use the setup() lifecycle hook (instead of `data`, `computed`, `methods` options).

```
export default defineComponent({
    setup() {
        // your code here
    }
})
```

TP 2: Composition API with script setup

Open the `workspaces/lab6-2` folder.

`src/App.vue` and `src/components/CardShow.vue` use the setup() lifecycle hook. Refactor those components to use only the script setup.

```
<script setup>
// your code here
</script>
```

Lab 5 : Reusability

In this lab, we want to create a feature to focus element easily in the template.

To do so, the easiest way for the developer is to add a `v-focus` on the targeted element.

As we saw earlier, there are two ways of writing this.

Custom directive

1. Add the code in the `src/main.ts` file using the `app.directive` syntax seen in the slides.
2. Use the directive on the `input` element.

Plugin (bonus)

1. Create a `plugins` folder
2. Create a file `focus.js` which export an `install` function

3. Register the directive `v-focus` inside the plugin

4. Back in the `main.ts` file, install the plugin by using the `app.use` method

Composable

Custom composable

1. Create a `composable` folder

2. Create a file `useShows.ts`

3. Create an arrow function with the same name

4. Put in your stateful logic (do not forget to `return` it!)

5. `export` your function

6. From App.vue import your composables and call your freshly created composable

App.vue should take 3 lines of code (inside the script tag).

VueUse (bonus)

1. Go to vueuse.org

2. Search for the `useClipboard` composable

3. Read the doc

4. Import the composable inside the `CardShow` component

5. Use it for the show's description

6. Put a button below the description in order to ease the use of the final user

Lab 6 : Routing

Now that we saw how the router works, we will reorganize our architecture to display our components properly.

1. Install `vue-router` using npm

   ```
   npm install vue-router
   ```

2. Create the router instance inside `src/router.js` with an empty routes array. Use the `createWebHistory()` method as history mode. Export the router instance and use it in your `main.js` file as a plugin.

3. Create a new directory `pages` inside `src`. Create a new component `ShowList.vue` inside the `src/pages` directory. Move all the tag `<div class="hero-body">` and its children from `App.vue` to the `ShowList.vue` component. Do not forget to use the useShow composable!

4. Create a new route named `shows` which will render the page created above to display our list of shows. This route should match the default path: `/`.

5. Copy the page `ShowList.vue` to create a new page `ShowListFavorites.vue`. Update the component to display only starred shows.

6. Create a new route named `favorites` which will display the component created above. The component associated to the route should be lazy-loaded. This route should match the path `/favorites`. Add a link in the toolbar in `App.vue` to access this page.

7. Create a new page `ShowListItemDetails.vue` in `src/pages`. This page will have a props `showId` and will find the show when mounted. It will only display the title of the show and the `CardShow` component.

8. In each card, we'd like the title to be clickable to navigate to a new route (`/shows/:showId`) that displays the page created above. The route must match an ID that will be passed as a prop to the detail component. The component associated to the route should be lazy-loaded.

9. Catch any other route by using a regexp after the param and redirect to the homepage own this scenario. For more details: https://router.vuejs.org/guide/essentials/dynamic-matching.html#catch-all-404-not-found-route

Lab 7 : Store

In this lab, we will use Pinia to set up the state management of our application.

All API calls will be made through the store, as well as searching for shows and retrieving favorited shows.

Thus, our store will have a state, some getters and some actions.

You will find below a relatively step by step guide and a proposal to properly structure your store. As often in software development, there are several ways to solve a problem. So feel free to structure your store differently if you wish.

1. Install Pinia

Install Pinia by following the getting started guide of the documentation. Add the `pinia` dependency and use the `createPinia` method in the `main.ts` file to instantiate the plugin.

2. Define the store

Define a store in a `src/stores/index.js` file:

```
// src/stores/index.js
import { defineStore } from 'pinia'

export const useStore = defineStore('main', () => {
  return {
  }
})
```

You can read this doc for more info.

2.1 State

Configure some state by defining 2 refs:

- `shows` : this property is an array of shows and will store the shows retrieved from the API
- `searchTerm` : this property is a string and will store the term entered by the user during the search

```
// src/stores/index.js
import { defineStore } from 'pinia'

export const useStore = defineStore('main', () => {
  const shows = ref([])
  const searchTerm = ref('')
  return {
    shows,
    searchTerm
  }
})
```

See https://pinia.vuejs.org/core-concepts/state.html

2.2 Getters

Add some getters by using the computed function:

- `filteredShows` : this getter will return a list of shows matching the search criteria entered by the user
- `favoriteShows` : this getter will return the favorite shows

```
// src/stores/index.js

// ...
const filteredShows = computed(() =>
  shows.value.filter((show) =>
    show.title.toUpperCase().includes(searchTerm.value.toUpperCase())
  )
)

const favoriteShows = computed(() =>
  shows.value.filter((show) => show.user.favorited)
)
```

Remember to return these variables to make them accessible externally.

See https://pinia.vuejs.org/core-concepts/getters.html

2.3 Actions

Define some actions:

- `fetchShows` : this asynchronous method retrieves the list of shows via the API and uses the API response to set the `shows` property of the state
- `toggleFavorite` : this asynchronous method takes a show ID in argument. Its role is to toggle the favorite status of the show, updating the `shows` property defined in the state. It also makes a POST request to the API to persist the new favorite status.

```
// src/stores/index.js

// ...

async function fetchShows() {
  const { data } = await API.get('/shows')
  shows.value = data
}
```

```
async function toggleFavorite(showId) {
  const show = shows.value.find((show) => show.id === showId)
  if (!show) {
    console.error('there is no show matching this ID')
    return
  }

  show.user.favorited = !show.user.favorited

  await API.post(`/shows/${showId}/favorites`, {
    isFavorite: show.user.favorited,
  })
}
```

Remember to return these methods to make them accessible externally.

See https://pinia.vuejs.org/core-concepts/actions.html

3. Use the store

We will ensure that the TV Shows are retrieved from the API when the user visits our application, regardless of the page accessed. So inside `src/App.vue`, invoke the `fetchShows` action from your store to retrieve the data and populate your store.

```
<script setup>
// src/App.vue
import { onBeforeMount } from 'vue'
import { useStore } from '@/stores'

const store = useStore()

onBeforeMount(() => {
  store.fetchShows()
})
</script>
```

Then, update the components inside the `pages` folder to use the `shows` from the store. To access the store from your component, the process is always the same:

- import `useStore` from `src/stores/index.js`
- invoke `useStore`
- access you state, getters and actions

For example:

```
import { useStore } from '@/stores'
```

```
const store = useStore()
console.log(store.shows)
```

Lab 8 : Typescript

In this lab, we will gradually migrate our project to Typescript.

To make it easier for you to get started, Typescript is already configured.

1. Create a `src/model.ts` file. Inside this file, export a minimal `ShowInterface` that contains the required properties of a show.

```
// src/model.ts

export interface ShowInterface {
  id: number
  title: string
  genres: string[]
  images: {
    poster: string
  }
  seasons: string
  user: {
    favorited: boolean
  }
  description: string
  creation: string
  status: string
}
```

2. Rename the `src/store/index.js` into `src/store/index.ts` and make it Typescript compliant.

3. Finally, use Typescript in your components inside the `pages` and `components` folders. Start by adding the `lang="ts"` attribute to the `script` tag of your component, then add the necessary return types so that the Typescript server no longer reports errors.