Deven November 13, 2020 0 Comments

Building an E-Commerce app with Vue.js, Vuex & Axios

Ecommerce web applications offer a whole range of new opportunities to business; it helps businesses reduce the costs and can do with fewer overheads & fewer Risks; e-commerce is more comfortable & more Convenient.

Related Articles

December 31, 2020
Build an Anime quiz app using Vuex helper methods

December 7, 2020
Build Hackernews clone with Vuejs by consuming restful API with Axios

November 19, 2020
Building a Reverse Geocoding app in Nuxtjs using Mapbox

November 10, 2020
Working with Nuxt, TypeScript, and ApolloClient: A beginner's Dilemma

October 30, 2020
Creating Custom Hooks with Vue 3 & Typescript

Most of the performance optimization for e-commerce web applications are front-end related, So, the use of a prominent framework, for example, Vue; a front-end centric, often preferred for its straightforwardness.

Vue is a progressive, incrementally-adoptable JavaScript framework for building UI on the web.
Vue is a Simple, minimal core with an incrementally adoptable stack that can handle apps of any scale. Vue is designed from the ground up to be incrementally adoptable.

With Vue.js, we can write the same JavaScript code providing the same functionality in a much simpler way & more comfortable to read and understand.

In this tutorial, we will build an e-commerce site using Vue.js, with the following features:

- Get products from the API
- List the products from the API
- Details Of The Product
- Basic cart management
- Basic user Authentication
- Add products to the cart
- A Checkout page

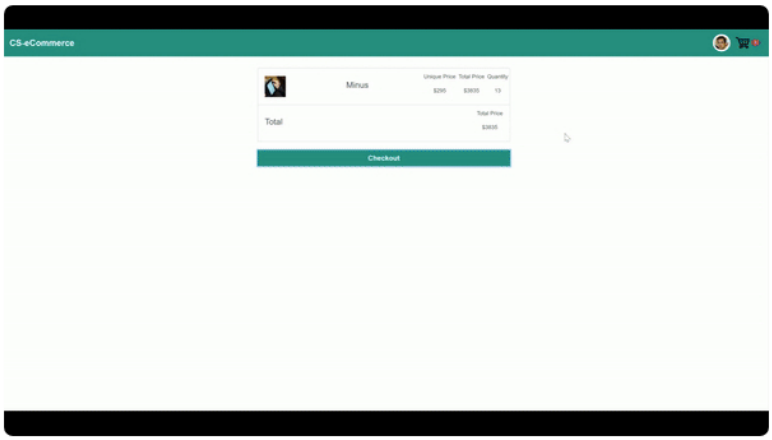
Furthermore, in creating the e-commerce site, we will use Vuex, Vue Router, and Axios. Also, we will show a primary method for handling authentication and cart management in this application.

Before we kick off

> Learn Vue.js and modern, cutting-edge front-end technologies from core-team members and industry

> experts with our premium tutorials and video courses on VueSchool.io.
>
> Click here to Browse all Courses on VueSchool.io

We'll end up with something like this:



Prerequisites

In the course of building this application, we would be using Vuex to manage our application state,
Vue router For navigation and Axios to fetch data from the API.

Sequel to the preceding, we should possess a basic knowledge of JavaScript and Vue to aid easy comprehension of the steps to be followed in this article.

We will also need Node.js for a Vue setup, so download and install it if you haven't already.

Step 1 - Setting up the Vue project

Now let's start by installing **Vue CLI into our machine.**

```
npm install -g @vue/cli
```

Vue CLI helps us to create and manage Vue projects from the command line.

To check if Vue CLI has been installed, run the following command on your terminal:

```
vue --version
```

We will Now use the Vue CLI to build a simple application. To do that open up your terminal and type the following:


```
vue create cs-ecommerce
```

After installation, move into the folder using the cd `cs-eCommerce` .

Installing required packages

Vuex – Vuex is a central store for Vue.js applications. Data stored in the Vuex store can be accessed from any component in the application. To use Vuex in our project, we need to install it in our project. After we have it installed, we can call it from any part of our project.

```
npm install vuex --save
```

Vue Router – It is the official routing package for Vue.js. Vue router helps us to navigate the pages of our application. Each page has a specific path or URL or route that we register in our project's routes.

```
vue add router
```

Axios – Axios is an NPM package for making HTTP requests in our node apps.
With Axios, our application can communicate with other web applications. We can send data and retrieve data from other web software through the Rest API.

There are other methods to communicate with web APIs, but we will install and use Axios for their ease and practicality in our project.

```
npm install axios
```

Creating the initial files.

Having Vuex already installed, we need to create a folder called

store in our project's root path.

In this directory, we will create two modules, one for the user and another for products.
We must also create an `index.js` file where we will import our store modules.

Inside the account and product folders, we need to create five files.

1. `state.js` to store data or states
2. `getters.js` to retrieve data,
3. `actions.js` to create our functions,
4. `mutations.js` to change states
5. `index.js` to import all files from that module.

```
store/
├── account
│   ├── actions.js
│   ├── getters.js
│   ├── index.js
│   ├── mutations.js
│   └── state.js
├── index.js
└── product
    ├── actions.js
    ├── getters.js
    ├── index.js
    ├── mutations.js
    └── state.js
```

To see if it looks similar, you can use the tree command in the store path.

```
tree store
```

We can put states, getters, mutations, and actions in only JavaScript file. But for the organization, we will put each one in different files and inside a folder that we will call a module.

In step 1, you globally installed Vue CLI in your computer, created a Vue project, installed the required Npm packages Vuex, Axios, and Vue Router, and finally created the initial files required for Vuex.

Step 2 – Working with User module

In the `index.js` of the store path, place the code below to import the modules from the store modules. This is the heart of the Vuex.

```
//cs-ecommerce/src/store/index.js
import Vue from 'vue'
import Vuex from 'vuex'
import account from './account'
import product from './product'
Vue.use(Vuex)
export default function () {
  const Store = new Vuex.Store({
    modules: {
      account,
      product
    },
    // enable strict mode (adds overhead!)
    // for dev mode only
    strict: process.env.DEV
  })
  return Store
}
```

We will use `state.js` to store the data that can be accessed by any part of the application. For account, we only need to store user data In the user module state, we will create a function that returns an object with states of that module; in this case, we have only one state: the userData.

```
// cs-ecommerce/src/store/account/state.js
export default function () {
  return {
    userData: {},
  }
}
```

Getters returns the states; for getters of account module, we will create a function that returns the `userData` state.

```
//cs-ecommerce/src/store/account/getters.js
export function user (state) {
    return state.userData
}
```

It is only possible to change states through mutations. In the account module mutation, we will create a function that receives the `userData` state as a parameter and a value that will be assigned to it in the `actions.js` when some function commits it. The mutation function will assign the received value to the `userData` state.

```
//cs-ecommerce/src/store/account/mutations.js
export function setUserData(state, val) {
    state.userData = val
}
```

`mutations.js` is used to set data in the state

In `action.js` of the account module, we will access a dummy API with Axios to fetch a user and then store it in our state userData committing the mutations `setUserData`

```
//cs-ecommerce/src/store/account/actions.js
import router from '../../router'
import Axios from 'axios';
export function login({ commit }) {
    let url = 'https://randomuser.me/api/';
    Axios.get(url).then(function (response) {
        let userData = {
            displayName: response.data.results[0].name.first,
            email: response.data.results[0].email,
            photoURL: response.data.results[0].picture.thumbnail,
            uid: response.data.results[0].login.uuid
        }
        commit("setUserData", userData)
        router.push('/')
    })
        .catch(function (error) {
            console.log(error)
        });
}
```

In the `index.js` we will import getters, mutations, actions, and state.

```
//cs-ecommerce/src/store/account/index.js
import state from './state'
import * as getters from './getters'
import * as mutations from './mutations'
import * as actions from './actions'
export default {
  namespaced: true,
  getters,
  mutations,
  actions,
  state
}
```

In step 2, you created the user module to separate user states from the other application states.

Step 3 – Working with product module

In the product module, we need products state, product state for details of a product, and cart state to store the cart products.

- `products` state – will store product list
- `product` state – will store a specific product
- `cart` state – will store product list in cart

```
//cs-ecommerce/src/store/product/state.js
export default function () {
  return {
```

```
    products: [],
    product: {},
    cart: []
  }
}
```

In the getters.js of the product module, we will create three functions:

- `products` function – Returns the state that store the product list
- `product` – Returns the state that store a specific product when the user wants to see the product details
- `cart` – Returns the state that stores the products in the cart

```
//cs-ecommerce/src/store/product/getters.js
export function products (state) {
    return state.products
}
export function product (state) {
    return state.product
}
export function cart (state) {
    return state.cart
}
```

In the `mutations.js` of product module, we will create four functions.

- `setProducts` – Assign product list to state products
- `setProduct` – Assign an object with a specific product to the product state
- `setCart` – Assigns a list of added products in the cart to cart state

```
//cs-ecommerce/src/store/product/mutations.js
export function setProducts(state, val) {
    state.products = val
}
export function setProduct(state, val) {
    state.product = val
}
export function setLoad(state, val) {
    state.uploadingData = val
}
export function setCart(state, val) {
    state.cart = val
}
```

Inside actions.js we will create a function to retrieve all products with Axios, a function to retrieve details of a product, a function to

add the product to the cart and a function to remove the product from the cart

For this tutorial, we will use a dummy API generated by https://jsonplaceholder.typicode.com/
For this project, we created a JSON server with https://my-json-server.typicode.com/Nelzio/ecommerce-fake-json/products
Basically, on this JSON server, we put a JSON product list.

First, we need to import Axios to fetch data

```
//cs-ecommerce/src/store/product/actions.js
import axios from "axios"

Action to get products list


    export function getProducts({ commit }) {
        let url = "https://my-json-server.typicode.com/Nelzio/ecommerc
        axios.get(url).then((response) => {
            commit("setProducts", response.data);
        }).catch(error => {
            console.log(error);
        });
    }
```

The action to get products is effortless.
We have an endpoint or URL to get the data from the server, and then with the Axios, we do the get at that endpoint.

Action to get product details

```
//cs-ecommerce/src/store/product/actions.js
export function productDetails({ commit }, id) {
    let url = "https://my-json-server.typicode.com/Nelzio/ecommerce-fa
    axios.get(url, { params: { id: id } }).then((response) => {
        commit("setProduct", response.data[0]);
    }).catch(function (error) {
        console.log(error);
    });
}
```

For details of a particular product, the concept does not change from the previous one. The difference is that in Axios, we pass a parameter to retrieve a product. The parameter is the id of the product that we want to see the details.

Action to add to cart.

```
//cs-ecommerce/src/store/product/actions.js
export function addCart({ commit, getters }, payload) {
```

```
    let cart = getters.cart
    cart.push(payload)
    commit("setCart", cart)
}
```

Our action will receive as a parameter an object of product and add it to the cart array.

```
let cart = getters.cart
```

It Catches what is in the cart and put it in a temporary variable

```
cart.push(payload)
```

It Pushes the product object to the temporary variable.

```
commit("setCart", cart)
```

Adds cart array to cart state.

```
//store/product/actions.js
export function removeCart({ commit, getters }, id) {
    let cart = []
    if (id) {
        for (let index = 0; index < getters.cart.length; index++) {
            const element = getters.cart[index];
            if (element.id !== id) {
                cart.push(element)
            }
        }
    }
    commit("setCart", cart)
}
```

To remove a particular object from the cart, we will iterate what is on the cart array and add items that not have the same product id that we want to remove to a temporary variable and then add the cart by committing its mutation.

In step 3, you created a product module to separate its states from other states of our application. The product module deals only with product states. You also used the Rest API generated on https://jsonplaceholder.typicode.com/ to fetch the data via Axios.

Step 4 - Creating the necessary Components

After having everything ready in our store, we will work with the visual part, our components.

In our project, we will use the bootstrap; you can use another framework for styling.
To add bootstrap, we need to add bootstrap CSS CDN inside the head tags of the index.html file in the public folder.

```
//cs-ecommerce/public/index.html
<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootst
```

We can now use bootstrap classes in our project.
Now let's create a parent component that we will call layout.

Layout component

In our project's root path, we will create a folder called layout, and inside of this folder, we'll create a file called `Base.vue` that will be the layout for all our pages.

```
    <div class="page-container">
      <router-view />
    </div>
  </div>
</template>
<script>
import { mapGetters } from "vuex";
export default {
  name: "Base",
  computed: {
    ...mapGetters("account", ["user"]),
    ...mapGetters("product", ["cart"])
  }
};
</script>

<style>
</style>
```

Basically, in our layout, we have a navigation bar represented by the `<nav> </nav>` tags and a main container in the div with the page-container class where the router-view will inject all pages.

To add user and cart information, we will get the data from vuex with `mapGetters`, which will return us user and cart data that are in the states created previously.

Information about the user and cart can only be shown if the user is authenticated.

```
//cs-ecommerce/src/Layout/Base.html

<div v-if="user.photoURL">
  <img
    :src="user.photoURL"
```

```
    class="img-thumbnail profile-image"
    alt
  />
  <router-link class="btn btn-primary my-2 my-sm-0" to="/cart">
    <img
      src="https://pngimg.com/uploads/shopping_cart/shopping_cart_PNG3
      width="50"
      alt
    />
    <span class="badge badge-danger badge-pill">{{ cart.length }}</spa
  </router-link>
</div>
```

For the user, we will show only his profile picture, and for cart, we only need to count how many items have been added, seeing the length of this array.

We will now create a folder called Home which will contain all products component.

Inside of Home folder, we'll create the card component of the products `ProductCard.vue`

```
          type="button"
          class="btn btn-primary btn-lg"
          :to="'/details/' + product.id"
        >Details</router-link>
      </div>
    </div>
  </div>
</template>
<script>
export default {
    name: "ProductCard",
    props: ["product"]
};
</script>
<style>
.card .product-image {
  height: 300px;
}
</style>
```

After creating the product card, we will import it into the Product view and then place it within the product iteration. to do so Create a new file called Products.vue inside the Home folder.

```
</template>
<script>
import { mapActions, mapGetters } from "vuex";
import ProductCard from "../../components/ProductCard";
export default {
  computed: {
    ...mapGetters("product", ["products"]),
  },
  components: { ProductCard },
  methods: {
    ...mapActions("product", ["getProducts", "addCart", "removeCart"
  },
  mounted() {
```

```
    this.getProducts();
  }
};
</script>
<style>
</style>
```

Action method to retrieve products

The code below is an excerpt from the `actions.js` of the products module that we made above.

The data comes from an API, and we will store it in a state by committing the mutation of that state. In that case, it is a product state.

```
//cs-ecommerce/src/store/product/actions.js
export function getProducts({ commit }) {
    let url = "https://my-json-server.typicode.com/Nelzio/ecommerce-fa
    axios.get(url).then((response) => {
        commit("setProducts", response.data);
    }).catch(error => {
        console.log(error);
    });
}
```

To list the products on the Products page, we take the data in the products state, iterate that data, and place each item on a card.

Add to cart component

First, let's create a folder called details & the component to add a product to the cart.
We will set the product, and the quantity to a state called cart.

```
//cs-ecommerce/src/components/details/AddToCart.html
<template>
  <div class="row">
    <div class="col-3">
      <label class="sr-only" for="inlineFormInputName2">Quantity</la
      <input type="number" v-model="quantity" class="form-control mb
    </div>
    <button
      v-if="!isInCardProp"
      @click.stop="addCart({product, quantity})"
      type="button"
      class="btn btn-primary btn-lg btn-block col-9"
    >ADD TO CART</button>
    <button
      v-else
      @click.stop="removeCart(product.id)"
      type="button"
      class="btn btn-primary btn-lg btn-block col-9"
```

To check if the product has been added to the cart, we have a

To check if the product has been added to the cart, we have a method to check `if this item is in the array`.
This method is invoked when the product state or state cart has some change. So they are within watch block of those states.

To add a product to the cart, we take its information and quantity to add it to a single object and then store it in a state.

```
//cs-ecommerce/src/store/product/actions.js
export function addCart({ commit, getters }, payload) {
    let cart = getters.cart
    let data = payload.product
    data["quantity"] = payload.quantity
    cart.push(data)
    commit("setCart", cart)
}
```

To remove a product from the cart, we need to iterate the array in the products state and add each item to a temporary array if it is not the product to be removed and add that temporary array to the cart state.

```
//cs-ecommerce/src/store/product/actions.js
export function removeCart({ commit, getters }, id) {
    let cart = []
    if (id) {
        for (let index = 0; index < getters.cart.length; index++) {
            const element = getters.cart[index];
            if (element.id !== id) {
                cart.push(element)
            }
        }
    }
    commit("setCart", cart)
}
```

In the details view, we will show more details of the product and import the component to add the product details to the cart. The component to add a product to the cart must be shown only if the user is authenticated. also, the directive `v-if` is used to conditionally render a block.

```
  methods: {
    ...mapActions("product", ["productDetails"]),
  },
  mounted() {
    this.productDetails(this.$route.params.idProduct);
  }
};
</script>
<style>
.container-fluid {
  padding: 30px;
}
.image-product {
  width: 100%;
}
```

```
.card * {
  max-height: 85vh;
}
</style>
```

The component to add a product to the cart must be shown only if the user is authenticated.

On the cart page, we will list the products added to the cart and details such as the image, name, price, and quantity of product. We will also calculate the total price of each product and the total to be paid. inside the home folder create a file called `Cart.vue` and add the following code

```
//cs-ecommerce/src/views/home/Cart.vue
<template>
  <div class="container" style="padding: 30px">
    <div class="row d-flex justify-content-center">
      <div class="list-group col-8">
        <a
          v-for="item in cart"
          :key="item.id"
          href="#"
          class="list-group-item list-group-item-action d-flex justi
        >
          <img :src="item.imageUrl" alt height="60" width="60" />
          <p class="h4">{{ item.name }}</p>
          <div class="row">
            <div class="mr-2">
              <p>Unique Price</p>
              <p>${{ item.price }}</p>
            </div>
```

We'll create a simple authentication form, but we will not retrieve data. For login, we have a form with email and password fields that will not be used. And we have a button that will call the login function in the action of the account module previously created. To do so create a folder called account and add the Login.vue file inside the account folder.

```
//cs-ecommerce/src/views/account/Login.vue
<template>
  <div>
    <div class="container" style="padding-top: 10%">
      <div class="row d-flex justify-content-center">
        <div class="col-5 text-left login-form-container">
          <div class="d-flex justify-content-center">
            <img src="https://cdn0.iconfinder.com/data/icons/set-ui-
          </div>
          <div>
            <div class="form-group">
              <label for="exampleInputEmail1">Email address</label>
              <input
                type="email"
                class="form-control"
                id="exampleInputEmail1"
                aria-describedby="emailHelp"
              />
```

For login, we have to get data from an API and save it in the user state.

In our application, we will not log in. We will take a user from randomuser.me and store it in our user state. If the object in the user state is empty, we will assume that the user has not been authenticated.

```
//cs-ecommerce/src/store/account/actions.js
export function login({ commit }) {
    let url = 'https://randomuser.me/api/';
    Axios.get(url).then(function (response) {
        let userData = {
            displayName: response.data.results[0].name.first,
            email: response.data.results[0].email,
            photoURL: response.data.results[0].picture.thumbnail,
            uid: response.data.results[0].login.uuid
        }
        commit("setUserData", userData)
        router.push('/')
    })
        .catch(function (error) {
            console.log(error)
        });
}
```

In step 4, first, you added the bootstrap in your project, then you created the necessary vue components required in your app, and finally, you used the randomuser.me API to store the user from the API to your User state.

Step 5 - Applying some aesthetics to the UI

Now let's add some style to our app. We will do this in our base layout. In fact, styling is made within the `<style> </style>` tags.

```
//cs-ecommerce/src/layout/Base.vue
<style>
nav {
  background-color: teal;
}
.navbar-brand {
  font-weight: bold;
  font-size: 25px;
  color: #ffffff !important;
}
.profile-image {
  width: 50px;
  border-radius: 100% !important;
}
.page-container {
  padding-top: 81px;
}
.btn {
  border-radius: 0%;
```

In the CSS snippet above we have taken the main classes and main tags and made some necessary modifications. we have also some CSS selectors including `.profile-image`, `.page-container`, `.navbar-brand`, `.btn`, `.btn:hover` and `.btn:focus`.

- `.profile-image` sets profile image size in our app
- `.page-container` just defines top padding in our app.
- `.input` makes buttons with square borders
- `.btn`, we changed the color.
- `.btn:hover` We set the hover color, which means when a user will hover the mouse on the button, the color will change to the color we have set.

Let's also add some styling to our login page.

```
//cs-ecommerce/src/views/account/login.vue
<style>
.form-control {
  border-radius: 0%;
  height: 50px;
}
.login-form-container {
  padding: 20px;
  box-shadow: 0px 2px 5px 2px #888888;
}
.btn {
  border-radius: 0%;
  font-weight: bold;
  background: teal;
  border: teal;
}
.btn:hover {
  background: #00b4b4;
}
```

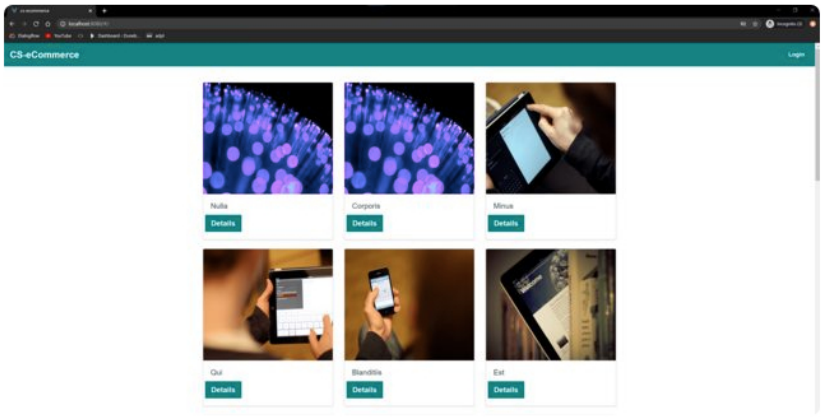
For the login component, we added a shadow border to the `div` of the form and changed the color of the button, and made the button inputs square. the `.login-form-container` CSS selector sets the padding of that form inputs with the box shadow around the inputs.

In step 5, you added the styling to your app and made your app look cleaner and nicer.

Run the application

We have finished building our eCommerce application. Now it's time to run our newly build application.

```
npm run serve
```



This starts a development server that allows you to view your app on http://localhost:8080.

Conclusion

Now we can build eCommerce using Vue and Vuex consuming data from an API. Although the data comes from a dummy API, the logic is the same, but in every API, it's essential to read the documentation to learn how to fetch data.

I hope you learned a few things about Vue. Every article can be made better, so please leave your suggestions and contributions in the comments below. If you have questions about any of the steps, please do also ask in the comments section below.

You can Checkout Source Code on Github

Share Article:

https://codesource.io/building-an-e-commerce

E-Commerce, Vue

Deven

Deven is an Entrepreneur, and Full-stack developer, Constantly learning and experiencing new things. He currently runs CodeSource.io and Dunebook.com

November 13, 2020

Advance model filtering using Laravel when method

November 18, 2020

Understanding API Lifecycle Management and Its Importance in API Design

Comments

ALSO ON **CODESOURCE.IO**

Build a Signature Capture Application ...

10 months ago • 2 comments

In today's post we're going to be exploring how to create an application that ...

Build a TikTok Clone with React and ...

5 months ago • 6 comments

In this article, we will build our TikTok clone with React and firebase, and also we ...

Build a CRUD Application with ...

5 months ago • 2 comments

In this tutorial, we would be building a simple Vue application with Hasura ...

Bui app

9 mo

In th text Nod

0 Comments codesource.io Disqus' Privacy Policy Login

Recommend Tweet Share Sort by Best

Start the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS

Name

Be the first to comment.

Subscribe Add Disqus to your site Do Not Sell My Data DISQUS

posts (and some back-end stuff too!) - Sent 2× a month.

Enter your email here

SIGN UP NOW

You can unsubscribe any time — obviously.

Dev Tips

December 31, 2020
15 frequently asked Flutter interview questions

November 19, 2020
Best PHP Frameworks to Use in 2020

November 18, 2020
Understanding API Lifecycle Management and Its Importance in API Design

October 13, 2020
Top 50 Emacs Commands and Emacs Shortcuts

October 13, 2020
50 Useful Vim Commands basic to Pro

New snippets

December 30, 2020
State in React - explained

December 7, 2020
Asynchronous JavaScript - The Beginners Guide

September 23, 2020
5 ways to clone an Array in JavaScript

September 22, 2020
Centering Things in CSS Using Flexbox

September 3, 2020
CSS box-sizing Property - explained

Android Angular Beginners Bootstrap Code examples CodeIgniter Courses CSS Dev Tips Django Electron Flask Flutter Git GoLang GraphQl Javascript Jquery Kotlin Laravel Machine Learning MySQL Node PHP Python React React Native Snippets SQL Svelte Tutorials Vue

Contact

Privacy

Dunebook

Learn React and React Native

Write for Us