Vue

3

Jour 3

Support réalisé par Pierrick Hauguel

01

Installation de Tanstack Query

Pourquoi Utiliser Tanstack Query ?

Tanstack Query (anciennement React Query) est une bibliothèque puissante pour la gestion des requêtes de données dans vos applications Vue.js. Elle simplifie le traitement des requêtes asynchrones, la mise en cache des résultats, et la gestion des états de chargement et d'erreur, vous permettant de vous concentrer sur l'expérience utilisateur.

Installation et Configuration de Tanstack Query

Pour utiliser Tanstack Query dans votre projet Vue 3, commencez par l'installer via npm ou yarn.

```
npm install @tanstack/vue-query
```

```
import { VueQueryPlugin } from '@tanstack/vue-query'

app.use(VueQueryPlugin)
```

Les Bases des Hooks de Tanstack Query

Les hooks de Tanstack Query comme useQuery et useMutation facilitent la gestion des requêtes et des mutations de données dans vos composants Vue 3, tout en intégrant la gestion des états de chargement et d'erreur.

```
<script setup>
import { useQuery } from '@tanstack/vue-query'

const { isPending, isError, data, error } = useQuery({
  queryKey: ['users'],
  queryFn: fetchUsers,
})
</script>

<template>
  <div v-if="isPending">Loading...</div>
  <div v-else-if="isError">Error: {{ error.message }}</div>
  <ul v-else>
    <li v-for="user in data" :key="user.id">{{ user.name }}</li>
  </ul>
</template>
```

Utilisation de useQuery pour les Requêtes de Données

useQuery est un hook puissant pour effectuer des requêtes de données. Il prend en charge la mise en cache, le refetching, et la gestion des états de chargement et d'erreur.

```
<script setup>
import { useQuery } from '@tanstack/vue-query'

const { isPending, isError, data, error } = useQuery({
  queryKey: ['posts'],
  queryFn: fetchPosts,
})
</script>

<template>
  <div v-if="isPending">Loading...</div>
  <div v-else-if="isError">Error: {{ error.message }}</div>
  <ul v-else>
    <li v-for="post in data" :key="post.id">{{ post.title }}</li>
  </ul>
</template>
```

Gestion des Etats de Chargement et d'Erreur

Tanstack Query gère automatiquement les états de chargement et d'erreur pour vous, vous permettant de fournir une meilleure expérience utilisateur sans code supplémentaire complexe.

```
<script setup>
import { useQuery } from '@tanstack/vue-query'

const { isPending, isError, data, error } = useQuery({
  queryKey: ['comments'],
  queryFn: fetchComments,
})
</script>

<template>
  <div v-if="isPending">Loading...</div>
  <div v-else-if="isError">Error: {{ error.message }}</div>
  <ul v-else>
    <li v-for="comment in data" :key="comment.id">{{ comment.body }}</li>
  </ul>
</template>
```

Paramétrage des Requêtes avec useQuery

useQuery permet de personnaliser les requêtes via des options telles que queryKey, queryFn, et config pour des comportements comme le refetching et la mise en cache.

```
<script setup>
import { useQuery } from '@tanstack/vue-query'

const fetcher = (key, id) => fetch(`/api/data/${id}`).then(res => res.json())

const { isPending, isError, data, error } = useQuery({
  queryKey: ['data', 1],
  queryFn: () => fetcher(1),
})
</script>

<template>
  <div v-if="isPending">Loading...</div>
  <div v-else-if="isError">Error: {{ error.message }}</div>
  <div v-else>{{ data }}</div>
</template>
```

Référencement de Données et Cache dans Tanstack Query

Tanstack Query maintient un cache des données pour améliorer les performances des applications. Les données référencées peuvent être utilisées dans différents composants sans recharger les données.

```
<script setup>
import { useQuery } from '@tanstack/vue-query'

const fetchData = () => fetch('/api/data').then(res => res.json())

const { data } = useQuery({
  queryKey: ['data'],
  queryFn: fetchData,
})
</script>

<template>
  <div>{{ data }}</div>
</template>
```

Introduction à useMutation pour les Opérations d'Ecriture

useMutation est utilisé pour les opérations d'écriture comme les créations, mises à jour ou suppressions de données. Il gère les états de succès et d'erreur pour ces opérations.

```
<script setup>
import { useMutation } from '@tanstack/vue-query'
import axios from 'axios'

const { error, mutate, reset } = useMutation({
  mutationFn: (newTodo) => axios.post('/todos', newTodo),
})

function addTodo() {
  mutate({ id: new Date(), title: 'Do Laundry' })
}
</script>

<template>
  <span v-if="error">
    An error occurred: {{ error.message }}
    <button @click="reset">Reset error</button>
  </span>
  <button @click="addTodo">Create Todo</button>
</template>
```

Gestion Optimiste des Mutations

La gestion optimiste permet de mettre à jour l'interface utilisateur immédiatement après une mutation, avant que la réponse serveur ne soit reçue, offrant une expérience utilisateur plus réactive.

```
<script setup>
import { useMutation, useQueryClient } from '@tanstack/vue-query'
import axios from 'axios'

const queryClient = useQueryClient()

const { mutate } = useMutation({
  mutationFn: (newTodo) => axios.post('/todos', newTodo),
  onMutate: async (newTodo) => {
    await queryClient.cancelQueries(['todos'])

    const previousTodos = queryClient.getQueryData(['todos'])
    queryClient.setQueryData(['todos'], (old) => [...old, newTodo])

    return { previousTodos }
  },
  onError: (err, newTodo, context) => {
    queryClient.setQueryData(['todos'], context.previousTodos)
  },
  onSettled: () => {
    queryClient.invalidateQueries(['todos'])
  },
})

function addTodo() {
  mutate({ id: new Date(), title: 'Do Laundry' })
}
</script>
```

Invalidation et Réactivation de Requêtes

Après une mutation, il est souvent nécessaire d'invalider les requêtes pour obtenir des données fraîches. Tanstack Query facilite cette tâche avec invalidateQueries et refetchQueries.

```
<script setup>
import { useMutation, useQueryClient } from '@tanstack/vue-query'
import axios from 'axios'

const queryClient = useQueryClient()

const { mutate } = useMutation({
  mutationFn: (newTodo) => axios.post('/todos', newTodo),
  onSuccess: () => {
    queryClient.invalidateQueries(['todos'])
  }
})

function addTodo() {
  mutate({ id: new Date(), title: 'Do Laundry' })
}
</script>

<template>
  <button @click="addTodo">Create Todo</button>
</template>
```

Pagination et Requêtes Infinies avec Tanstack Query

Tanstack Query prend en charge la pagination et les requêtes infinies pour charger des données en lots. Cela améliore la performance et l'expérience utilisateur lors de la navigation dans de grandes collections de données.

```
<script setup lang="ts">
import { ref } from 'vue'
import { useQuery } from '@tanstack/vue-query'

const fetcher = (page) =>
  fetch(`https://jsonplaceholder.typicode.com/posts?_page=${page}&_limit=10`
    .then(res => res.json())

const page = ref(1)
const { isPending, isError, data, error } = useQuery({
  queryKey: ['projects', page],
  queryFn: () => fetcher(page.value),
  keepPreviousData: true,
})

const prevPage = () => {
  page.value = Math.max(page.value - 1, 1)
}
const nextPage = () => {
  page.value += 1
}
</script>
```

02

Les Plugins

Introduction aux Plugins Vue

Les plugins Vue.js permettent d'étendre les fonctionnalités de votre application. Ils peuvent ajouter des méthodes globales, des directives, des mixins et bien plus encore. Utiliser des plugins facilite la modularité et la réutilisation du code.

Création de votre Premier Plugin

Créer un plugin Vue est simple. Commencez par définir un fichier JavaScript exportant une fonction d'installation qui ajoute la fonctionnalité souhaitée à l'application Vue.

```
// plugins/monPlugin.js
export default {
  install(app) {
    console.log('Plugin installé');
  }
}
```

Structure de base d'un plugin Vue

Un plugin Vue de base est un objet avec une méthode install. Cette méthode prend l'instance de l'application en argument et optionnellement des options de configuration.

```
// plugins/basePlugin.js
export default {
  install: (app, options) => {
    console.log('Base Plugin installé avec les options:', options)
  }
}

// main.js
import { createApp } from 'vue'
import App from './App.vue'
import basePlugin from './plugins/basePlugin'

const app = createApp(App)
app.use(basePlugin, { option1: 'valeur1' })
app.mount('#app')
```

Ajout de directives personnalisées

Les directives personnalisées permettent d'étendre les fonctionnalités des éléments du DOM.

```
// plugins/directivePlugin.js
export default {
  install: (app, options) => {
    app.directive('color', {
      beforeMount(el, binding) {
        el.style.color = binding.value
      }
    })
  }
}

// main.js
import { createApp } from 'vue'
import App from './App.vue'
import directivePlugin from './plugins/directivePlugin'

const app = createApp(App)
app.use(directivePlugin)
app.mount('#app')
```

Création de composants globaux dans un plugin

Les plugins peuvent également enregistrer des composants globaux accessibles dans toute l'application.

```
// plugins/globalComponents.js
import MyComponent from './components/MyComponent.vue'

export default {
  install: (app, options) => {
    app.component('MyComponent', MyComponent)
  }
}

// main.js
import { createApp } from 'vue'
import App from './App.vue'
import globalComponents from './plugins/globalComponents'

const app = createApp(App)
app.use(globalComponents)
app.mount('#app')
```

Utilisation de provide/inject dans les plugins

Les plugins peuvent utiliser provide et inject pour partager des données ou des méthodes entre les composants.

```
// plugins/providePlugin.js
export default {
  install: (app, options) => {
    app.provide('pluginData', options.data)
  }
}

// main.js
import { createApp } from 'vue'
import App from './App.vue'
import providePlugin from './plugins/providePlugin'

const app = createApp(App)
app.use(providePlugin, { data: 'someData' })
app.mount('#app')
```

03

Mise en place d'un layout

Introduction à la mise en place d'un Layout

Un layout dans une application Vue.js permet de structurer la disposition de l'interface utilisateur, en définissant des sections communes comme l'en-tête, le pied de page et la barre de navigation. Utiliser des layouts permet de réutiliser ces sections à travers différentes pages de l'application.

```
<template>
  <div>
    <Header />
    <main>
      <router-view />
    </main>
    <Footer />
  </div>
</template>

<script setup>
import Header from './components/Header.vue'
import Footer from './components/Footer.vue'
</script>

<style scoped>
/* Styles du layout */
</style>
```

Utilisation de la balise <component :is>

La balise <component :is> est utilisée pour rendre dynamiquement un composant selon une condition. Cela est particulièrement utile pour gérer les layouts variés basés sur les routes.

```
<template>
  <component :is="layout">
    <router-view />
  </component>
</template>

<script setup>
import { computed } from 'vue'
import { useRoute } from 'vue-router'

const route = useRoute()
const layout = computed(() => route.meta.layout || 'default-layout')
</script>

<style scoped>
/* Styles du layout dynamique */
</style>
```

Configuration de Vue Router

Vue Router est utilisé pour définir les routes de l'application et les layouts associés à chaque route via les métadonnées (meta).

```
// router/index.js
import { createRouter, createWebHistory } from 'vue-router'
import DefaultLayout from '../layouts/DefaultLayout.vue'
import AdminLayout from '../layouts/AdminLayout.vue'
import Home from '../views/Home.vue'
import Admin from '../views/Admin.vue'

const routes = [
  {
    path: '/',
    component: Home,
    meta: { layout: DefaultLayout }
  },
  {
    path: '/admin',
    component: Admin,
    meta: { layout: AdminLayout }
  }
]

const router = createRouter({
  history: createWebHistory(),
  routes
})

export default router
```

Création de Layouts personnalisés

Définir des layouts personnalisés en tant que composants Vue permet de réutiliser la structure de base sur différentes pages.

```
<!-- layouts/DefaultLayout.vue -->
<template>
  <div>
    <Header />
    <main>
      <slot />
    </main>
    <Footer />
  </div>
</template>

<script setup>
import Header from '../components/Header.vue'
import Footer from '../components/Footer.vue'
</script>

<style scoped>
/* Styles du layout par défaut */
</style>
```

Intégration des Layouts dans l'Application Vue

Intégrer les layouts dans l'application en utilisant Vue Router et la balise <component :is> pour rendre dynamiquement le layout approprié.

```
<!-- App.vue -->
<template>
  <component :is="layout">
    <router-view />
  </component>
</template>

<script setup>
import { computed } from 'vue'
import { useRoute } from 'vue-router'

const route = useRoute()
const layout = computed(() => route.meta.layout || 'default-layout')
</script>

<style scoped>
/* Styles de l'application */
</style>
```

Exemple complet avec Vue Router et Layouts

Un exemple complet intégrant Vue Router, layouts dynamiques et métadonnées pour définir des layouts spécifiques à chaque route.

```
// main.js
import { createApp } from 'vue'
import App from './App.vue'
import router from './router'
import DefaultLayout from './layouts/DefaultLayout.vue'
import AdminLayout from './layouts/AdminLayout.vue'

const app = createApp(App)

app.component('default-layout', DefaultLayout)
app.component('admin-layout', AdminLayout)

app.use(router)
app.mount('#app')
```

04

Mise en place de tests dans Vue

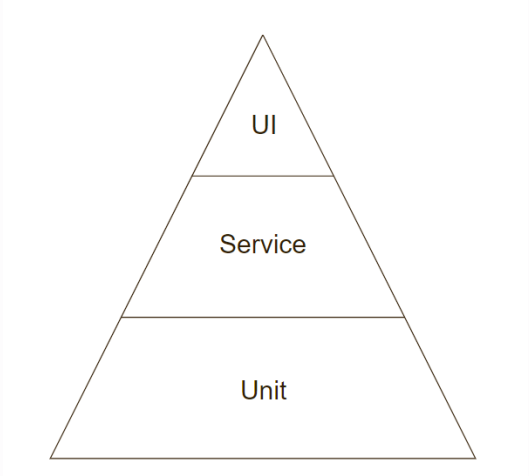
Introduction au test moderne de vue

Les tests automatisés permettent de s'assurer que les fonctionnalités précédemment fonctionnelles continuent de l'être sans intervention manuelle, augmentant ainsi la confiance dans les modifications du code.

Les tests automatisés apportent confiance lors des refactorisations, servent de documentation toujours à jour et préviennent les bugs et régressions.

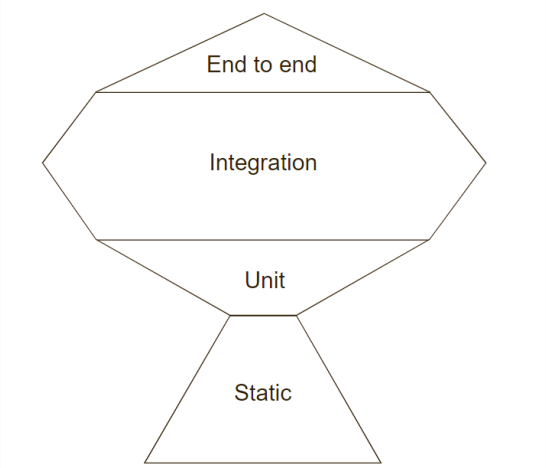
Présentation de la pyramide de tests

Explication de la pyramide des tests qui conseille une grande quantité de tests unitaires par rapport aux tests d'UI qui sont plus coûteux.



Approches alternatives de test : Le testing trophy de Kent C. Dodds

“It says that integration tests give you the biggest return on investment, so you should write more integration tests than any other kinds of tests.”



Quels outils ?

L'analyse statique permet de détecter les erreurs de syntaxe, les mauvaises pratiques et l'utilisation incorrecte des API : Formateurs de code, comme Prettier ; Les linters, comme ESLint ; Les vérificateurs de type, comme TypeScript et Flow.

Les tests unitaires vérifient que les algorithmes délicats fonctionnent correctement. Outils : Jest / Vitest.

Quels outils ?

Les tests d'intégration vous donnent l'assurance que toutes les fonctionnalités de votre application fonctionnent comme prévu. Outils : Vitest & vue-testing-library.

Les tests de bout en bout permettent de s'assurer que votre application fonctionne comme un tout : le frontend, le backend, la base de données et tout le reste. Outils : Cypress ; Playwright

Introduction à Vitest

Vitest est un framework de test moderne et rapide conçu pour les applications front-end, particulièrement bien adapté aux projets utilisant Vite.

```
// Un test simple avec Vitest
import { test, expect } from 'vitest';

test('addition fonctionne correctement', () => {
  expect(1 + 1).toBe(2);
});
```

Configuration de Vitest

Configurez Vitest en ajoutant une section de configuration dans vite.config.js.

```
import { defineConfig } from 'vite';
import vue from '@vitejs/plugin-vue';
import { defineConfig } from 'vitest/config';

export default defineConfig({
  plugins: [vue()],
  test: {
    globals: true,
    environment: 'jsdom',
  },
});
```

Intégration de Vitest avec Vue 3

Vitest s'intègre facilement avec Vue 3 pour tester des composants et des fonctionnalités réactives.

```
import { mount } from '@vue/test-utils';
import { describe, it, expect } from 'vitest';
import MyComponent from './MyComponent.vue';

describe('MyComponent', () => {
  it('renders properly', () => {
    const wrapper = mount(MyComponent);
    expect(wrapper.text()).toContain('Hello Vue 3');
  });
});
```

Utilisation de plugins Vitest

Vitest prend en charge divers plugins pour étendre ses fonctionnalités.

```
// Utilisation du plugin de couverture de code
import { defineConfig } from 'vitest/config';
import { plugin as coveragePlugin } from 'vite-plugin-istanbul';

export default defineConfig({
  plugins: [coveragePlugin()],
});
```

Introduction aux tests unitaires

Les tests unitaires permettent de vérifier le bon fonctionnement des unités individuelles de votre code.

```
// Un test unitaire simple
import { test, expect } from 'vitest';

test('multiplication fonctionne correctement', () => {
  expect(2 * 2).toBe(4);
});
```

Création de tests unitaires avec Vitest

Créez des tests unitaires pour valider le comportement des fonctions et des composants individuels.

```
import { test, expect } from 'vitest';
import { sum } from './sum';

test('sum additionne correctement deux nombres', () => {
  expect(sum(1, 2)).toBe(3);
});
```

Utilisation des assertions de base

Vitest fournit des assertions de base pour vérifier les résultats des tests.

```
import { test, expect } from 'vitest';

test('les chaînes de caractères sont identiques', () => {
  expect('vue').toBe('vue');
});
```

Tests de fonctions JavaScript simples

Testez des fonctions JavaScript simples pour vous assurer qu'elles retournent les résultats attendus.

```
import { test, expect } from 'vitest';
import { isEven } from './isEven';

test('isEven détecte les nombres pairs', () => {
  expect(isEven(2)).toBe(true);
  expect(isEven(3)).toBe(false);
});
```

Configuration de vue-test-utils avec Vitest

Pour tester les composants Vue, configurez vue-test-utils avec Vitest.

```
// Exemples de tests de composants
import { mount } from '@vue/test-utils';
import { describe, it, expect } from 'vitest';
import MyComponent from './MyComponent.vue';

describe('MyComponent', () => {
  it('renders properly', () => {
    const wrapper = mount(MyComponent);
    expect(wrapper.text()).toContain('Hello Vue 3');
  });
});
```

Tests de composants Vue de base

Testez les composants Vue pour vérifier leur rendu et leur comportement.

```
import { mount } from '@vue/test-utils';
import MyComponent from './MyComponent.vue';

test('renders the correct text', () => {
  const wrapper = mount(MyComponent);
  expect(wrapper.text()).toContain('Hello Vue 3');
});
```

Tests des props des composants

Vérifiez que les composants reçoivent et utilisent correctement les props.

```vue
<!-- MyComponent.vue -->
<template>
  <div>Hello {{ name }}</div>
</template>

<script setup>
defineProps(['name']);
</script>
```

```js
import { mount } from '@vue/test-utils';
import MyComponent from './MyComponent.vue';

test('affiche le nom correctement', () => {
  const wrapper = mount(MyComponent, {
    props: { name: 'Vue' },
  });
  expect(wrapper.text()).toContain('Hello Vue');
});
```

Tests des événements des composants

Testez les événements émis par les composants pour vérifier les interactions.

```vue
<!-- MyButton.vue -->
<template>
  <button @click="$emit('click')">Click me</button>
</template>
```

```js
import { mount } from '@vue/test-utils';
import MyButton from './MyButton.vue';

test('émet l\'événement click', async () => {
  const wrapper = mount(MyButton);
  await wrapper.find('button').trigger('click');
  expect(wrapper.emitted()).toHaveProperty('click');
});
```

Tests des slots des composants

Vérifiez que les slots sont correctement rendus et utilisés dans les composants.

```vue
<!-- MyComponent.vue -->
<template>
  <div>
    <slot></slot>
  </div>
</template>
```

```js
import { mount } from '@vue/test-utils';
import MyComponent from './MyComponent.vue';

test('rend le contenu du slot', () => {
  const wrapper = mount(MyComponent, {
    slots: {
      default: '<p>Contenu du slot</p>',
    },
  });
  expect(wrapper.html()).toContain('<p>Contenu du slot</p>');
});
```

Tests des propriétés réactives

Testez les propriétés réactives pour vérifier qu'elles réagissent correctement aux changements.

```vue
<script setup>
import { ref } from 'vue';

const count = ref(0);

function increment() {
  count.value++;
}
</script>

<template>
  <div>
    <p>{{ count }}</p>
    <button @click="increment">Increment</button>
  </div>
</template>
```

```js
import { mount } from '@vue/test-utils';
import MyComponent from './MyComponent.vue';

test('test reactive properties', async () => {
  const wrapper = mount(MyComponent);
  expect(wrapper.text()).toContain('0');
  await wrapper.find('button').trigger('click');
  expect(wrapper.text()).toContain('1');
});
```

Tests des routes avec Vue Router et Vitest

Testez la navigation et le comportement des routes avec Vue Router et Vitest.

```
import { mount } from '@vue/test-utils';
import { createRouter, createWebHistory } from 'vue-router';
import { routes } from './router';
import App from './App.vue';

const router = createRouter({
  history: createWebHistory(),
  routes,
});

test('test route navigation', async () => {
  router.push('/about');
  await router.isReady();
  const wrapper = mount(App, {
    global: {
      plugins: [router],
    },
  });
  expect(wrapper.html()).toContain('About Page');
});
```

Tests des stores Vuex/Pinia avec Vitest

Testez les stores Vuex/Pinia pour vérifier leur comportement et leur intégration avec les composants.

```
import { setActivePinia, createPinia } from 'pinia';
import { useStore } from './store';
import { mount } from '@vue/test-utils';
import MyComponent from './MyComponent.vue';

setActivePinia(createPinia());

test('test Pinia store', () => {
  const store = useStore();
  const wrapper = mount(MyComponent);
  store.increment();
  expect(store.count).toBe(1);
});
```

Introduction aux tests E2E

Les tests End-to-End (E2E) permettent de tester une application entière, du front-end au back-end, pour vérifier le fonctionnement complet.

```
// Un test E2E basique avec Cypress
describe('My First Test', () => {
  it('visits the app root url', () => {
    cy.visit('/');
    cy.contains('h1', 'Welcome to Your Vue.js + TypeScript App');
  });
});
```

Création d'un premier test avec Cypress

Créez un fichier de test dans le dossier cypress/integration et écrivez votre premier test.

```
// cypress/integration/sample_spec.js
describe('Page d\'accueil', () => {
  it('doit afficher le titre de la page', () => {
    cy.visit('/');
    cy.contains('h1', 'Bienvenue sur votre application Vue.js');
  });
});
```

Interactions de base avec Cypress

Cypress permet de simuler des interactions utilisateur telles que les clics, la saisie de texte, et la navigation entre les pages.

```
describe('Formulaire de connexion', () => {
  it('permet à l\'utilisateur de se connecter', () => {
    cy.visit('/login');
    cy.get('input[name="username"]').type('monNomUtilisateur');
    cy.get('input[name="password"]').type('monMotDePasse');
    cy.get('button[type="submit"]').click();
    cy.url().should('include', '/dashboard');
  });
});
```

Utilisation des fixtures pour les données de test

Les fixtures sont utilisées pour fournir des données de test statiques à vos tests. Créez un fichier JSON dans le dossier cypress/fixtures.

```
// cypress/fixtures/user.json
{
  "username": "testuser",
  "password": "password123"
}
```

```
describe('Formulaire de connexion', () => {
  beforeEach(() => {
    cy.fixture('user').then((user) => {
      this.user = user;
    });
  });

  it('permet à l\'utilisateur de se connecter', function() {
    cy.visit('/login');
    cy.get('input[name="username"]').type(this.user.username);
    cy.get('input[name="password"]').type(this.user.password);
    cy.get('button[type="submit"]').click();
    cy.url().should('include', '/dashboard');
  });
});
```

Tests des requêtes API avec Cypress

Cypress permet de tester les requêtes API et de vérifier les réponses pour s'assurer qu'elles sont correctes.

```
describe('Requêtes API', () => {
  it('vérifie la réponse de l\'API', () => {
    cy.request('GET', '/api/users').then((response) => {
      expect(response.status).to.eq(200);
      expect(response.body).to.have.length.greaterThan(0);
    });
  });
});
```

Utilisation de commandes personnalisées

Les commandes personnalisées permettent de réutiliser des séquences d'actions dans différents tests. Elles sont définies dans cypress/support/commands.js.

```
// cypress/support/commands.js
Cypress.Commands.add('login', (username, password) => {
  cy.visit('/login');
  cy.get('input[name="username"]').type(username);
  cy.get('input[name="password"]').type(password);
  cy.get('button[type="submit"]').click();
});

// Utilisation de la commande personnalisée dans un test
describe('Tableau de bord', () => {
  it('affiche le tableau de bord après connexion', () => {
    cy.login('testuser', 'password123');
    cy.url().should('include', '/dashboard');
  });
});
```