Vue 3

Support réalisé par Pierrick Hauguel

22

Mise en place de tests dans Vue

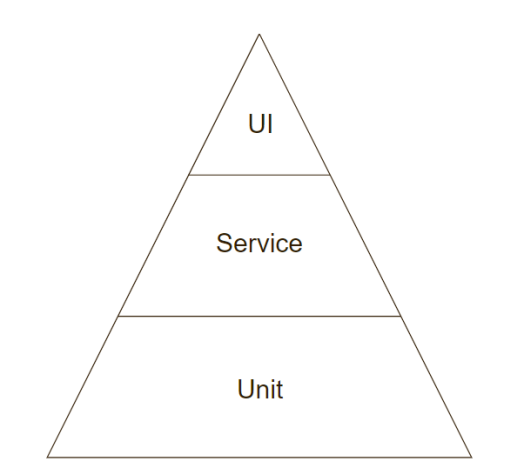
Introduction au test moderne de vue

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.

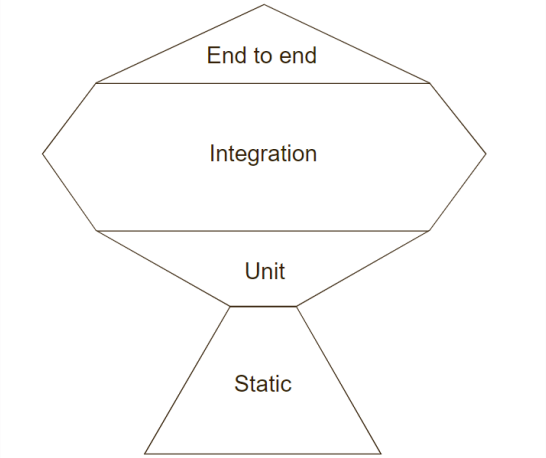
Présentation de la pyramide de tests

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

Vitest

Présentation de Vitest

- Description: Vitest est un framework moderne pour le test JavaScript, inspiré de Jest, rapide et conçu pour être intégré avec Vite.
- Points forts: Support des ESM, test en parallèle, rapidité.

```
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'

export default defineConfig({
  plugins: [react()],
  test: {
    globals: true,
    environment: 'jsdom',
    setupFiles: './src/setupTests.js',
  },
})
```

```
npm install vite vitest @vitejs/plugin-react --save-dev
```

```
import '@testing-library/jest-dom';
```

```
{
  "scripts": {
    "test": "vitest"
  }
}
```

Premier test : qu'est-ce qu'un test ?

Un test vérifie le comportement attendu d'une fonction ou d'un bout de code.

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

test('addition works', () => {
  expect(1 + 1).toBe(2); // Verifies that the result of 1 + 1 is indeed 2
});
```

Qu'est-ce qu'une assertion ?

Une assertion vérifie qu'une condition est vraie. Vitest utilise expect() pour définir les assertions.

```
expect(1 + 1).toBe(2); // This assertion checks if 1 + 1 equals 2
```

Liste des assertions dans Vitest

Liste des principales assertions utilisées dans Vitest.

```
expect(value).toBe(expected); // Asserts that value === expected
expect(value).toEqual(expected); // Asserts deep equality (for objects or arrays)
expect(value).toBeTruthy(); // Asserts that the value is truthy
expect(value).toBeFalsy(); // Asserts that the value is falsy
expect(value).toContain(item); // Asserts that the array or string contains the item
expect(value).toBeGreaterThan(number); // Asserts that the value is greater than a nu
expect(value).toThrow(); // Asserts that a function throws an error
```

Détail sur toBe vs toEqual

Différence entre toBe et toEqual. toBe vérifie l'égalité stricte (===), tandis que toEqual compare la structure des objets et tableaux.

```
expect({ a: 1 }).toEqual({ a: 1 }); // Passes, because the objects are deeply equal
expect({ a: 1 }).toBe({ a: 1 });    // Fails, because they are different object refere
```

Test d'une fonction utilitaire : multiply

Exemple de test sur une fonction utilitaire.

```
export function multiply(a, b) {
  return a * b;
}
```

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

test('multiply function works correctly', () => {
  expect(multiply(2, 3)).toBe(6); // Multiplication of 2 and 3 should return 6
  expect(multiply(2, 0)).toBe(0); // Multiplying by zero should return 0
});
```

Groupement de tests avec describe

Utilisation de describe pour organiser des tests par thème ou par fonction.

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

describe('multiplication tests', () => {
  test('multiply positive numbers', () => {
    expect(multiply(2, 3)).toBe(6); // 2 * 3 equals 6
  });

  test('multiply by zero', () => {
    expect(multiply(2, 0)).toBe(0); // Multiplying any number by 0 returns 0
  });

  test('multiply by negative number', () => {
    expect(multiply(2, -2)).toBe(-4); // 2 * -2 equals -4
  });
});
```

Tester des fonctions asynchrones

Comment tester des fonctions asynchrones dans Vitest.

```
export async function fetchData() {
  return new Promise((resolve) => {
    setTimeout(() => resolve('data'), 1000);
  });
}
```

```
import { fetchData } from '../services/dataService';
import { expect, test } from 'vitest';

test('fetchData returns correct data', async () => {
  const data = await fetchData();
  expect(data).toBe('data'); // Asserts that fetchData resolves to 'data'
});
```

Utilisation de beforeEach et afterEach

beforeEach permet d'exécuter du code avant chaque test, et afterEach après chaque test. Très utile pour initialiser ou nettoyer des variables.

```
let counter;

beforeEach(() => {
  counter = 0; // Initialize counter before each test
});

test('increments counter', () => {
  counter++;
  expect(counter).toBe(1); // The counter should be 1 after incrementing
});

afterEach(() => {
  // Optionally clean up after each test
});
```

Mocking des services HTTP avec Vitest

Utilisation de vi.mock() pour simuler des appels HTTP dans les tests, sans avoir à réellement appeler des services externes.

```
import axios from 'axios';
import { vi } from 'vitest';

vi.mock('axios'); // Mock axios to prevent real HTTP requests
const mockedAxios = vi.mocked(axios);

mockedAxios.get.mockResolvedValue({ data: { userId: 1 } }); // Mock the get request

test('fetches user data', async () => {
  const response = await axios.get('/user/1');
  expect(response.data.userId).toBe(1); // Expect the mocked data to be returned
});
```

Utilisation des stubs pour simuler des comportements

Les stubs permettent de remplacer temporairement une fonction pour tester différentes situations.

```
import { vi } from 'vitest';

const logStub = vi.fn(); // Stub a function

function greet() {
  logStub('Hello'); // Use the stub instead of the real implementation
}

test('log is called with correct argument', () => {
  greet();
  expect(logStub).toHaveBeenCalledWith('Hello'); // Verify that the stub was called
});
```

Mocking de fonctions asynchrones

Comment créer des mocks de fonctions asynchrones avec Vitest.

```
import { vi } from 'vitest';

const asyncFunction = vi.fn().mockResolvedValue('mocked data'); // Mock an async funct

test('async function returns mocked data', async () => {
  const result = await asyncFunction();
  expect(result).toBe('mocked data'); // Expect the mocked result
});
```

Tester des erreurs avec toThrow()

Utilisation de l'assertion toThrow() pour tester si une fonction lance une erreur

```
function throwError() {
  throw new Error('This is an error!');
}

test('function throws an error', () => {
  expect(() => throwError()).toThrow('This is an error!'); // Expect an error to be th
});
```

Gestion des tests avec des valeurs paramétrées

Utilisation de test.each() pour exécuter le même test avec différentes valeurs.

```
test.each([[1, 1, 2], [2, 2, 4], [3, 3, 6]])(
  'adds %i and %i to equal %i',
  (a, b, expected) => {
    expect(a + b).toBe(expected); // Test addition with different values
  }
);
```

Utilisation de test.only pour exécuter un test unique

test.only permet de n'exécuter qu'un seul test, utile pour le débogage.

```
test.only('runs this test only', () => {
  expect(1 + 1).toBe(2); // Only this test will run
});
```

Combiner les assertions pour tester plusieurs aspects

Exemple combinant plusieurs assertions pour vérifier différentes parties du code.

```
test('checks multiple aspects', () => {
  const user = { name: 'Bob', age: 30 };
  expect(user.name).toBe('Bob'); // Verify the name
  expect(user.age).toBeGreaterThan(20); // Verify the age is greater than 20
});
```

Mocking de modules entiers

Utilisation de vi.mock() pour simuler des modules entiers, pas seulement des fonctions spécifiques.

```
vi.mock('../utils/math', () => ({
  multiply: vi.fn(() => 10) // Mock the entire module with custom implementations
}));

import { multiply } from '../utils/math';

test('multiply function is mocked', () => {
  expect(multiply(2, 3)).toBe(10); // The mocked function returns 10 regardless of i
});
```

Écrire des tests pour des fonctions purement utilitaires

Les fonctions utilitaires peuvent être testées isolément car elles n'ont pas de dépendances extérieures.

```
function add(a, b) {
  return a + b;
}

test('add function works', () => {
  expect(add(2, 3)).toBe(5); // Test the utility function directly
});
```

Utilisation de spyOn pour observer les appels de fonction

vi.spyOn() permet de surveiller les appels à une fonction sans la remplacer.

```
const obj = {
  log: (message) => console.log(message),
};

test('spy on log function', () => {
  const spy = vi.spyOn(obj, 'log'); // Spy on the log function

  obj.log('Hello, world!');
  expect(spy).toHaveBeenCalledWith('Hello, world!'); // Check that log was called
});
```

Tester les effets secondaires

Tester des fonctions qui modifient un état externe ou un environnement, par exemple en modifiant des variables globales ou le stockage local.

```
test('modifies global variable', () => {
  global.counter = 0; // Set a global variable
  function incrementCounter() {
    global.counter++;
  }

  incrementCounter();
  expect(global.counter).toBe(1); // Check if the global variable was modified
});
```

Tests avec des timers simulés

Utilisation de vi.useFakeTimers() pour simuler des délais ou des timers dans les tests.

```
test('delays execution', () => {
  vi.useFakeTimers(); // Use fake timers for the test

  let value = false;
  setTimeout(() => {
    value = true;
  }, 1000);

  vi.runAllTimers(); // Fast-forward all timers

  expect(value).toBe(true); // The value should now be true after the timer has run
});
```

Contrôler le temps avec vi.advanceTimersByTime

Avancer manuellement le temps pour simuler des retards dans l'exécution du code.

```
test('advances timers by specific time', () => {
  vi.useFakeTimers();

  let value = false;
  setTimeout(() => {
    value = true;
  }, 1000);

  vi.advanceTimersByTime(500); // Move time forward by 500ms
  expect(value).toBe(false); // The timer hasn't completed yet

  vi.advanceTimersByTime(500); // Move time forward by another 500ms
  expect(value).toBe(true); // Now the timer should have completed
});
```

Mocking des modules externes

Utilisation de vi.mock() pour simuler des modules ou des bibliothèques externes dans les tests.

```
vi.mock('axios', () => ({
  get: vi.fn(() => Promise.resolve({ data: { id: 1 } }))
}));

import axios from 'axios';

test('fetches data with mocked axios', async () => {
  const response = await axios.get('/api/user');
  expect(response.data.id).toBe(1); // Test the mocked response
});
```

Tests de performance avec vi.measure

Utilisation de vi.measure() pour mesurer la performance d'une fonction ou d'un morceau de code.

```
test('measures performance of function', () => {
  const timeTaken = vi.measure(() => {
    let sum = 0;
    for (let i = 0; i < 1000000; i++) {
      sum += i;
    }
    return sum;
  });

  expect(timeTaken.duration).toBeLessThan(100); // Expect the function to execute
});
```

Mocking des dates avec vi.setSystemTime

Simuler des dates et des heures spécifiques pour tester des fonctions dépendantes du temps.

```
test('mocks system date', () => {
  const mockDate = new Date(2020, 1, 1);
  vi.setSystemTime(mockDate); // Set the system time to a specific date

  expect(new Date().getFullYear()).toBe(2020); // The current year should now be 2020
});
```

Mocking des événements avec des callbacks

Utilisation de mocks pour simuler des événements ou des callbacks dans les tests.

```
function onClick(callback) {
  callback('Button clicked');
}

test('calls the callback on click', () => {
  const mockCallback = vi.fn(); // Mock the callback function

  onClick(mockCallback);
  expect(mockCallback).toHaveBeenCalledWith('Button clicked'); // The callback should
});
```

Tests d'interaction entre plusieurs services

Tester l'intégration entre plusieurs services ou modules dans une application.

```
import { getUser, saveUser } from '../services/userService';

test('gets and saves user data', () => {
  const user = getUser(1);
  saveUser(user);

  expect(user.id).toBe(1); // Verify that the correct user was fetched and saved
});
```

Tests de services dépendants de l'état global

Comment tester des services qui dépendent de l'état global ou du contexte de l'application.

```
let globalState = {
  loggedIn: false,
};

function login() {
  globalState.loggedIn = true;
}

test('logs in user and updates global state', () => {
  login();
  expect(globalState.loggedIn).toBe(true); // Check if the global state was updated
});
```

Mocking d'API externes dans les services

Simuler les appels API externes dans les tests de services.

```
vi.mock('../api/externalApi', () => ({
  fetchData: vi.fn(() => Promise.resolve({ data: 'mocked data' }))
}));

import { fetchData } from '../api/externalApi';

test('fetches mocked data', async () => {
  const result = await fetchData();
  expect(result.data).toBe('mocked data'); // Test the mocked API response
});
```

Tester les retours d'erreur des services

Vérifier la gestion des erreurs dans les services en simulant des échecs.

```
function getUser(id) {
  if (id <= 0) {
    throw new Error('Invalid user ID');
  }
  return { id, name: 'John Doe' };
}

test('throws error for invalid user ID', () => {
  expect(() => getUser(-1)).toThrow('Invalid user ID'); // Expect an error to be throw
});
```

Utilisation de toMatchObject pour vérifier des objets partiels

toMatchObject() permet de vérifier partiellement un objet sans nécessiter une égalité complète.

```
const user = { id: 1, name: 'John', age: 30 };

test('matches partial object', () => {
  expect(user).toMatchObject({ id: 1, name: 'John' }); // Only checks for matching
});
```

Exécuter des Tests avec Vitest

Exécution des tests avec Vitest via la ligne de commande et en mode watch.

```
npm run test  # Exécute tous les tests
npm run test -- --watch  # Mode watch pour réexécuter automatiquement les tests
```

Exécution des Tests avec Couverture

Comment générer des rapports de couverture de tests avec Vitest pour visualiser les zones non testées du code.

```
vitest run --coverage
```

Optimisation des Tests avec Vitest

Optimisation des performances des tests en activant les tests en parallèle et le mode watch.

```
test: {
  environment: 'jsdom',
  maxThreads: 4,
  minThreads: 2,
},
```

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()],
});
```

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.

```
<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>
```

```
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);
});
```

Utiliser des Mocks de Données

Les mocks de données permettent de simuler des réponses d'API ou des données complexes dans vos tests.

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

test('renders with mock data', () => {
  const mockData = ref([{ id: 1, name: 'Test Item' }]);
  const wrapper = mount(MyComponent, {
    setup() {
      return { data: mockData };
    },
  });
  expect(wrapper.text()).toContain('Test Item');
});
```

Mock de Modules

Vous pouvez mocker des modules entiers pour simuler des fonctions ou des classes dans vos tests.

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

vi.mock('../src/api', () => ({
  fetchData: () => [{ id: 1, name: 'Mocked Item' }],
}));

test('uses mocked module', async () => {
  const wrapper = mount(MyComponent);
  await wrapper.vm.$nextTick();
  expect(wrapper.text()).toContain('Mocked Item');
});
```

Mock de Fonctions

Mocker des fonctions individuelles permet de contrôler le comportement de certaines méthodes pendant les tests.

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

const mockFunction = vi.fn(() => 'Mocked Value');

test('calls mocked function', () => {
  const wrapper = mount(MyComponent, {
    setup() {
      return { myFunction: mockFunction };
    },
  });
  wrapper.vm.myFunction();
  expect(mockFunction).toHaveBeenCalled();
});
```

Utilisation de vi.spyOn

vi.spyOn permet de surveiller et de mocker les implémentations des méthodes d'objets.

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

const mockFetch = vi.spyOn(window, 'fetch').mockResolvedValue({
  json: () => Promise.resolve([{ id: 1, name: 'SpyOn Item' }]),
});

test('fetches data using spyOn', async () => {
  const wrapper = mount(MyComponent);
  await wrapper.vm.fetchData();
  expect(wrapper.text()).toContain('SpyOn Item');
});
```

Mock d'API

Les mocks d'API permettent de simuler des appels réseau et de contrôler les réponses pour les tests.

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

const mockApi = ref([{ id: 1, name: 'Mocked API Item' }]);

test('renders with mocked API data', () => {
  const wrapper = mount(MyComponent, {
    setup() {
      return { apiData: mockApi };
    },
  });
  expect(wrapper.text()).toContain('Mocked API Item');
});
```

Mock de Composants Enfants

Mocker des composants enfants permet de tester un composant parent sans dépendre des implémentations des enfants.

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

test('renders with mocked child component', () => {
  const wrapper = mount(MyComponent, {
    global: {
      stubs: {
        ChildComponent: {
          template: '<div>Mocked Child Component</div>',
        },
      },
    },
  });
  expect(wrapper.html()).toContain('Mocked Child Component');
});
```

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');
  });
});
```

Premier test avec Playwright

Exemple de test basique avec Playwright pour vérifier le rendu de la page d'accueil.

```
// tests/home.spec.js
const { test, expect } = require('@playwright/test');

test('La page d\'accueil doit afficher le titre', async ({ page }) => {
  await page.goto('http://localhost:3000');
  await expect(page).toHaveTitle(/Bienvenue à Playwright avec Vue 3/);
});
```

Test d'interactions utilisateur

Exemple de test d'une interaction utilisateur sur un composant.

```
// tests/counter.spec.js
const { test, expect } = require('@playwright/test');

test('Le bouton doit incrémenter le compteur', async ({ page }) => {
  await page.goto('http://localhost:3000');
  await page.click('button');
  await expect(page.locator('button')).toHaveText('Compteur: 1');
});
```

Test de soumission de formulaire

Vérifier la soumission de formulaire avec Playwright.

```
// tests/form.spec.js
const { test, expect } = require('@playwright/test');

test('Le formulaire doit soumettre le nom d\'utilisateur', async ({ page }) => {
  await page.goto('http://localhost:3000');
  await page.fill('input', 'testuser');
  await page.click('button[type="submit"]');
  page.on('dialog', dialog => {
    expect(dialog.message()).toBe("Nom d'utilisateur: testuser");
    dialog.dismiss();
  });
});
```

Mocking des requêtes HTTP

Utiliser Playwright pour intercepter et mocker les requêtes HTTP.

```
// tests/mock.spec.js
const { test, expect } = require('@playwright/test');

test('Mocker une requête API', async ({ page }) => {
  await page.route('https://api.example.com/data', route =>
    route.fulfill({
      contentType: 'application/json',
      body: JSON.stringify({ data: 'Mocked Data' }),
    })
  );
  await page.goto('http://localhost:3000');
  // Continue with further assertions
});
```

Introduction à Storybook

Storybook est un outil open-source pour développer et documenter des composants UI de manière isolée. Il permet aux développeurs de tester, visualiser et organiser les composants dans un environnement sandbox.

Installation de Storybook pour Vue 3

L'installation de Storybook pour un projet Vue 3 se fait via npm ou yarn. Cela inclut l'ajout des dépendances nécessaires et la configuration initiale pour démarrer Storybook.

```
npx storybook init
```

Configuration Initiale

La configuration initiale de Storybook implique la modification des fichiers de configuration pour s'assurer que Storybook peut charger et rendre les composants Vue correctement.

```
my-project/
├── src/
│   ├── components/
│   │   └── MyComponent.vue
│   └── stories/
│       └── MyComponent.stories.js
└── .storybook/
    ├── main.js
    └── preview.js
```

Création de Stories

Les stories sont des représentations visuelles des composants. Chaque story décrit un état ou une variation d'un composant, permettant de les visualiser et de les tester en isolation.

Création de votre première story

Pour créer une première story, il suffit de définir un composant, de créer un template, et de spécifier les arguments (props) nécessaires pour cette instance particulière du composant.

```
import { fn } from '@storybook/test';

import Task from './Task.vue';

export const ActionsData = {
  onPinTask: fn(),
  onArchiveTask: fn(),
};

export default {
  component: Task,
  title: 'Task',
  tags: ['autodocs'],
  //👇 Our exports that end in "Data" are not stories.
  excludeStories: /.*Data$/,
  args: {
    ...ActionsData
  }
};
```

```
export const Default = {
  args: {
    task: {
      id: '1',
      title: 'Test Task',
      state: 'TASK_INBOX',
    },
  },
};

export const Pinned = {
  args: {
    task: {
      ...Default.args.task,
      state: 'TASK_PINNED',
    },
  },
};
```

Stories et Composition API

La Composition API de Vue 3 permet de structurer le code de manière plus flexible et modulaire. Les stories peuvent intégrer la Composition API pour créer des composants plus complexes et maintenables.

```
// src/stories/CompositionAPI.stories.js
import { ref } from 'vue';
import MyComponent from '../components/MyComponent.vue';

export default {
  title: 'Example/CompositionAPI',
  component: MyComponent,
};

const Template = (args) => ({
  components: { MyComponent },
  setup() {
    const state = ref(args.initialState);
    return { state };
  },
  template: '<MyComponent :state="state" />',
});

export const UsingCompositionAPI = Template.bind({});
UsingCompositionAPI.args = {
  initialState: 'State from Composition API',
};
```

Utilisation de la Composition API dans les stories

La Composition API peut être utilisée dans les stories pour gérer les états locaux, les réactifs, et les computed properties, rendant les stories plus interactives et dynamiques.

```
// src/stories/UseCompositionAPI.stories.js
import { ref, computed } from 'vue';
import MyComponent from '../components/MyComponent.vue';

export default {
  title: 'Example/UseCompositionAPI',
  component: MyComponent,
};

const Template = (args) => ({
  components: { MyComponent },
  setup() {
    const state = ref(args.initialState);
    const doubled = computed(() => state.value * 2);
    return { state, doubled };
  },
  template: '<MyComponent :state="state" :doubled="doubled" />',
});

export const DynamicState = Template.bind({});
DynamicState.args = {
  initialState: 2,
};
```

Intégration des hooks de la Composition API

Les hooks de la Composition API, tels que ref, reactive, computed, et watch, peuvent être intégrés dans les stories pour créer des scénarios interactifs et sophistiqués.

```vue
// src/stories/CompositionHooks.stories.js
import { ref, watch } from 'vue';
import MyComponent from '../components/MyComponent.vue';

export default {
  title: 'Example/CompositionHooks',
  component: MyComponent,
};

const Template = (args) => ({
  components: { MyComponent },
  setup() {
    const count = ref(args.initialCount);
    watch(count, (newVal) => {
      console.log('Count changed:', newVal);
    });
    return { count };
  },
  template: '<MyComponent :count="count" />',
});

export const UsingHooks = Template.bind({});
UsingHooks.args = {
  initialCount: 0,
};
```

Addons de Storybook

Les addons augmentent les capacités de Storybook en ajoutant des fonctionnalités supplémentaires comme les actions, les knobs, les backgrounds, etc. Ils permettent d'améliorer l'expérience de développement et de documentation.

Configuration des addons populaires

Certains addons sont très populaires et souvent utilisés, comme @storybook/addon-actions pour gérer les événements et @storybook/addon-knobs pour manipuler les props dynamiques des composants.

```
// .storybook/main.js
module.exports = {
  addons: ['@storybook/addon-knobs', '@storybook/addon-essentials'],
};
```

Documentation des composants avec addon-info

@storybook/addon-info permet de documenter automatiquement les composants et leurs props. Cela inclut les descriptions, les types de props, et les valeurs par défaut, offrant une documentation complète et interactive.

```
// .storybook/main.js
module.exports = {
  addons: ['@storybook/addon-info'],
};
```

E2E avec Cypress

Introduction à Cypress

Cypress est un outil puissant pour les tests end-to-end (E2E) conçu pour les applications web modernes.

```
npm install cypress --save-dev
```

Lancer Cypress

Une fois installé, Cypress peut être lancé en mode interactif ou en mode headless.

```
npx cypress open   # Ouvre l'interface graphique de Cypress
npx cypress run    # Exécute les tests en mode headless
```

Configuration de Base de Cypress

Configuration de Cypress dans cypress.config.js pour adapter le comportement des tests (timeout, viewport, etc.).

```
// cypress.config.js
module.exports = {
  e2e: {
    baseUrl: 'http://localhost:3000',
    viewportWidth: 1280,
    viewportHeight: 720,
  },
};
```

Premier Test End-to-End (E2E)

Écrire un test simple avec Cypress qui vérifie le rendu de la page d'accueil d'une application.

```
describe('Homepage Test', () => {
  it('should load the homepage', () => {
    cy.visit('/');
    cy.contains('Welcome to My App').should('be.visible');
  });
});
```

Sélection des Éléments avec Cypress

Utiliser cy.get() pour sélectionner des éléments dans le DOM.Sélection par classe, ID, attributs, etc.

```
cy.get('.button-class').click();
cy.get('#element-id').should('have.text', 'Expected Text');
cy.get('[data-cy=submit-button]').click();
```

Assertions avec Cypress

Cypress utilise Chai pour les assertions. Tester les états des éléments, les URL, etc.

```
cy.url().should('include', '/dashboard');
cy.get('.alert').should('have.class', 'success');
cy.get('input').should('have.value', 'Cypress Test');
```

Interaction Utilisateur avec Cypress

Simuler des actions utilisateur comme click(), type(), clear().

```
cy.get('input[name="email"]').type('test@example.com');
cy.get('button[type="submit"]').click();
```

Tests Asynchrones avec Cypress

Cypress gère automatiquement les actions asynchrones. Il attend que les éléments soient disponibles avant d'exécuter les actions suivantes.

```
cy.get('button').click();
cy.get('.loading-spinner').should('not.exist');
cy.get('.result').should('contain', 'Success');
```

Gérer les Requêtes HTTP avec cy.intercept()

Intercepter et mocker les requêtes HTTP pour simuler des réponses d'API.

```
cy.intercept('GET', '/api/data', { fixture: 'data.json' }).as('getData');
cy.visit('/');
cy.wait('@getData');
```

Utilisation des Fixtures

Les fixtures sont des fichiers JSON ou autres formats qui contiennent des données de test pour simuler des réponses de l'API.

```
cy.fixture('user.json').then((user) => {
  cy.get('input[name="username"]').type(user.username);
  cy.get('input[name="password"]').type(user.password);
});
```

Tests de Formulaires avec Cypress

Tester des formulaires en validant les inputs, soumissions et réponses.

```
cy.get('input[name="email"]').type('test@example.com');
cy.get('input[name="password"]').type('password123');
cy.get('form').submit();
cy.get('.success-message').should('be.visible');
```

Assertions d'URL et de Navigation

Vérifier que l'URL et la navigation sont correctes après certaines actions.

```
cy.url().should('include', '/dashboard');
cy.get('a[href="/profile"]').click();
cy.url().should('include', '/profile');
```

Gestion des Cookies et Local Storage

Cypress permet d'interagir avec les cookies et le local storage pour gérer des sessions d'utilisateur.

```
cy.setCookie('session_id', '123ABC');
cy.getCookie('session_id').should('have.property', 'value', '123ABC');

cy.window().then((win) => {
  win.localStorage.setItem('token', 'authToken');
});
```

Tests avec Authentication

Gérer l'authentification avec Cypress, simuler des connexions utilisateurs et des sessions.

```
cy.request('POST', '/login', {
  username: 'user1',
  password: 'password',
}).then((response) => {
  cy.setCookie('authToken', response.body.token);
  cy.visit('/dashboard');
});
```

Intégration avec CI/CD

Intégration de Cypress dans un pipeline CI/CD comme GitHub Actions ou GitLab CI.

```yaml
name: Cypress Tests

on: [push]

jobs:
  cypress-run:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Install dependencies
        run: npm install
      - name: Run Cypress tests
        run: npx cypress run
```

Mocks avec MSW

Installation de MSW

Installer MSW pour simuler les requêtes réseau dans les tests d'intégration.

```
npm install --save-dev msw
```

Initialisation du Service Worker

Générer le script du Service Worker avec la commande msw init.

```
import * as msw from 'msw';
import { setupWorker } from 'msw/browser';
import { handlers } from './handlers';

export const worker = setupWorker(...handlers);
window.msw = { worker, ...msw };
```

Démarrage du Service Worker

Démarrer le Service Worker en mode développement.

```
async function enableMocking() {
  if (process.env.NODE_ENV !== 'development') return;

  const { worker } = await import('./mocks/browser');
  return worker.start();
}

enableMocking().then(() => {
  const root = createRoot(document.getElementById('root'));
  root.render(<App />);
});
```

Handlers MSW

Créer des définitions de mocks pour les requêtes réseau.

```
import { http, HttpResponse } from 'msw';

export const handlers = [
  http.get('https://httpbin.org/anything', () => {
    return HttpResponse.json({
      args: { ingredients: ['bacon', 'tomato', 'mozzarella', 'pineapples'] }
    });
  })
];
```

E2E avec Playwright

Qu'est-ce que Playwright ?

Playwright est un outil de test de bout en bout qui permet de tester des applications web en simulant des interactions utilisateur réelles.

```
const { test, expect } = require('@playwright/test');

test('example test', async ({ page }) => {
  await page.goto('https://example.com');
  await expect(page).toHaveTitle('Example Domain');
});
```

Avantages de Playwright

Playwright offre une expérience de test et de débogage optimale, permet d'inspecter la page à tout moment et supporte plusieurs navigateurs.

```
const { chromium } = require('playwright');

(async () => {
  const browser = await chromium.launch();
  const page = await browser.newPage();
  await page.goto('https://example.com');
  await browser.close();
})();
```

Playwright vs Cypress

Playwright offre une API plus cohérente, une configuration plus simple, supporte les multi-onglets et est plus rapide que Cypress.

```
const { test, expect } = require('@playwright/test');

test('compare with cypress', async ({ page }) => {
  await page.goto('https://example.com');
  await expect(page.locator('h1')).toContainText('Example Domain');
});
```

Fichier de configuration Playwright

Définir les paramètres globaux et les projets pour chaque navigateur.

```
const { defineConfig, devices } = require('@playwright/test');

module.exports = defineConfig({
  testDir: './tests',
  fullyParallel: true,
  reporter: 'html',
  use: {
    baseURL: 'http://localhost:3000',
    trace: 'on-first-retry'
  },
  projects: [
    { name: 'chromium', use: { ...devices['Desktop Chrome'] } },
    { name: 'firefox', use: { ...devices['Desktop Firefox'] } },
    { name: 'webkit', use: { ...devices['Desktop Safari'] } }
  ],
  webServer: {
    command: 'npm run start',
    url: 'http://localhost:3000',
    reuseExistingServer: !process.env.CI
```

Test de base avec Playwright

Créer un test de base pour vérifier le texte "welcome back".

```
const { test, expect } = require('@playwright/test');

test('hello world', async ({ page }) => {
  await page.goto('/');
  await expect(page.getByText('welcome back')).toBeVisible();
});
```

Requête d'éléments avec Playwright

Utiliser des sélecteurs sémantiques pour requêter les éléments DOM.

```
await page.getByRole('button', { name: 'submit' }).click();
```

Tester les interactions de base

Tester la navigation et l'interaction de base.

```
const { test, expect } = require('@playwright/test');

test('navigates to another page', async ({ page }) => {
  await page.goto('/');
  await page.getByRole('link', { name: 'remotepizza' }).click();
  await expect(page.getByRole('heading', { name: 'pizza' })).toBeVisible();
});
```

Tester les formulaires avec Playwright

Utiliser des locators pour remplir et soumettre des formulaires.

```
const { test, expect } = require('@playwright/test');

test('should show success page after submission', async ({ page }) => {
  await page.goto('/signup');
  await page.getByLabel('first name').fill('Chuck');
  await page.getByLabel('last name').fill('Norris');
  await page.getByLabel('country').selectOption({ label: 'Russia' });
  await page.getByLabel('subscribe to our newsletter').check();
  await page.getByRole('button', { name: 'sign in' }).click();
  await expect(page.getByText('thank you for signing up')).toBeVisible();
});
```

Tester les liens ouvrant dans un nouvel onglet

Vérifier l'attribut href ou obtenir le handle de la nouvelle page après avoir cliqué sur le lien.

```
const pagePromise = context.waitForEvent('page');
await page.getByRole('link', { name: 'terms and conditions' }).click();
const newPage = await pagePromise;
await expect(newPage.getByText("I'm baby")).toBeVisible();
```