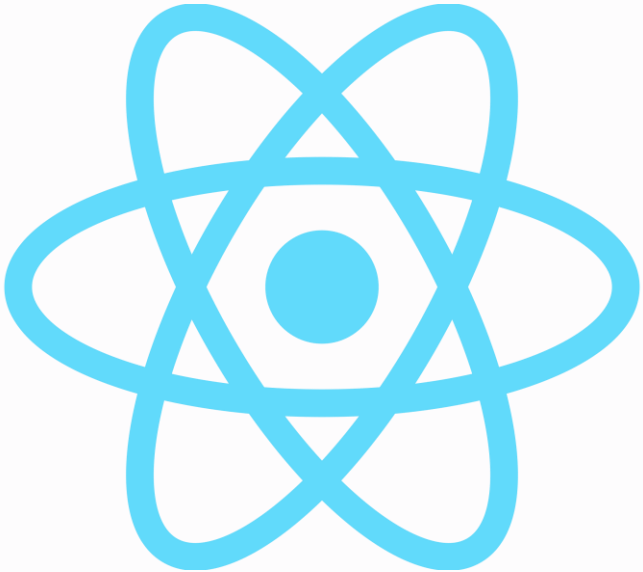
React JS



Support by Pierrick Hauguel

01

Training presentation

First Day

- **Introduction and Review of Best Practices in React**
- - Source Organization
- - Use of hooks
- - Optimizations: memoization, virtual DOM
- - ErrorBoundary
- - ESLint rules
- - Strict mode

- **Advanced Patterns in React**
- - Combining hooks
- - Using useEffect and useContext to trigger actions
- - Pattern of functions as children

- **Introduction to TanStack Query**
- - Fundamentals of TanStack Query
- - Management of asynchronous data states
- - Caching, refetching, and optimizations
- - Integration with React

Second Day

- **Introduction to Zustand**
- - Basic principles of Zustand
- - Creation and management of simplified global states
- - Using Zustand in React components
- - Optimizations and best practices

- **Improving Application Performance**
- - Using React Dev Tools
- - Concurrent mode and Server Side Rendering (introduction)
- - Splitting code

Third Day

- **Advanced Testing in React**
- - Testing hooks
- - Testing components using hooks
- - Asynchronous tests
- - Advanced mocks

- **New Features in React 19**
- - New hooks
- - Changes and breaking changes
- - Review of new features

More topics ?

- Complex forms ?
- Good practices with types ?
- How do you manage complex local data ? useState ? Something else ?
- How to override the TS namespace of React? Which usecase ?

02

How to test?

Understanding the test objective

- **What you need to test :**
  - **Identify precisely the element to be tested:** is it a utility function, a React component, a specific user interaction?
    - *Example:* Test whether the "Send" button correctly triggers form submission.

  - **Determine the importance of the test:**
    - Prioritize testing of user-critical features.
    - *Example:* If authentication is essential, concentrate on login and logout tests.

Identify appropriate test types

- **Choose the right type of test:**
- **Unit testing :**
  - For testing isolated functions or components.
  - *Example:* Check that the tax calculation function returns the correct amount.
- **Integration testing :**
  - To test the interaction between several components or modules.
  - *Example:* Check that the basket component updates the total when items are added.
- **End-to-end testing (E2E) :**
  - To simulate real user behavior.
  - *Example:* Test the entire purchasing process, from product selection to payment.

Defining test cases

Typical use scenarios :

- **List the common uses of the component or function.**
  - *Example:* For a contact form, test submission with valid and invalid data.
- **Possible entries :**
- **Consider borderline cases and invalid values.**
  - *Example:* What happens if the user leaves a mandatory field blank or enters an incorrect date format?
- **Expected results :**
- **Clearly define what should happen in each scenario.**
  - *Example:* An error message will be displayed if the email field is invalid.

Writing behavior-based tests

Best practices :

- **Test visible effects:**
  - *Solution:* Check that the interface reacts as expected. For example, after clicking on "Add to cart", the number of items should increase.

- **Avoid tests that are too specific to the internal implementation :**
  - *Solution:* Don't test the component's internal states, which may change during refactoring.

Integrating testing into the development workflow

- **Questions to ask yourself :**
  - **When should I write the tests?**
    - *Solution:* Adopt the method that suits you best. TDD (Test Driven Development) forces you to write tests before code, which can improve quality.
  - **How do tests fit into the continuous integration process?**
    - *Solution:* Configure your CI/CD pipeline to run tests automatically on commits or pull requests.

Adopt a test-driven mentality

Tips for developing this mindset:

- **Think like a user:**
  - *Solution:* Try to anticipate the actions and mistakes the user might make.
    - *Example:* Testing navigation with invalid session data.

- **Consider potential errors :**
  - *Solution:* Identify your application's weak points.
    - *Example:* What happens if the server is down?

- **Be systematic:**
  - *Solution:* Use checklists or test matrices to cover different scenarios without forgetting important cases.

Continuously review and improve tests

Questions to ask yourself :

- **Are my tests reliable and robust?**
  - *Solution:* Avoid unstable tests by correctly managing asynchronisms and cleaning up the DOM after each test.
- **Are they easy for other team members to understand?**
  - *Solution:* Use clear language and simple code structures.
- **Do they sufficiently cover the application's critical functions?**
  - *Solution:* Use code coverage tools to identify untested areas.

03

Setting up tests in React

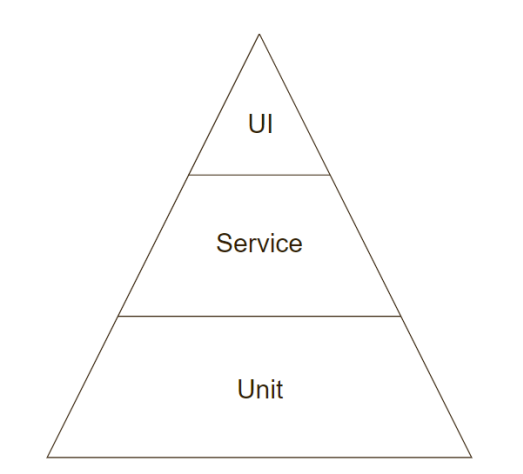
Introduction to modern React testing

Automated testing ensures that previously functional features continue to work without manual intervention, thus increasing confidence in code modifications.

Automated tests provide confidence during refactoring, serve as up-to-date documentation and prevent bugs and regressions.

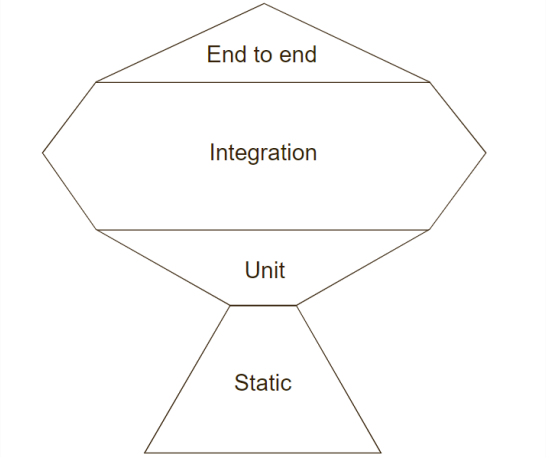
Presentation of the test pyramid

Explanation of the test pyramid, which recommends a large number of unit tests as opposed to the more costly UI tests.



Alternative approaches to testing: Kent C. Dodds' testing trophy. 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."



Which tools?

Static analysis detects syntax errors, bad practices and incorrect use of APIs: Code formatters, such as Prettier ;
Linters, such as ESLint; Type checkers, such as TypeScript and Flow.

Unit tests check that tricky algorithms are working properly. Tools: Jest.

Which tools?

Integration testing gives you the assurance that all your application's features work as intended. Tools: Jest and Enzyme or react-testing-library.

End-to-end testing ensures that your application works as a whole: frontend, backend, database and all the rest. Tools: Jest and Enzyme or react-testing-library: Cypress.

Introduction to Unit and Integration Testing

Unit testing: Validation of a unit of code (function or component) in isolation.
Integration testing: Validation of interaction between several modules/functions.

```
// Exemple d'un test unitaire
function add(a, b) {
  return a + b;
}

test('add function returns sum', () => {
  expect(add(1, 2)).toBe(3);
});

// Exemple d'un test d'intégration simulant une interaction entre API et UI
import { render, screen } from '@testing-library/react';
import App from './App';

test('displays fetched data', async () => {
  render(<App />);
  expect(await screen.findByText('Data loaded')).toBeInTheDocument();
});
```

UI testing vs. unit testing: a comparison

UI tests are expensive and slow, while unit tests are fast and inexpensive, ideal for testing isolated functions or components.

```
// Exemple de test unitaire avec Jest
test('add function', () => {
  expect(add(1, 2)).toBe(3);
});
```

Unit testing with Jest / vite

Jest is a tool for unit testing complex algorithms or React components.

```
// Test unitaire avec Jest
test('checks text content', () => {
  const { getByText } = render(<Button text="Click me!" />);
  expect(getByText(/click me/i)).toBeInTheDocument();
});
```

Integration testing

Integration tests cover entire functionalities or pages, offering a better return on investment than unit tests.

```
// Test avec React Testing Library
test('loads items eventually', async () => {
  const { getByText } = render(<ItemsList />);
  const item = await waitForElement(() => getByText('Item 1'));
  expect(item).toBeInTheDocument();
});
```

Understanding end-to-end testing with Cypress (E2E)

Cypress enables end-to-end testing by simulating real application use in a browser.

```
it('logs in successfully', () => {
  cy.visit('/login');
  cy.get('input[name=username]').type('user');
  cy.get('input[name=password]').type('password');
  cy.get('form').submit();
  cy.contains('Welcome, user!');
});
```

04

Vitest

Introducing Vitest

- **Description:** Vitest is a modern framework for JavaScript testing, inspired by Jest, fast and designed to be integrated with Vite.
- **Highlights:** ESM support, parallel testing, speed.

```
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"
  }
}
```

First test: what is a test?

A test verifies the expected behavior of a function or piece of code.

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

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

What is an assertion?

An assertion verifies that a condition is true. Vitest uses expect() to define assertions.

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

List of assertions in Vitest

List of the main assertions used in 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
```

Details on toBe vs toEqual

Difference between toBe and toEqual. toBe checks for strict equality (===), while toEqual compares the structure of objects and arrays.

```
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
```

Testing a utility function: multiply

Example of a test on a utility function.

```
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
});
```

Test grouping with describe

Use describe to organize tests by theme or function.

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

Testing asynchronous functions

How to test asynchronous functions in 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'
});
```

Using beforeEach and afterEach

beforeEach allows you to execute code before each test, and afterEach after each test. Very useful for initializing or cleaning up 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 HTTP services with Vitest

Use vi.mock() to simulate HTTP calls in tests, without having to actually call external services.

```
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
});
```

Using stubs to simulate behavior

Stubs can be used to temporarily replace a function to test different 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 asynchronous functions

How to create asynchronous function mocks with 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
});
```

Testing errors with toThrow()

Using the toThrow() assertion to test whether a function throws an error

```
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
});
```

Test management with parameterized values

Use test.each() to run the same test with different values.

```
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
  }
);
```

Using test.only to run a single test

test.only allows you to run a single test, useful for debugging.

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

Combine assertions to test several aspects

Example combining several assertions to check different parts of the 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 whole modules

Use vi.mock() to simulate entire modules, not just specific functions.

```
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
});
```

Writing tests for purely utilitarian functions

Utility functions can be tested in isolation, as they have no external dependencies.

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

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

Using spyOn to observe function calls

vi.spyOn() allows you to monitor calls to a function without replacing it.

```
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
});
```

Testing side effects

Test functions that modify an external state or environment, for example by modifying global variables or local storage.

```
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 with simulated timers

Use vi.useFakeTimers() to simulate delays or timers in 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
});
```

Time control with vi.advanceTimersByTime

Manually advance time to simulate delays in code execution.

```
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 external modules

Use vi.mock() to simulate external modules or libraries in 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
});
```

Performance testing with vi.measure

Use vi.measure() to measure the performance of a function or piece of 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 dates with vi.setSystemTime

Simulate specific dates and times to test time-dependent functions.

```
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 events with callbacks

Use mocks to simulate events or callbacks in 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
});
```

Interaction testing between multiple services

Test integration between several services or modules in an 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
});
```

Global state-dependent service testing

How to test services that depend on the application's global state or context.

```
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 external APIs into services

Simulate external API calls in service tests.

```
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
});
```

Test service error feedback

Check error handling in services by simulating failures.

```
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
});
```

Using toMatchObject to check partial objects

toMatchObject() allows you to partially check an object without requiring a complete match.

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

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

Running tests with Vitest

Run tests with Vitest via the command line and in watch mode.

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

Test execution with Coverage

How to generate test coverage reports with Vitest to visualize untested areas of the code.

```
vitest run --coverage
```

Test optimization with Vitest

Optimize test performance by enabling parallel tests and watch mode.

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

05

React Testing Library

React Testing Library: Introduction

React Testing Library (RTL) is a library focused on accessibility and user interaction testing. Key principle: Test components as a user interacts with them.

```
npm install @testing-library/react @testing-library/jest-dom
```

```
import { render, screen } from '@testing-library/react';
import App from './App';

test('renders hello world', () => {
  render(<App />);
  const linkElement = screen.getByText(/hello world/i);
  expect(linkElement).toBeInTheDocument();
});
```

Vitest and RTL installation and configuration

- How to configure a React project to use Vitest and RTL together.
- Installation of dependencies: Vitest, RTL, and JSDOM (test environment).

```
npm install vitest @testing-library/react jsdom --save-dev
```

```
import { defineConfig } from 'vitest/config';

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

Render and screen: what are they?

Explanation of render and screen methods in RTL.

```
render(<Component />); // Render the component into a virtual DOM
screen.getByText('text'); // Find an element by its text
```

Component rendering

Use React Testing Library's render() method to check component rendering.

```
import { render, screen } from '@testing-library/react';
import App from './App';

test('trouve le lien learn react', () => {
  render(<App />);
  const linkElement = screen.getByText(/learn react/i);
  expect(linkElement).toBeInTheDocument();
});
```

RTL search methods: getBy, queryBy, findBy

Differences between getBy, queryBy and findBy in RTL.

```
// getBy throws an error if element is not found
const button = screen.getByRole('button');

// queryBy returns null if element is not found
const optionalButton = screen.queryByRole('button');

// findBy is asynchronous and waits for the element to appear
const asyncButton = await screen.findByRole('button');
```

Testing components with props

How to test components that take props as input.

```
import { render, screen } from '@testing-library/react';
import MyComponent from './MyComponent';

test('renders with props', () => {
  render(<MyComponent title="My Title" />);
  expect(screen.getByText('My Title')).toBeInTheDocument(); // Verify the prop
});
```

Testing children's components with mocks

Mocking child components to isolate tests from parent components.

```
vi.mock('./ChildComponent', () => () => <div>Mocked Child</div>);

import ParentComponent from './ParentComponent';
import { render, screen } from '@testing-library/react';

test('renders parent with mocked child', () => {
  render(<ParentComponent />);
  expect(screen.getByText('Mocked Child')).toBeInTheDocument(); // Ensure the mocked
});
```

User interaction

Simulate user events with fireEvent in the React Testing Library.

```
import { render, fireEvent } from '@testing-library/react';
import Button from './Button';

test('clique sur le bouton', () => {
  const handleClick = jest.fn();
  const { getByText } = render(<Button onClick={handleClick}>Cliquez</Button>);
  fireEvent.click(getByText(/Cliquez/i));
  expect(handleClick).toHaveBeenCalledTimes(1);
});
```

Using userEvent to simulate interactions

Use userEvent to simulate complex interactions.

```
import { render, screen } from '@testing-library/react';
import userEvent from '@testing-library/user-event';

test('handles typing in input', async () => {
  render(<Input />);
  const input = screen.getByPlaceholderText('Type something');
  await userEvent.type(input, 'Hello, React!'); // Simulate typing in the input field
  expect(input.value).toBe('Hello, React!'); // Check if the input value is updated
});
```

Using fireEvent vs. userEvent

Differences between fireEvent and userEvent for simulating interactions.

```
// fireEvent triggers an immediate interaction
fireEvent.click(button);

// userEvent simulates a more realistic user interaction with delays
await userEvent.click(button);
```

Form input testing

Form input tests with fireEvent.change() to simulate text input.

```
import { render, fireEvent } from '@testing-library/react';
import Login from './Login';

test('saisit le nom d\'utilisateur', () => {
  const { getByLabelText } = render(<Login />);
  const input = getByLabelText(/nom d'utilisateur/i);
  fireEvent.change(input, { target: { value: 'john_doe' } });
  expect(input.value).toBe('john_doe');
});
```

Simulating form events

Simulate form events such as submit with fireEvent.

```
import { render, fireEvent } from '@testing-library/react';
import Login from './Login';

test('soumet le formulaire', () => {
  const handleSubmit = jest.fn();
  const { getByTestId } = render(<Login onSubmit={handleSubmit} />);
  fireEvent.submit(getByTestId('login-form'));
  expect(handleSubmit).toHaveBeenCalled();
});
```

Component testing with context

Test components that use the Context API by wrapping them in the context provider.

```
import { render } from '@testing-library/react';
import { UserProvider } from './UserContext';
import UserProfile from './UserProfile';

test('affiche le profil utilisateur', () => {
  const user = { name: 'John Doe' };
  const { getByText } = render(
    <UserProvider value={user}>
      <UserProfile />
    </UserProvider>
  );
  expect(getByText(/John Doe/i)).toBeInTheDocument();
});
```

Using act() for synchronous updates

Use of the act() method to ensure that all state and DOM updates are applied before assertions.

```
import { render, act } from '@testing-library/react';
import Counter from './Counter';

test('met à jour le compteur', () => {
  const { getByText } = render(<Counter />);
  const button = getByText(/increment/i);

  act(() => {
    fireEvent.click(button);
  });

  expect(getByText(/count: 1/i)).toBeInTheDocument();
});
```

Component testing with Redux

Tests components connected to Redux using the Redux provider.

```
import { render } from '@testing-library/react';
import { Provider } from 'react-redux';
import store from './store';
import App from './App';

test('rend le composant avec Redux', () => {
  const { getByText } = render(
    <Provider store={store}>
      <App />
    </Provider>
  );
  expect(getByText(/learn react/i)).toBeInTheDocument();
});
```

Testing a Redux action

Testing a simple action with Redux.

```
import counterReducer, { increment } from './counterSlice';

test('should handle increment', () => {
  const previousState = { value: 0 };
  expect(counterReducer(previousState, increment())).toEqual({ value: 1 });
});
```

Using Redux Toolkit

Introduction to Redux Toolkit to simplify Redux testing.

```
import { createSlice } from '@reduxjs/toolkit';

const counterSlice = createSlice({
  name: 'counter',
  initialState: { value: 0 },
  reducers: {
    increment: (state) => { state.value += 1; },
  },
});

export const { increment } = counterSlice.actions;
export default counterSlice.reducer;
```

```
import counterReducer, { increment } from './counterSlice';

test('should handle increment', () => {
  const previousState = { value: 0 };
  expect(counterReducer(previousState, increment())).toEqual({ value: 1 });
});
```

Testing the configured blind

Test blind behavior with Redux Toolkit.

```
test('should handle store actions', () => {
  store.dispatch(increment());
  expect(store.getState().counter.value).toBe(1);
});
```

Mocking Redux State with RTL

How to mock the Redux state in tests with RTL.

```
import { render } from '@testing-library/react';
import { Provider } from 'react-redux';
import { store } from './store';

const renderWithRedux = (ui, { initialState, store = createStore(reducer, initialState) } = {}) => {
  return {
    ...render(<Provider store={store}>{ui}</Provider>),
    store,
  };
};
```

```
const mockStore = {
  counter: { value: 42 },
};

const { getByText } = renderWithRedux(<Counter />, { initialState: mockStore });
expect(getByText(/42/i)).toBeInTheDocument();
```

Testing API calls with Redux and MSW

How to test the side effects of a Redux action with MSW and RTL.

```
import { fetchCounter } from './counterSlice';

test('fetch counter from API', async () => {
  const result = await store.dispatch(fetchCounter());
  expect(result.payload.value).toBe(42);
});
```

Test createAsyncThunk

How to test asynchronous thunks with Vitest.

```
import { createAsyncThunk } from '@reduxjs/toolkit';

export const fetchCounter = createAsyncThunk('counter/fetch', async () => {
  const response = await fetch('/api/counter');
  return response.json();
});
```

```
test('fetchCounter fulfills successfully', async () => {
  const result = await store.dispatch(fetchCounter());
  expect(result.type).toBe('counter/fetch/fulfilled');
  expect(result.payload.value).toBe(42);
});
```

Testing errors in asynchronous actions

Test errors and error states in asynchronous calls with Redux.

```
test('fetchCounter rejected with error', async () => {
  server.use(
    rest.get('/api/counter', (req, res, ctx) => {
      return res(ctx.status(500));
    })
  );
  const result = await store.dispatch(fetchCounter());
  expect(result.type).toBe('counter/fetch/rejected');
});
```

Introduction to Suspense

Suspense handles asynchronous rendering in React components. Let's see how to test it with Vitest and RTL.

```
import { Suspense } from 'react';

test('renders fallback during suspense', () => {
  const { getByText } = render(
    <Suspense fallback={<div>Loading...</div>}>
      <MyLazyComponent />
    </Suspense>
  );
  expect(getByText(/Loading/i)).toBeInTheDocument();
});
```

Complete test example with RTL

Test the rendering of a React component and user interactions.

```
// Button.js
const Button = ({ onClick, label }) => (
  <button onClick={onClick}>{label}</button>
);

// Button.test.js
import { render, fireEvent, screen } from '@testing-library/react';
import Button from './Button';

test('calls onClick when clicked', () => {
  const handleClick = vi.fn();
  render(<Button onClick={handleClick} label="Click Me" />);

  fireEvent.click(screen.getByText('Click Me'));
  expect(handleClick).toHaveBeenCalledTimes(1);
});
```

Simulating Asynchronous Requests with RTL

Manage asynchronous functions in tests with async/await and waitFor.

```
test('fetches and displays data', async () => {
  global.fetch = vi.fn(() =>
    Promise.resolve({
      json: () => Promise.resolve({ message: 'Hello World' }),
    })
  );

  render(<MyComponent />);
  expect(screen.getByText(/loading/i)).toBeInTheDocument();

  await waitFor(() => expect(screen.getByText(/hello world/i)).toBeInTheDocument());
});
```

```
await waitFor(() => expect(screen.getByText('Success')).toBeInTheDocument(), {
  timeout: 2000, // ici on attend jusqu'à 2 secondes avant que le test échoue
});
```

Test Error Management

Simulate component errors to test exception handling and asynchronous errors

```
const ErrorComponent = ({ fetchData }) => {
  const [error, setError] = React.useState(null);

  React.useEffect(() => {
    fetchData().catch((err) => setError(err.message));
  }, [fetchData]);

  if (error) return <div>Error: {error}</div>;
  return <div>Loading...</div>;
};

test('displays error message', async () => {
  const fetchData = vi.fn().mockRejectedValue(new Error('API Error'));

  render(<ErrorComponent fetchData={fetchData} />);
  await waitFor(() => expect(screen.getByText('Error: API Error')).toBeInTheDocument()
});
```

Testing Components with Contexts

Test components that use React.Context to share global data.

```
// MyContext.js
const MyContext = React.createContext();

// Component.js
const Component = () => {
  const value = React.useContext(MyContext);
  return <div>{value}</div>;
};

test('provides context value', () => {
  render(
    <MyContext.Provider value="Hello from context">
      <Component />
    </MyContext.Provider>
  );

  expect(screen.getByText('Hello from context')).toBeInTheDocument();
});
```

Testing Asynchronous Components and Hooks

Test components that use React hooks to manage asynchronous effects.

```
const useFetchData = () => {
  const [data, setData] = React.useState(null);

  React.useEffect(() => {
    fetch('/api/data')
      .then((res) => res.json())
      .then((data) => setData(data));
  }, []);

  return data;
};

test('tests a hook with async data', async () => {
  global.fetch = vi.fn(() => Promise.resolve({ json: () => Promise.resolve({ message:

  const { result, waitForNextUpdate } = renderHook(() => useFetchData());

  await waitForNextUpdate();
  expect(result.current).toEqual({ message: 'Success' });
});
```

06

E2E with Cypress

Introduction to Cypress

Cypress is a powerful end-to-end (E2E) testing tool designed for modern web applications.

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

Launch Cypress

Once installed, Cypress can be run in either interactive or headless mode.

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

Basic Cypress configuration

Configuration of Cypress in cypress.config.js to adapt test behavior (timeout, viewport, etc.).

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

First End-to-End (E2E) test

Write a simple test with Cypress that verifies the rendering of an application's home page.

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

Selecting elements with Cypress

Use cy.get() to select elements in the DOM by class, ID, attributes, etc.

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

Assertions with Cypress

Cypress uses Chai for assertions. Test element states, URLs, etc.

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

User interaction with Cypress

Simulate user actions such as click(), type(), clear().

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

Asynchronous testing with Cypress

Cypress automatically handles asynchronous actions. It waits until elements are available before executing subsequent actions.

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

Managing HTTP requests with cy.intercept()

Intercept and mock HTTP requests to simulate API responses.

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

Using Fixtures

Fixtures are JSON files or other formats that contain test data to simulate API responses.

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

Form testing with Cypress

Test forms by validating inputs, submissions and responses.

```
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');
```

URL and Navigation assertions

Check that the URL and navigation are correct after certain actions.

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

Cookies and Local Storage

Cypress lets you interact with cookies and local storage to manage user sessions.

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

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

Testing with Authentication

Manage authentication with Cypress, simulate user connections and sessions.

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

CI/CD integration

Integration of Cypress in a CI/CD pipeline such as GitHub Actions or 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
```

07

Mocks with MSW

Installing MSW

Install MSW to simulate network requests in integration tests.

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

Service Worker initialization

Generate the Service Worker script with the msw init command.

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

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

Starting the Service Worker

Start the Service Worker in development mode.

```
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

Create mock definitions for network requests.

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

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

08

Storybook tests

Write your first story

Create your first story for a React component to visualize it in Storybook.

```
import React from 'react';
import Button from './Button';

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

export const Primary = () => <Button primary label="Button" />;
```

Addon Knobs for manipulating props

Use the Knobs addon to dynamically interact with the properties of your components in Storybook.

```
// Import the React library and the Button component
import React from 'react';
import Button from './Button';

// Define metadata for the story
export default {
  title: 'Example/Button', // Title in Storybook's sidebar
  component: Button,       // The component being documented
};

// Define a story named 'Primary'
export const Primary = () => <Button primary label="Button" />;
// Renders the Button component with 'primary' prop and 'Button' label
```

Addon Actions to track events

Learn how to use the Actions addon to log component actions and events.

```
// Import the necessary modules for knobs
import React from 'react';
import { withKnobs, text } from '@storybook/addon-knobs';
import Button from './Button';

// Add the knobs decorator to the story
export default {
  title: 'Example/Button',
  component: Button,
  decorators: [withKnobs], // Enables knobs in this story
};

// Create a story with a dynamic label
export const DynamicLabel = () => (
  <Button label={text('Label', 'Button')} />
);
// Allows you to change the 'label' prop via the knobs panel
```

Addon Controls for manipulating props

Use the Controls addon to interact with your component props directly from Storybook's interface.

```
import React from 'react';
import { action } from '@storybook/addon-actions';
import Button from './Button';

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

export const Clickable = () => (
  <Button onClick={action('button-click')} label="Click Me" />
);
```

Static asset management

Learn how to include and manage static files such as images in your stories.

```
// Import an image file to use in your component
import logo from './assets/logo.png';

// Use the image in your component
const Logo = () => <img src={logo} alt="Logo" />;
// Displays the imported logo image
```

Using Storybook with Redux

Integrate Redux into your stories to manage the global state of your components.

```
import React from 'react';
import { Provider } from 'react-redux';
import store from './store';
import TodoList from './TodoList';

export default {
  title: 'Example/TodoList',
  component: TodoList,
  decorators: [(Story) => <Provider store={store}><Story /></Provider>],
};

export const Default = () => <TodoList />;
```

09

E2E with Playwright

What is Playwright?

Playwright is an end-to-end testing tool that lets you test web applications by simulating real-life user interactions.

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

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

Playwright benefits

Playwright offers an optimal testing and debugging experience, allows page inspection at any time, and supports multiple browsers.

```
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 offers a more consistent API, simpler configuration, multi-tab support and is faster than 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');
});
```

Playwright configuration file

Define global settings and projects for each browser.

```
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
```

Basic test with Playwright

Create a basic test to check the "welcome back" text.

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

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

Querying elements with Playwright

Use semantic selectors to query DOM elements.

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

Testing basic interactions

Test basic navigation and interaction.

```
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();
});
```

Testing forms with Playwright

Use locators to fill in and submit forms.

```
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();
});
```

Test links opening in a new tab

Check the href attribute or get the handle of the new page after clicking on the link.

```
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();
```

10

Jest

Writing a simple test with Jest

Structure of a simple test with Jest and how to run a test.

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

test('additionne 1 + 2 pour obtenir 3', () => {
  expect(sum(1, 2)).toBe(3);
});
```

Matchers Jest

Introduction to the various matchers available in Jest to check test results.

```
test('deux plus deux fait quatre', () => {
  expect(2 + 2).toBe(4);
});
```

Assertion tests

Use of basic assertions in Jest such as toBe, toEqual, toBeNull, toBeUndefined, and toBeDefined.

```
test('null est nul', () => {
  expect(null).toBeNull();
  expect(undefined).toBeUndefined();
  expect(1).toBeDefined();
});
```

Array and object testing

Check the contents of arrays and objects with the toContain and toHaveProperty matchers.

```
test('la liste contient le mot spécifique', () => {
  const shoppingList = ['couche', 'kleenex', 'savon'];
  expect(shoppingList).toContain('savon');
});
```

Exception testing

Test exceptions thrown by functions with the toThrow matcher.

```
function compilesAndroidCode() {
  throw new Error('Vous utilisez la mauvaise JDK');
}

test('compilation d\'Android génère une erreur', () => {
  expect(() => compilesAndroidCode()).toThrow('Vous utilisez la mauvaise JDK');
});
```

Testing asynchronous functions

Testing asynchronous functions with callbacks, promises and async/await.

```
test('les données de l\'API sont des utilisateurs', async () => {
  const data = await fetchData();
  expect(data).toEqual({ users: [] });
});
```

Jest Mock Functions

Create and use mock functions in Jest with jest.fn().

```
const myMock = jest.fn();
myMock();
expect(myMock).toHaveBeenCalled();
```

```
jest.mock('axios');
const axios = require('axios');

test('mocking axios', () => {
  axios.get.mockResolvedValue({ data: 'some data' });
  // test axios.get()
});
```

Running conditional tests

Run conditional tests with test.only and test.skip, and group tests with describe.

```
test.only('ce test est le seul à être exécuté', () => {
  expect(true).toBe(true);
});

test.skip('ce test est ignoré', () => {
  expect(true).toBe(false);
});

describe('groupe de tests', () => {
  test('test A', () => {
    expect(true).toBe(true);
  });

  test('test B', () => {
    expect(true).toBe(true);
  });
});
```