

React

тестирование

Что такое тестирование

Что такое пирамида тестирования

Что такое TDD

Кто такие QA и тестировщики

Jest

react-testing-library

Vitest

Что такое тестирование

Тестирование на фронтенде — это процесс проверки работы пользовательского интерфейса (UI) веб-приложений, чтобы убедиться, что оно функционирует правильно и соответствует требованиям. Оно охватывает различные аспекты, такие как проверка взаимодействия с элементами интерфейса, корректность отображения компонентов, а также функциональность на разных устройствах и браузерах.

Существует несколько типов тестирования на фронтенде:

- Юнит-тесты
- Интеграционные тесты
- Тесты пользовательского интерфейса (UI тесты)
- Тесты E2E (end-to-end)
- Регрессионные тесты

Юнит-тесты

Проверяют отдельные компоненты или функции приложения на предмет их корректной работы. Например, юнит-тесты могут проверить работу одной кнопки или функцию, отвечающую за обработку событий.

Интеграционные тесты

Проверяют взаимодействие между несколькими компонентами, чтобы убедиться, что они работают вместе правильно. Например, могут тестировать передачу данных между формой и сервером.

Тесты пользовательского интерфейса (UI тесты)

Ориентированы на проверку того, как элементы интерфейса отображаются и взаимодействуют с пользователем. Они могут включать проверку анимаций, взаимодействий при клике или проверку работы модальных окон.

Тесты E2E (end-to-end)

Имитируют поведение реального пользователя, проверяя весь пользовательский путь от начала до конца. Эти тесты могут включать переходы между страницами, отправку данных на сервер и обработку ответов.

Тесты пользовательского интерфейса (UI тесты)

Используются для проверки того, что недавние изменения в коде не привели к сбоям в уже работающем функционале. Для этого могут использоваться снимки состояния интерфейса (snapshot testing).

Итого

Тестирование помогает поддерживать стабильность приложения и уменьшает риск возникновения багов при развертывании новых версий.

Популярные инструменты для фронтенд-тестирования:

Jest — для юнит-тестов и тестов интеграции.

React Testing Library — для тестирования React-компонентов с фокусом на поведение пользователя.

Cypress — для тестов E2E.

Selenium — для тестирования взаимодействия с UI в браузере.

Тестирование на фронтенде необходимо по ряду причин:

Обеспечение корректной работы приложения: Тестирование помогает убедиться, что все компоненты пользовательского интерфейса работают правильно. Это особенно важно в крупных приложениях, где изменение одного компонента может случайно повлиять на другие.

Предотвращение регрессий: При внесении изменений или добавлении новых функций существует риск сломать уже работающие части приложения. Регрессионные тесты позволяют обнаружить такие проблемы на раннем этапе.

Ускорение разработки: Автоматические тесты позволяют разработчикам быстро проверять свой код, не полагаясь только на ручное тестирование. Это снижает вероятность возникновения багов и упрощает процесс рефакторинга.

Поддержка кросс-браузерной совместимости: Тестирование гарантирует, что приложение корректно работает во всех целевых браузерах и устройствах. Это особенно важно для приложений с широкой аудиторией.

Повышение качества кода: Написание тестов стимулирует разработчиков создавать более структурированный и качественный код. Хорошо протестированные компоненты, как правило, проще в сопровождении и изменении.

Ускорение поиска и устранения ошибок: Тесты помогают быстро находить ошибки и указывают на конкретные проблемные места. Это экономит время при отладке и снижает вероятность внесения новых ошибок при исправлении существующих.

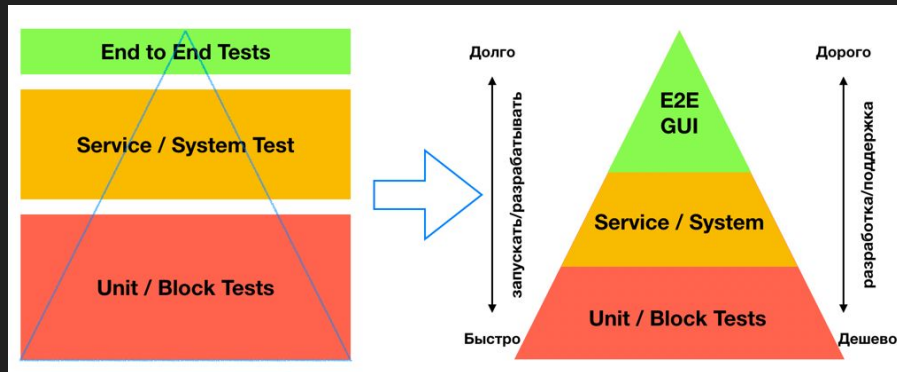
Документирование поведения: Тесты служат своеобразной документацией, которая описывает ожидаемое поведение приложения. Это полезно для новых разработчиков в команде, которым нужно быстро понять, как должен работать код.

Уверенность при развертывании: Перед развертыванием новой версии приложения тесты позволяют убедиться, что все критические функции работают, и снизить риск выпуска багов в продакшн.

Пирамида тестирования

Цель использования данного процесса:

- Экономия времени и ресурсов в процессах обеспечения качества
- Сократить количество сложных кейсов тестирования
- Группировка и типизация тестов
- Снижение рисков возникновения критических дефектов ПО
- Формализация тех. долга в части QA
- Формализация правил написания тестов
- Формализация требований к кодовой базе: code coverage, тестовые планы
- Автоматизация рутинных и частых операций
- Четко-регламентированная изоляция ресурсов в зависимости от вида тестирования



Пирамида тестирования — один из способов обеспечения качества ПО, визуализация, которая помогает группировать тесты по типу их назначения. Так же, позволяет согласовать правила написания тестов, разделения их на типы, обозначить основной фокус тестирования в каждой из групп.

Принцип пирамиды тестирования:

Основание пирамиды (юнит-тесты) должно быть самым широким: их должно быть больше всего, так как они дешевы в реализации и выполняются быстро.

Средний уровень (интеграционные тесты) должен быть шире, чем E2E-тесты, но уже, чем юнит-тесты. Они покрывают важные взаимодействия между компонентами системы.

Верхушка пирамиды (E2E-тесты) — самая узкая, поскольку они наиболее ресурсоемкие и медленные. E2E-тесты должны покрывать только ключевые пользовательские сценарии, чтобы не перегружать систему тестирования.

Преимущества пирамиды тестирования:

Баланс между скоростью и качеством:

Основная часть тестирования приходится на юнит-тесты, которые работают быстро, в то время как более сложные и медленные тесты (интеграционные и E2E) используются ограниченно.

Повышение устойчивости системы:

Широкое покрытие юнит-тестами помогает находить и исправлять баги на ранних этапах разработки, предотвращая ошибки в более сложных взаимодействиях.

Оптимизация ресурсов:

Пирамида помогает эффективно распределить время и ресурсы команды разработчиков, фокусируясь на автоматизации дешевых и быстрых тестов.

Ограничения пирамиды тестирования:

В некоторых случаях может быть сложнее применить этот подход для сложных или динамичных интерфейсов, где юнит-тесты не всегда могут отразить реальное поведение системы.

Пирамида тестирования — это ориентир, а не жесткое правило, и каждый проект может требовать уникального подхода к распределению тестов.

TDD (Test-Driven Development)

TDD (Test-Driven Development), или разработка через тестирование, — это подход к разработке программного обеспечения, при котором процесс написания кода начинается с создания тестов. Основная идея TDD заключается в том, чтобы сначала написать тест, который описывает желаемое поведение или функциональность системы, а затем — минимальный код, который проходит этот тест.



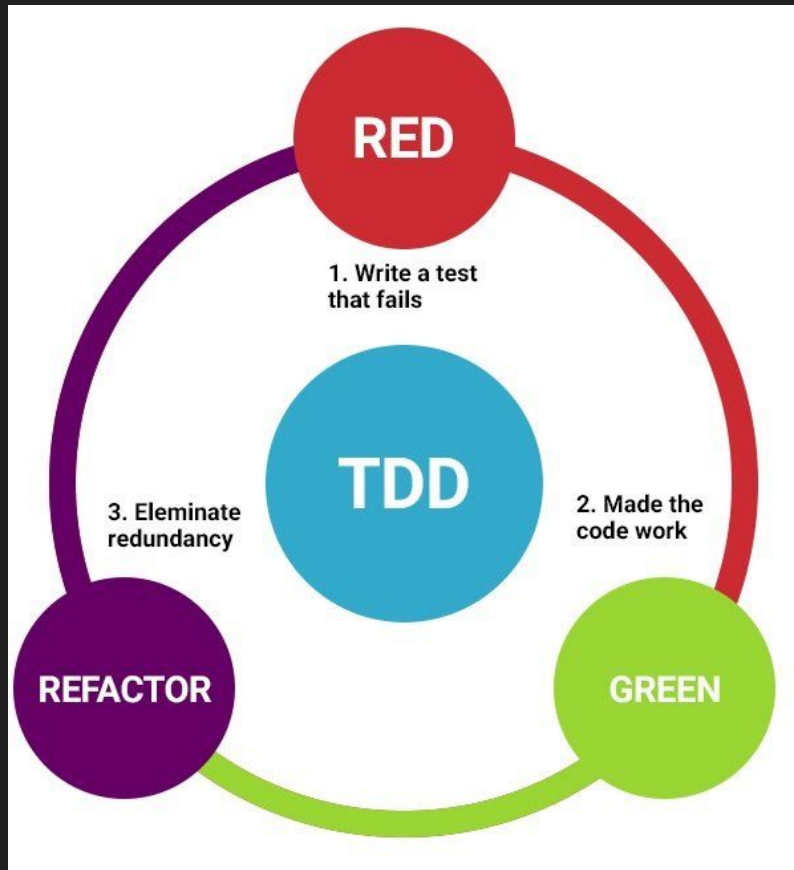
Этапы TDD

TDD состоит из трех ключевых этапов, известных как "красный — зелёный — рефакторинг":

Красный этап (Red): Сначала пишется тест, который проверяет поведение или функциональность, которую вы хотите реализовать. Этот тест будет провален, так как функциональность ещё не реализована.

Зелёный этап (Green): Затем пишется минимальное количество кода, необходимое для того, чтобы тест прошёл успешно. Код может быть не идеальным, главное — добиться прохождения теста.

Рефакторинг (Refactor): После того как тест пройден, производится рефакторинг кода, чтобы улучшить его структуру, оптимизировать или сделать более читабельным, при этом тесты должны продолжать проходить успешно.



Преимущества TDD:

Улучшенное качество кода: TDD стимулирует разработчиков писать модульный код, что упрощает его тестирование и сопровождение. Код становится более структурированным и надёжным.

Быстрое обнаружение ошибок: Благодаря тому, что тесты пишутся заранее, ошибки и баги могут быть обнаружены на ранних стадиях разработки, что снижает затраты на их исправление.

Уверенность при изменении кода: Имея набор тестов, разработчики могут быть уверены, что изменения или рефакторинг не сломают существующий функционал.

Автоматическая документация: Написанные тесты выполняют роль живой документации, объясняющей, как должен работать код. Это полезно для других членов команды, которые могут использовать тесты, чтобы понять поведение системы.

Фокус на требования: TDD помогает разработчикам сосредоточиться на реальных требованиях к системе, так как тесты пишутся с учетом ожидаемого поведения. Это снижает вероятность написания избыточного или ненужного кода.

Недостатки TDD:

Больше времени на начальном этапе: Процесс написания тестов перед кодом может показаться более медленным на старте проекта. Однако это компенсируется снижением числа багов и повышением стабильности в долгосрочной перспективе.

Не всегда подходит для UI: TDD может быть сложнее применять к разработке пользовательских интерфейсов, так как тестирование UI часто требует сложных сценариев и взаимодействий.

Тесты могут устаревать: Если требования к проекту часто меняются, тесты могут устаревать, и их нужно будет переписывать, что может занимать время.

Пример TDD

- 1) Написание теста:
На этом этапе тест будет красным, потому что функция add ещё не реализована.
- 2) Написание минимального кода для прохождения теста:
- 3) Рефакторинг (если требуется): Если код работает корректно, можно улучшить его, при этом тесты должны продолжать проходить.

TDD помогает построить более надежные системы, где поведение каждого компонента проверяется еще до того, как он становится частью основного кода.

```
test('add(2, 3) should return 5', () => {  
  expect(add(2, 3)).toBe(5);  
});
```

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

Кто такие QA и тестировщики

QA (Quality Assurance) и тестировщики — это роли в процессе обеспечения качества программного обеспечения, которые направлены на предотвращение и выявление дефектов. Хотя их задачи могут пересекаться, есть различия в их функциях и подходах.



Кто такой QA (специалист по обеспечению качества)?

QA (Quality Assurance) — это человек или команда, которые занимаются обеспечением качества продукта на всех этапах разработки. Их основная задача — выстраивать процессы, которые помогают предотвратить появление ошибок и дефектов. QA работает над улучшением методов разработки, создания документации, проведения тестов, и даже после выхода продукта на рынок контролирует соблюдение стандартов.

Основные задачи QA:

- Проектирование и улучшение процессов разработки.
 - QA специалисты следят за тем, чтобы в команде соблюдались стандарты качества.
 - Они разрабатывают и внедряют процессы тестирования и контроля качества.
- Создание документации по качеству.
 - QA часто создают и поддерживают документацию, которая описывает, как должны проводиться тесты, как управлять рисками и какие стандарты следует соблюдать.
- Предотвращение дефектов.
 - Вместо того чтобы искать ошибки постфактум, QA стремятся сделать так, чтобы они не возникли вовсе (например, с помощью улучшенных процессов разработки).
- Анализ метрик качества.
 - QA специалисты отслеживают показатели качества, такие как количество найденных дефектов, время выполнения тестов, производительность команды и так далее.

QA — это про процесс и предотвращение проблем, а не только про их исправление.

Кто такой тестировщик?

Тестировщик — это специалист, который непосредственно занимается тестированием программного обеспечения с целью выявления багов и ошибок. Тестировщики обычно выполняют конкретные тесты (ручные или автоматизированные), чтобы убедиться, что продукт работает корректно и соответствует требованиям.

Основные задачи тестировщика:

- Ручное тестирование:
 - Тестировщики проверяют функциональность продукта, выявляя ошибки, недоработки и дефекты.
 - Они могут тестировать как отдельные компоненты (модули), так и продукт в целом.
- Автоматизированное тестирование:
 - Многие тестировщики пишут скрипты для автоматизированного тестирования, чтобы ускорить и упростить проверку рутинных задач.
 - Например, проверка формы на ввод неверных данных или тестирование производительности.
- Создание тест-кейсов:
 - Тестировщики разрабатывают сценарии тестирования (тест-кейсы), которые описывают шаги для проверки определённых функций приложения.
- Регрессионное тестирование:
 - После исправления багов тестировщики проводят регрессионное тестирование, чтобы убедиться, что новые изменения не привели к появлению новых ошибок.
- Поддержка отчётности по багам:
 - Тестировщики создают отчёты об ошибках, которые помогают разработчикам исправлять проблемы.

Тестировщики фокусируются на выявлении существующих дефектов и проверке функциональности.

Основные различия между QA и тестировщиками:

QA (Обеспечение качества)	Тестировщик
Строит процессы для предотвращения дефектов.	Ищет ошибки в уже существующем продукте.
Отвечает за контроль качества на всех этапах разработки.	Работает с конкретным тестированием продукта.
Занимается документацией, метриками и анализом процессов.	Разрабатывает и выполняет тесты для проверки продукта.
Предотвращает проблемы до их появления.	Выявляет уже существующие проблемы.

React - тестирование

Тестирование в React — это процесс проверки работоспособности компонентов и их функциональности в приложении, разработанном с использованием библиотеки React. Оно помогает обнаружить ошибки, улучшить качество кода и упростить его поддержку.

Основные цели тестирования:

- **Проверка корректности:** Убедиться, что компоненты работают так, как задумано.
- **Регрессия:** Избежать повторного появления ошибок при внесении изменений в код.
- **Документация:** Тесты могут служить документацией для разработчиков, объясняя, как компоненты должны вести себя.



Основные библиотеки для тестирования react приложений:

- React-Testing-Library (RTL)
- JEST
- VITEST

Jest

Jest — это популярный фреймворк для тестирования JavaScript-приложений, который изначально был создан Facebook и отлично подходит для тестирования React-приложений, хотя может быть использован и для других JavaScript-проектов. Jest делает упор на простоту в использовании и предоставляет множество встроенных функций для модульного, интеграционного и даже end-to-end тестирования.

Основные возможности Jest:

- **Модульное тестирование:** Тестирование отдельных функций или компонентов в изоляции.
- **Мокирование (Mocking):** Возможность имитировать зависимости (функции, модули, таймеры) для изолирования тестируемого кода.
- **Снапшот-тестирование (Snapshot Testing):** Сравнение рендеренного результата (например, UI компонента) с эталонным (снимком) для обнаружения непреднамеренных изменений.
- **Тестирование асинхронного кода:** Поддержка промисов, коллбеков и `async/await`.
- **Покрывтие кода тестами:** Генерация отчетов о покрытии кода тестами.
- **Watch mode:** Запуск тестов в режиме просмотра, что позволяет автоматически перезапускать тесты при изменении кода.



Установка и настройка Jest

```
npm install jest --save-dev
```

Чтобы установить Jest в проекте, используйте npm или yarn

После установки можно добавить скрипт в package.json для запуска тестов:

Jest поддерживает файл конфигурации jest.config.js, где можно настроить дополнительные параметры

```
{  
  "scripts": {  
    "test": "jest"  
  }  
}
```

```
module.exports = {  
  ...  
  testEnvironment: 'jsdom', // среда выполнения тестов, полезна для тестирования React  
  setupFilesAfterEnv: ['./jest.setup.js'], // файлы настройки, которые запускаются перед каждым тестом  
  coverageDirectory: 'coverage', // директория для хранения отчетов о покрытии кода  
  collectCoverage: true, // включить сбор информации о покрытии  
  moduleNameMapper: {  
    '\\.(css|less)$': 'identity-obj-proxy' // маппинг файлов стилей для тестирования компонентов  
  }  
};
```

Пример Jest

```
// Counter.jsx
import React, { useState } from 'react';

export const Counter = () => {
  const [count, setCount] = useState(0);

  const increment = () => setCount(count + 1);
  const decrement = () => setCount(count - 1);

  return (
    <div>
      <h1>Counter: {count}</h1>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
};
```

```
// Counter.test.js
import React from 'react';
import ReactDOM from 'react-dom';
import { Counter } from './Counter';

describe('Counter Component', () => {
  let container;

  beforeEach(() => {
    container = document.createElement('div');
    document.body.appendChild(container);
  });

  afterEach(() => {
    document.body.removeChild(container);
  });

  test('initial render should show counter with 0', () => {
    ReactDOM.render(<Counter />, container);
    const counterText = container.querySelector('h1').textContent;
    expect(counterText).toBe('Counter: 0');
  });

  test('increments counter value when increment button is clicked', () => {
    ReactDOM.render(<Counter />, container);
    const incrementButton = container.querySelector('button');
    incrementButton.dispatchEvent(new MouseEvent('click', { bubbles: true }));
    const counterText = container.querySelector('h1').textContent;
    expect(counterText).toBe('Counter: 1');
  });

  test('decrements counter value when decrement button is clicked', () => {
    ReactDOM.render(<Counter />, container);
    const decrementButton = container.querySelectorAll('button')[1];
    decrementButton.dispatchEvent(new MouseEvent('click', { bubbles: true }));
    const counterText = container.querySelector('h1').textContent;
    expect(counterText).toBe('Counter: -1');
  });
});
```

RTL (React-Testing-Library)

React Testing Library — это популярная библиотека для тестирования компонентов React. Она предоставляет набор утилит, которые упрощают процесс написания тестов, ориентируясь на то, как пользователи взаимодействуют с приложением. Основная идея библиотеки заключается в том, чтобы тестировать компоненты так, как они используются на практике, вместо того чтобы сосредотачиваться на их внутренней реализации.



Установка:

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

Основные особенности React Testing Library:

Пользовательский опыт: Библиотека поощряет тестирование компонентов с точки зрения пользователя, что помогает создавать более надежные и стабильные приложения.

Простота использования: API библиотеки прост и интуитивно понятен, что позволяет быстро писать и поддерживать тесты.

Совместимость с другими инструментами: React Testing Library легко интегрируется с другими инструментами для тестирования, такими как Jest.

Поддержка различных методик: Она позволяет тестировать компоненты в различных состояниях, поддерживая асинхронные операции и различные взаимодействия с пользователем.

Пример Jest + RTL

```
// src/components/Counter.jsx
import React, { useState } from 'react';

const Counter = () => {
  const [count, setCount] = useState(0);

  const increment = () => setCount(count + 1);
  const decrement = () => setCount(count - 1);

  return (
    <div>
      <h1>Счетчик: {count}</h1>
      <button onClick={increment}>Увеличить</button>
      <button onClick={decrement}>Уменьшить</button>
    </div>
  );
};

export default Counter;
```

```
// src/__tests__/Counter.test.js
import React from 'react';
import { render, screen, fireEvent } from '@testing-library/react';
import Counter from '../components/Counter';

describe('Counter Component', () => {
  test('отображает начальное значение счетчика', () => {
    render(<Counter />);
    const counterText = screen.getByText(/счетчик: 0/i);
    expect(counterText).toBeInTheDocument();
  });

  test('увеличивает значение счетчика при нажатии на кнопку "Увеличить"', () => {
    render(<Counter />);
    const incrementButton = screen.getByRole('button', { name: /увеличить/i });

    fireEvent.click(incrementButton);

    const counterText = screen.getByText(/счетчик: 1/i);
    expect(counterText).toBeInTheDocument();
  });

  test('уменьшает значение счетчика при нажатии на кнопку "Уменьшить"', () => {
    render(<Counter />);
    const decrementButton = screen.getByRole('button', { name: /уменьшить/i });
    const incrementButton = screen.getByRole('button', { name: /увеличить/i });

    fireEvent.click(incrementButton); // увеличиваем на 1
    fireEvent.click(decrementButton); // уменьшаем на 1

    const counterText = screen.getByText(/счетчик: 0/i);
    expect(counterText).toBeInTheDocument();
  });
});
```

Vitest

Vitest — это современный фреймворк для тестирования, разработанный для работы с проектами, созданными с помощью Vite. Он предлагает высокую производительность и интеграцию с Vite, что делает его отличным выбором для тестирования приложений на React и других фреймворках. Vitest поддерживает такие функции, как тестирование компонентов, мокирование, снэпшоты и многое другое.



Пример Vitest + RTL

```
// src/components/Counter.jsx
import React, { useState } from 'react';

const Counter = () => {
  const [count, setCount] = useState(0);

  const increment = () => setCount(count + 1);
  const decrement = () => setCount(count - 1);

  return (
    <div>
      <h1>Счетчик: {count}</h1>
      <button onClick={increment}>Увеличить</button>
      <button onClick={decrement}>Уменьшить</button>
    </div>
  );
};

export default Counter;
```

```
// src/__tests__/Counter.test.js
import React from 'react';
import { render, screen, fireEvent } from '@testing-library/react';
import Counter from '../components/Counter';
import { describe, test, expect } from 'vitest';

describe('Counter Component', () => {
  test('отображает начальное значение счетчика', () => {
    render(<Counter />);
    const counterText = screen.getByText(/счетчик: 0/i);
    expect(counterText).toBeTruthy(); // Проверка наличия текста
  });

  test('увеличивает значение счетчика при нажатии на кнопку "Увеличить"', () => {
    render(<Counter />);
    const incrementButton = screen.getByText(/увеличить/i);

    fireEvent.click(incrementButton);

    const counterText = screen.getByText(/счетчик: 1/i);
    expect(counterText).toBeTruthy(); // Проверка увеличенного значения
  });

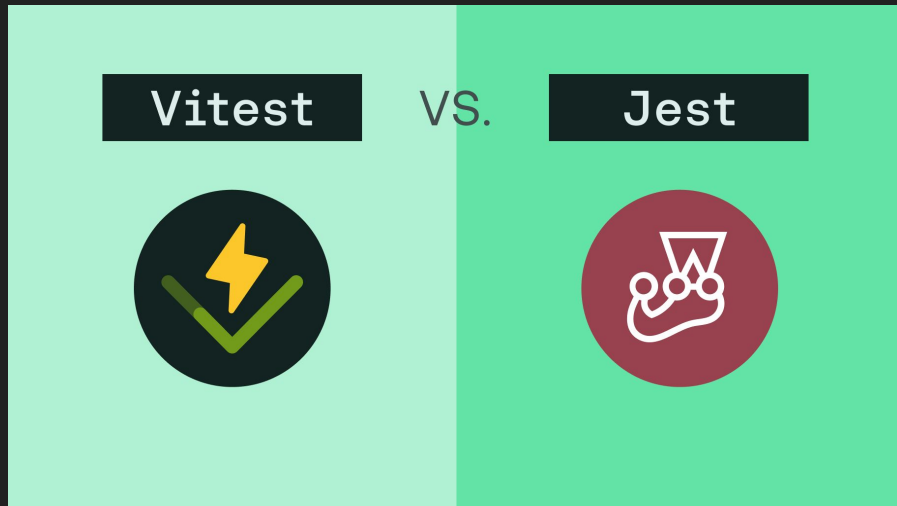
  test('уменьшает значение счетчика при нажатии на кнопку "Уменьшить"', () => {
    render(<Counter />);
    const incrementButton = screen.getByText(/увеличить/i);
    const decrementButton = screen.getByText(/уменьшить/i);

    fireEvent.click(incrementButton); // увеличиваем на 1
    fireEvent.click(decrementButton); // уменьшаем на 1

    const counterText = screen.getByText(/счетчик: 0/i);
    expect(counterText).toBeTruthy(); // Проверка уменьшенного значения
  });
});
```

Vitest vs Jest

Vitest является альтернативой Jest и был разработан с целью обеспечения более быстрой и эффективной среды для тестирования, особенно в проектах, созданных с использованием Vite. Вот несколько ключевых моментов, которые помогают понять, чем Vitest отличается от Jest и в каких случаях он может быть предпочтительным выбором:



Основные отличия Vitest и Jest

Производительность:

- Vitest использует подход, основанный на Vite, что позволяет ему значительно ускорить время запуска тестов за счет использования ES-модулей и приоритетного кэширования.
- Jest также быстро работает, но в проектах, использующих Vite, Vitest может быть еще быстрее благодаря более эффективному разрешению модулей.

Интеграция с Vite:

- Vitest оптимизирован для работы с Vite, что делает его идеальным выбором для проектов, построенных на этом инструменте. Он использует те же конфигурации и плагины, что и Vite.
- Jest может использоваться в проектах Vite, но требует дополнительных настроек и конфигураций для интеграции.

API и синтаксис:

- Оба инструмента имеют похожие API и предоставляют аналогичные функции для написания тестов. Например, в обоих случаях можно использовать `describe`, `test`, `expect` и т.д.
- Однако Vitest предлагает некоторые дополнительные функции и улучшения, такие как поддержка нативных ES-модулей.

Поддержка TypeScript:

- Vitest предоставляет отличную поддержку TypeScript из коробки, что может быть полезно для проектов, использующих TypeScript.
- Jest также поддерживает TypeScript, но требует дополнительной настройки.

Мокирование и шпионские функции:

- Vitest имеет встроенные функции для мокирования и шпионства, что делает его мощным инструментом для тестирования сложных приложений.
- Jest также предоставляет множество функций для мокирования, но Vitest предлагает более гибкий подход.

Что когда использовать

Когда использовать Vitest

- Если ваш проект уже использует Vite и вам нужно быстрое тестирование.
- Если вы хотите более современный инструмент, использующий нативные ES-модули.
- Если вы предпочитаете тестовый фреймворк, который легко интегрируется с вашими существующими конфигурациями Vite.

Когда использовать Jest

- Если ваш проект не использует Vite и вам нужен стабильный и проверенный инструмент для тестирования.
- Если у вас уже есть существующие тесты на Jest, и вы не хотите их переписывать.
- Если вам необходима поддержка дополнительных функций, которые предоставляет Jest, такие как встроенная поддержка снимков и более широкий экосистемный пакет.

Vitest — это мощная и быстрая альтернатива Jest, особенно для проектов на Vite. Оба инструмента имеют свои преимущества и недостатки, и выбор между ними зависит от конкретных требований вашего проекта и предпочтений команды разработчиков.

Ресурсы

Код урока (github репозиторий) - [ТЫК](#)

Статья про пирамиду тестирования - [ТЫК](#)

Статья про TDD - [ТЫК](#)

Документация Jest - [ТЫК](#)

Документация RTL - [ТЫК](#)

Документация Vitest - [ТЫК](#)