```
{() => fs}
```

Тестирование React apps

Существует множество различных способов тестирования приложений React. Давайте рассмотрим их далее.

Ранее в курсе использовалась библиотека Jest, разработанная Facebook для тестирования компонентов React. Теперь мы используем новое поколение инструментов тестирования от разработчиков Vite под названием Vitest. Помимо конфигураций, библиотеки предоставляют одинаковый программный интерфейс, поэтому в тестовом коде практически нет различий.

Давайте начнем с установки Vitest и библиотеки jsdom, имитирующей веб-браузер:

```
npm install --save-dev vitest jsdom
```

Копировать

В дополнение к Vitest нам также нужна другая библиотека тестирования, которая поможет нам отрисовывать компоненты для целей тестирования. На данный момент лучшим вариантом для этого является react-testing-library, популярность которой в последнее время стремительно растет. Также стоит расширить выразительные возможности тестов с помощью библиотеки jest-dom.

Давайте установим библиотеки с помощью команды:

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

Копировать

Прежде чем мы сможем провести первый тест, нам понадобятся некоторые конфигурации.

Мы добавляем скрипт в файл *package.json* для запуска тестов:

```
{
  "scripts": {
    // ...
    "test": "vitest run"
  }
  // ...
}
```

Копировать

Давайте создадим файл `testSetup.js` в корне проекта со следующим содержимым

```
import { afterEach } from 'vitest'
import { cleanup } from '@testing-library/react'
import '@testing-library/jest-dom/vitest'

afterEach(() => {
  cleanup()
})
```

Копировать

Теперь после каждого теста выполняется функция `cleanup` для сброса jsdom, которая имитирует работу браузера.

Разверните `vite.config.js` файл следующим образом

```
export default defineConfig({
  // ...
  test: {
    environment: 'jsdom',
    globals: true,
    setupFiles: './testSetup.js',
```

Копировать

```
  }
})
```

При использовании `globals: true` нет необходимости импортировать в тесты такие ключевые слова, как `describe`, `test` и `expect`.

Давайте сначала напишем тесты для компонента, который отвечает за рендеринг заметки:

```
const Note = ({ note, toggleImportance }) => {
  const label = note.important
    ? 'make not important'
    : 'make important'

  return (
    <li className='note'>
      {note.content}
      <button onClick={toggleImportance}>{label}</button>
    </li>
  )
}
```

Обратите внимание, что элемент *li* имеет значение *note* для CSS атрибута className, который может использоваться для доступа к компоненту в наших тестах.

Рендеринг компонента для тестов

Мы запишем наш тест в *файл src/components/Note.test.jsx*, который находится в том же каталоге, что и сам компонент.

В ходе первого теста проверяется, что компонент отображает содержимое заметки:

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

test('renders content', () => {
  const note = {
    content: 'Component testing is done with react-testing-library',
    important: true
  }

  render(<Note note={note} />)

  const element = screen.getByText('Component testing is done with react-testing-library')
  expect(element).toBeDefined()
})
```

После первоначальной настройки тест отрисовывает компонент с помощью функции render, предоставляемой библиотекой react-testing-library:

```
render(<Note note={note} />)
```

Обычно компоненты React визуализируются в DOM. Используемый нами метод визуализации визуализирует компоненты в формате, подходящем для тестов, без их рендеринга в DOM.

Мы можем использовать экран объекта для доступа к визуализируемому компоненту. Мы используем метод screen getByText для поиска элемента, содержащего содержимое заметки, и убедились, что он существует:

```
  const element = screen.getByText('Component testing is done with react-testing-library')
  expect(element).toBeDefined()
```

Существование элемента проверяется с помощью команды Vitest expect. Expect генерирует утверждение для своего аргумента, достоверность которого можно проверить с помощью различных функций определения условий. Теперь мы использовали toBeDefined, который проверяет, существует ли аргумент `элемента` expect.

Запустите тест с помощью команды `npm test`:

```
$ npm test

> notes-frontend@0.0.0 test
> vitest


 DEV  v1.3.1 /Users/mluukkai/opetus/2024-fs/part3/notes-frontend

 ✓ src/components/Note.test.jsx (1)
   ✓ renders content

 Test Files  1 passed (1)
```

```
      Tests  1 passed (1)
   Start at  17:05:37
   Duration  812ms (transform 31ms, setup 220ms, collect 11ms, tests 14ms, environment 395ms, prepare 70ms)

 PASS  Waiting for file changes...
```

Eslint жалуется на ключевые слова `test` и `expect` в тестах. Проблему можно решить, установив eslint-plugin-vitest-globals:

```
npm install --save-dev eslint-plugin-vitest-globals
```

Копировать

и включите плагин, отредактировав `.eslintrc.cjs` файл следующим образом:

```
module.exports = {
  root: true,
  env: {
    browser: true,
    es2020: true,
    "vitest-globals/env": true
  },
  extends: [
    'eslint:recommended',
    'plugin:react/recommended',
    'plugin:react/jsx-runtime',
    'plugin:react-hooks/recommended',
    'plugin:vitest-globals/recommended',
  ],
  // ...
}
```

Копировать

Расположение тестового файла

В React есть (как минимум) два разных соглашения о расположении тестового файла. Мы создали наши тестовые файлы в соответствии с текущим стандартом, поместив их в тот же каталог, что и тестируемый компонент.

Другое соглашение заключается в "обычном" хранении тестовых файлов в отдельном `тестовом` каталоге. Какое бы соглашение мы ни выбрали, оно почти гарантированно будет неправильным, согласно чьему-либо мнению.

Мне не нравится такой способ хранения тестов и кода приложения в одном каталоге. Мы решили следовать этому соглашению, потому что оно настроено по умолчанию в приложениях, созданных с помощью Vite или create-react-app.

Поиск содержимого в компоненте

Пакет react-testing-library предлагает множество различных способов исследования содержимого тестируемого компонента. На самом деле, `ожидать` в нашем тесте вообще не нужно:

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

test('renders content', () => {
  const note = {
    content: 'Component testing is done with react-testing-library',
    important: true
  }

  render(<Note note={note} />)

  const element = screen.getByText('Component testing is done with react-testing-library')

  expect(element).toBeDefined()
})
```

Копировать

Тест завершается неудачей, если `getByText` не находит элемент, который он ищет.

Мы могли бы также использовать CSS-селекторы для поиска визуализируемых элементов с помощью метода querySelector объекта container, который является одним из полей, возвращаемых визуализацией:

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

test('renders content', () => {
  const note = {
    content: 'Component testing is done with react-testing-library',
    important: true
  }
```

Копировать

```
  const { container } = render(<Note note={note} />)

  const div = container.querySelector('.note')
  expect(div).toHaveTextContent(
    'Component testing is done with react-testing-library'
  )
})
```

ПРИМЕЧАНИЕ: Более последовательным способом выбора элементов является использование атрибута данных, который специально определен для целей тестирования. Используя `react-testing-library`, мы можем использовать метод getByTestId для выбора элементов с указанным атрибутом `data-testid`.

Отладочные тесты

Обычно мы сталкиваемся со множеством различных проблем при написании наших тестов.

У объекта `screen` есть метод debug, который можно использовать для печати HTML-кода компонента в терминале. Если мы изменим тест следующим образом:

Копировать

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

test('renders content', () => {
  const note = {
    content: 'Component testing is done with react-testing-library',
    important: true
  }

  render(<Note note={note} />)

  screen.debug()

  // ...

})
```

HTML-код выводится на консоль:

Копировать

```
console.log
  <body>
    <div>
      <li
        class="note"
      >
        Component testing is done with react-testing-library
        <button>
          make not important
        </button>
      </li>
    </div>
  </body>
```

Также можно использовать тот же метод для печати требуемого элемента на консоли:

Копировать

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

test('renders content', () => {
  const note = {
    content: 'Component testing is done with react-testing-library',
    important: true
  }

  render(<Note note={note} />)

  const element = screen.getByText('Component testing is done with react-testing-library')

  screen.debug(element)

  expect(element).toBeDefined()
})
```

Теперь печатается HTML нужного элемента:

Копировать

```
  <li
    class="note"
  >
    Component testing is done with react-testing-library
```

```
    <button>
      make not important
    </button>
  </li>
```

Нажатие кнопок в тестах

Помимо отображения содержимого, компонент *Заметка* также гарантирует, что при нажатии кнопки, связанной с заметкой, вызывается функция обработки событий `toggleImportance`.

Давайте установим библиотеку user-event, которая немного упрощает имитацию пользовательского ввода:

```
npm install --save-dev @testing-library/user-event
```

Копировать

Тестирование этой функциональности может быть выполнено следующим образом:

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

// ...

test('clicking the button calls event handler once', async () => {
  const note = {
    content: 'Component testing is done with react-testing-library',
    important: true
  }

  const mockHandler = vi.fn()

  render(
    <Note note={note} toggleImportance={mockHandler} />
  )

  const user = userEvent.setup()
  const button = screen.getByText('make not important')
  await user.click(button)

  expect(mockHandler.mock.calls).toHaveLength(1)
})
```

Копировать

С этим тестом связано несколько интересных вещей. Обработчик событий представляет собой макет функции, определенной в Vitest:

```
const mockHandler = vi.fn()
```

Копировать

Для взаимодействия с визуализируемым компонентом запускается сеанс:

```
const user = userEvent.setup()
```

Копировать

Тест находит кнопку *по тексту* в отображаемом компоненте и нажимает на элемент:

```
const button = screen.getByText('make not important')
await user.click(button)
```

Копировать

Щелчок происходит с помощью метода click библиотеки UserEvent.

Ожидаемый результат теста использует toHaveLength для проверки того, что *имитационная функция* была вызвана ровно один раз:

```
expect(mockHandler.mock.calls).toHaveLength(1)
```

Копировать

Вызовы макетной функции сохраняются в массиве mock.calls внутри объекта макетной функции.

Макетные объекты и функции - это обычно используемые в тестировании заглушки компонентов, которые используются для замены зависимостей тестируемых компонентов. Макеты позволяют возвращать жестко запрограммированные ответы и проверять, сколько раз вызывались макетные функции и с какими параметрами.

В нашем примере функция mock является идеальным выбором, поскольку ее можно легко использовать для проверки того, что метод вызывается ровно один раз.

Тесты для *переключаемого* компонента

Давайте напишем несколько тестов для *переключаемого* компонента. Давайте добавим *togglableContent* имя_класса CSS в div, который возвращает дочерние компоненты.

```
const Togglable = forwardRef((props, ref) => {
  // ...

  return (
    <div>
      <div style={hideWhenVisible}>
        <button onClick={toggleVisibility}>
          {props.buttonLabel}
        </button>
      </div>
      <div style={showWhenVisible} className="togglableContent">
        {props.children}
        <button onClick={toggleVisibility}>cancel</button>
      </div>
    </div>
  )
})
```

Копировать

Тесты показаны ниже:

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

describe('<Togglable />', () => {
  let container

  beforeEach(() => {
    container = render(
      <Togglable buttonLabel="show...">
        <div className="testDiv" >
          togglable content
        </div>
      </Togglable>
    ).container
  })

  test('renders its children', async () => {
    await screen.findAllByText('togglable content')
  })

  test('at start the children are not displayed', () => {
    const div = container.querySelector('.togglableContent')
    expect(div).toHaveStyle('display: none')
  })

  test('after clicking the button, children are displayed', async () => {
    const user = userEvent.setup()
    const button = screen.getByText('show...')
    await user.click(button)

    const div = container.querySelector('.togglableContent')
    expect(div).not.toHaveStyle('display: none')
  })
})
```

Копировать

Функция `beforeEach` вызывается перед каждым тестированием, которое затем отображает *переключаемый* компонент и сохраняет поле `контейнера` возвращаемого значения.

Первый тест проверяет, что *Переключаемый* компонент отображает свой дочерний компонент

```
<div className="testDiv">
  togglable content
</div>
```

Копировать

В оставшихся тестах используется метод `toHaveStyle` для проверки того, что дочерний компонент *переключаемого* компонента изначально не виден, путем проверки того, что стиль элемента *div* содержит `{ display: 'none' }`. Другой тест проверяет, что при нажатии кнопки компонент становится видимым, что означает, что стиль для его скрытия *больше* не присваивается компоненту.

Давайте также добавим тест, который можно использовать для проверки того, что видимое содержимое может быть скрыто нажатием второй кнопки компонента:

```
describe('<Togglable />', () => {

  // ...

  test('toggled content can be closed', async () => {
    const user = userEvent.setup()
    const button = screen.getByText('show...')
    await user.click(button)

    const closeButton = screen.getByText('cancel')
    await user.click(closeButton)

    const div = container.querySelector('.togglableContent')
    expect(div).toHaveStyle('display: none')
  })
})
```

Тестирование форм

В предыдущих тестах мы уже использовали функцию `click` user-event для нажатия кнопок.

```
const user = userEvent.setup()
const button = screen.getByText('show...')
await user.click(button)
```

Мы также можем имитировать ввод текста с помощью *UserEvent*.

Давайте проведем тест для компонента *NoteForm*. Код компонента выглядит следующим образом.

```
import { useState } from 'react'

const NoteForm = ({ createNote }) => {
  const [newNote, setNewNote] = useState('')

  const handleChange = (event) => {
    setNewNote(event.target.value)
  }

  const addNote = (event) => {
    event.preventDefault()
    createNote({
      content: newNote,
      important: true,
    })

    setNewNote('')
  }

  return (
    <div className="formDiv">
      <h2>Create a new note</h2>

      <form onSubmit={addNote}>
        <input
          value={newNote}
          onChange={handleChange}
        />
        <button type="submit">save</button>
      </form>
    </div>
  )
}

export default NoteForm
```

Форма работает путем вызова функции, полученной в качестве реквизита `createNote` , с указанием деталей новой заметки.

Тест заключается в следующем:

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

test('<NoteForm /> updates parent state and calls onSubmit', async () => {
  const createNote = vi.fn()
  const user = userEvent.setup()

  render(<NoteForm createNote={createNote} />)
```

```
  const input = screen.getByRole('textbox')
  const sendButton = screen.getByText('save')

  await user.type(input, 'testing a form...')
  await user.click(sendButton)

  expect(createNote.mock.calls).toHaveLength(1)
  expect(createNote.mock.calls[0][0].content).toBe('testing a form...')
})
```

Тесты получают доступ к полю ввода с помощью функции getByRole.

Метод type UserEvent используется для ввода текста в поле ввода.

Ожидаемый результат первого тестирования гарантирует, что отправка формы вызовет метод `createNote`. Второе ожидание проверяет, вызывается ли обработчик события с правильными параметрами - создается ли заметка с правильным содержанием при заполнении формы.

Стоит отметить, что старая добрая `консоль.log` в тестах работает как обычно. Например, если вы хотите посмотреть, как выглядят вызовы, сохраненные макетным объектом, вы можете сделать следующее

```
test('<NoteForm /> updates parent state and calls onSubmit', async() => {
  const user = userEvent.setup()
  const createNote = vi.fn()

  render(<NoteForm createNote={createNote} />)

  const input = screen.getByRole('textbox')
  const sendButton = screen.getByText('save')

  await user.type(input, 'testing a form...')
  await user.click(sendButton)

  console.log(createNote.mock.calls)
})
```

Копировать

В середине выполнения тестов в консоли выводится следующее:

```
[ [ { content: 'testing a form...', important: true } ] ]
```

Копировать

О поиске элементов

Предположим, что форма содержит два поля для ввода

```
const NoteForm = ({ createNote }) => {
  // ...

  return (
    <div className="formDiv">
      <h2>Create a new note</h2>

      <form onSubmit={addNote}>
        <input
          value={newNote}
          onChange={handleChange}
        />
        <input
          value={...}
          onChange={...}
        />
        <button type="submit">save</button>
      </form>
    </div>
  )
}
```

Копировать

Теперь рассмотрим подход, который используется в нашем тесте для поиска поля ввода

```
const input = screen.getByRole('textbox')
```

Копировать

приведет к ошибке:

```
FAIL src/components/NoteForm.test.js
  ● <NoteForm /> updates parent state and calls onSubmit

    TestingLibraryElementError: Found multiple elements with the role "textbox"

    Here are the matching elements:

    Ignored nodes: comments, <script />, <style />
    <input
      value=""
    />

    Ignored nodes: comments, <script />, <style />
    <input />

    (If this is intentional, then use the `*AllBy*` variant of the query (like
queryAllByText`, `getAllByText`, or `findAllByText`)).
```

В сообщении об ошибке предлагается использовать *getAllByRole*. Тест можно было бы исправить следующим образом:

```
const inputs = screen.getAllByRole('textbox')

await user.type(inputs[0], 'testing a form...')
```

Метод *getAllByRole* теперь возвращает массив, а правое поле ввода является первым элементом массива. Однако такой подход немного подозрителен, поскольку он основан на порядке расположения полей ввода.

Довольно часто поля ввода содержат текст-*заполнитель*, который подсказывает пользователю, какой тип ввода ожидается. Давайте добавим заполнитель в нашу форму.:

```
const NoteForm = ({ createNote }) => {
  // ...

  return (
    <div className="formDiv">
      <h2>Create a new note</h2>

      <form onSubmit={addNote}>
        <input
          value={newNote}
          onChange={handleChange}
          placeholder='write note content here'
        />
        <input
          value={...}
          onChange={...}
        />
        <button type="submit">save</button>
      </form>
    </div>
  )
}
```

Теперь легко найти нужное поле ввода с помощью метода getByPlaceholderText:

```
test('<NoteForm /> updates parent state and calls onSubmit', () => {
  const createNote = vi.fn()

  render(<NoteForm createNote={createNote} />)

  const input = screen.getByPlaceholderText('write note content here')
  const sendButton = screen.getByText('save')

  userEvent.type(input, 'testing a form...')
  userEvent.click(sendButton)

  expect(createNote.mock.calls).toHaveLength(1)
  expect(createNote.mock.calls[0][0].content).toBe('testing a form...')
})
```

Наиболее гибким способом поиска элементов в тестах является метод *querySelector* объекта- `контейнера` , который возвращается с помощью `рендеринга` , как упоминалось ранее в этой части . С помощью этого метода можно использовать любой CSS-селектор для поиска элементов в тестах.

Рассмотрим, например. что мы бы определили уникальный `id` для поля ввода:

```
const NoteForm = ({ createNote }) => {
  // ...

  return (
    <div className="formDiv">
      <h2>Create a new note</h2>

      <form onSubmit={addNote}>
        <input
          value={newNote}
          onChange={handleChange}
          id='note-input'
        />
        <input
          value={...}
          onChange={...}
        />
        <button type="submit">save</button>
      </form>
    </div>
  )
}
```

Теперь элемент input можно найти в тесте следующим образом:

```
const { container } = render(<NoteForm createNote={createNote} />)

const input = container.querySelector('#note-input')
```

Тем не менее, мы будем придерживаться подхода использования `getByPlaceholderText` в тестировании.

Давайте рассмотрим пару деталей, прежде чем двигаться дальше. Предположим, что компонент будет отображать текст в HTML-элементе следующим образом:

```
const Note = ({ note, toggleImportance }) => {
  const label = note.important
    ? 'make not important' : 'make important'

  return (
    <li className='note'>
      Your awesome note: {note.content}
      <button onClick={toggleImportance}>{label}</button>
    </li>
  )
}

export default Note
```

метод `getByText` , используемый в тестировании, *не* находит элемент

```
test('renders content', () => {
  const note = {
    content: 'Does not work anymore :(',
    important: true
  }

  render(<Note note={note} />)

  const element = screen.getByText('Does not work anymore :(')

  expect(element).toBeDefined()
})
```

Метод `getByText` ищет элемент, который содержит **тот же текст**, что и в качестве параметра, и ничего более. Если мы хотим найти элемент, *содержащий* текст, мы могли бы использовать дополнительную опцию:

```
const element = screen.getByText(
  'Does not work anymore :(', { exact: false }
)
```

или мы могли бы использовать метод `findByText`:

```
const element = await screen.findByText('Does not work anymore :(')
```

Копировать

Важно заметить, что, в отличие от других методов `ByText`, `findByText` возвращает обещание!

Бывают ситуации, когда полезна еще одна форма метода `queryByText`. Метод возвращает элемент, но *он не вызывает исключения*, если он не найден.

Мы могли бы, например. использовать метод, чтобы гарантировать, что что-то *не отображается* в компоненте:

```
test('does not render this', () => {
  const note = {
    content: 'This is a reminder',
    important: true
  }

  render(<Note note={note} />)

  const element = screen.queryByText('do not want this thing to be rendered')
  expect(element).toBeNull()
})
```

Копировать

Охват тестированием

Мы можем легко узнать охват наших тестов, запустив их с помощью команды.

```
npm test -- --coverage
```

Копировать

При первом запуске команды Vitest спросит вас, хотите ли вы установить требуемую библиотеку `@vitest/coverage-v8`. Установите ее и снова запустите команду:

```
 % Coverage report from v8
-------------------|---------|----------|---------|---------|-------------------
File               | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-------------------|---------|----------|---------|---------|-------------------
All files          |   22.92 |    55.55 |   41.66 |   22.92 |
 src               |       0 |        0 |       0 |       0 |
  App.jsx          |       0 |        0 |       0 |       0 | 1-150
  main.jsx         |       0 |        0 |       0 |       0 | 1-5
 src/components    |   51.28 |    71.42 |    62.5 |   51.28 |
  Footer.jsx       |       0 |        0 |       0 |       0 | 1-16
  LoginForm.jsx    |       0 |        0 |       0 |       0 | 1-44
  Note.jsx         |     100 |       50 |     100 |     100 | 3
  NoteForm.jsx     |     100 |      100 |     100 |     100 |
  Notification.jsx |       0 |        0 |       0 |       0 | 1-13
  Togglable.jsx    |    92.3 |      100 |     100 |    92.3 | 15-17
 src/services      |       0 |        0 |       0 |       0 |
  login.js         |       0 |        0 |       0 |       0 | 1-9
  notes.js         |       0 |        0 |       0 |       0 | 1-29
-------------------|---------|----------|---------|---------|-------------------
```

Для каталога *coverage* будет сгенерирован отчет в формате HTML. В отчете будут указаны строки непроверенного кода в каждом компоненте.:

Вы можете найти код для нашего текущего приложения полностью в *части 5-8* в этом репозитории на GitHub.

Упражнения 5.13.-5.16.

5.13: Тесты списка блогов, шаг 1

Создайте тест, который проверяет, что компонент, отображающий блог, показывает название блога и его автора, но по умолчанию не показывает URL-адрес или количество лайков.

При необходимости добавьте CSS-классы в компонент, чтобы облегчить тестирование.

5.14: Тесты списка блогов, шаг 2

Проведите тест, который проверяет, отображаются ли URL блога и количество лайков при нажатии кнопки, управляющей отображаемыми сведениями.

5.15: Тесты списка блогов, шаг 3

Проведите тест, который гарантирует, что при двойном нажатии кнопки "*Нравится*" обработчик событий, полученный компонентом в качестве реквизита, вызывается дважды.

5.16: Тесты списка блогов, шаг 4

Проведите тест для новой формы блога. Тест должен проверить, вызывает ли форма обработчик события, полученный ею в качестве реквизита, с нужными данными при создании нового блога.

Тесты интеграции с внешним интерфейсом

В предыдущей части материала курса мы написали интеграционные тесты для серверной части, которые тестировали ее логику и подключали базу данных через API, предоставляемый серверной частью. При написании этих тестов мы приняли сознательное решение не писать модульные тесты, поскольку код для этого серверного модуля довольно прост, и вполне вероятно, что ошибки в нашем приложении возникают в более сложных сценариях, для которых модульные тесты не подходят.

До сих пор все наши тесты для интерфейса представляли собой модульные тесты, которые подтверждали правильность функционирования отдельных компонентов. Модульное тестирование иногда полезно, но даже полного набора модульных тестов недостаточно для проверки работоспособности приложения в целом.

Мы также могли бы проводить интеграционные тесты для интерфейса. Интеграционное тестирование проверяет совместную работу нескольких компонентов. Это значительно сложнее, чем модульное тестирование, поскольку нам пришлось бы, например, моделировать данные с сервера. Мы решили сосредоточиться на создании комплексных тестов для тестирования всего приложения. Мы будем работать над комплексными тестами в последней главе этой части.

Тестирование моментальных снимков

Vitest предлагает совершенно иную альтернативу "традиционному" тестированию, называемому snapshot тестирование. Интересной особенностью моментального тестирования является то, что разработчикам не нужно самим определять какие-либо тесты, внедрить моментальное тестирование достаточно просто.

Фундаментальный принцип заключается в сравнении HTML-кода, определенного компонентом после его изменения, с HTML-кодом, который существовал до его изменения.

Если моментальный снимок замечает некоторые изменения в HTML, определенном компонентом, то это либо новая функциональность, либо "ошибка", вызванная случайностью. Тесты моментальных снимков уведомляют разработчика об изменении HTML-кода компонента. Разработчик должен сообщить Vitest, было ли изменение желательным или нежелательным. Если изменение HTML-кода является неожиданным, это однозначно указывает на ошибку, и разработчик может легко узнать об этих потенциальных проблемах благодаря тестированию моментальных снимков.

Предлагайте изменения в материале

Часть 5b
Предыдущая часть

Часть 5d
Следующая часть

About course

Course contents

FAQ

Partners

Challenge

HOUSTON