**Что такое тестирование**

В разработке ПО тестирование — способ проверить код на работоспособность и устойчивость в различных ситуациях. А ещё — возможность удостовериться, что все элементы системы работают так, как должны, и не работают так, как не должны.

С помощью тестов вы можете проверить результат выполнения функции при разных входных параметрах, отладить обработку ошибок и удостовериться, что модули приложения правильно взаимодействуют между собой. Также тесты помогают разработчику найти ошибки и баги, которые в прочих условиях было бы сложно обнаружить.

**Зачем тестировать приложения**

Вводная часть может показаться сложной, но по мере продвижения внутрь темы, всё станет понятнее. Тестирование в первую очередь нужно для того, чтобы сэкономить время разработчика, команды и бизнеса.

Вот, как тестирование экономит время:

* разработчику не нужно делать дополнительную ручную проверку;
* благодаря тому, что разработчики тщательно продумывают структуру компонента, который потребуется покрывать тестами, появляется возможность быстро отслеживать его работу при внесении изменений. Это особенно эффективно, когда в команде появляются новые разработчики.

Чтобы тесты были полезными, их нужно правильно применять. Если бездумно покрывать код тестами, это может привести к захламлению проекта, замедлит разработку и сделает её неудобной. Об этом мы поговорим в отдельном небольшом уроке.

**Виды тестирования**

Тестирование бывает ручное и автоматическое. С ручным тестированием сталкивается каждый разработчик. К примеру, когда вы написали "Hello world!" на новом языке, скомпилировали и запустили код — он был проверен на работоспособность. Получается, был проведён процесс тестирования.

Но приложения состоят из множества компонентов и тестировать вручную каждую конструкцию — не самая интересная задача (а ещё и достаточно времязатратная). Поэтому программисты придумали писать код, который проверяет другой код, и назвали всё это автоматическим тестированием, или автотестами.

Подробнее об этом — в следующих уроках.

**Подходы к тестированию**

Существует множество подходов к тестированию. Часто коллективы разработчиков приходят к собственным методам, которые удобно применять в конкретной команде и на конкретном проекте. Мы выделили несколько основных методологий:

* TDD — Test Driven Development;
* TLD — Test Last Development;
* BDD — Behaviour Driven Development.

Рассмотрим, в чём заключается смысл каждой из них.

**Test Driven Development**

Это подход к тестированию, когда тест пишется до продуктового кода. Например, у разработчика есть функциональность добавления комментариев к фотографии. В TDD методологии разработчик сначала напишет тест, который проверит корректность добавления комментариев с тестовыми данными. И только после этого — компоненты, которые покрывают эту функциональность и удовлетворяют написанному тесту. TDD можно представить в виде нескольких последовательных шагов:

1. Написание теста для новой функциональности.
2. Проверка теста, который должен «падать» при запуске.
3. Написание минимального рабочего кода компонента для успешного прохождения теста.
4. Запуск тестов и исправление кода до тех пор, пока тесты функциональности не пройдут успешно.
5. Насыщение кода компонента дополнительной функциональностью.
6. Проверка прохождения тестов.
7. Переход к написанию следующего теста для новой функциональности.

В основе TDD подхода лежит "black box" — игнорирование внутреннего устройства компонента. Другими словами, разработчик ещё не знает, с какими данными предстоит работать внутри компонента, но уже пишет тест на основе требуемой функциональности. По мнению практикующих TDD разработчиков, это позволяет обнаруживать те баги, которые тяжело было бы найти в уже написанном коде компонента.

Некоторые разработчики считают, что TDD хорошо сказывается на устойчивости компонента и архитектуре приложения в целом. А поддержка приложений с TDD подходом отличается удобством, ведь почти каждый компонент покрыт тестами.

Про оптимальность TDD спорят до сих пор. Но кое-что мы можем отметить однозначно — у этого подхода невероятно высокий порог вхождения. При написании тестов до появления продуктового кода приходится работать с абстракциями. Новички в TDD могут потратить уйму времени на такую работу. А это полностью противоречит концепции экономии времени при использовании автотестов.

**Test Last Development**

В случае с TLD сначала разрабатывается функциональность, а тесты пишутся в самом конце. Этот подход противопоставляется TDD и предполагает взаимодействие с «white box». Это значит, что компонент или его часть уже написаны, и у разработчика есть представление о данных и особенностях работы компонента.

Вернёмся к примеру с функциональностью добавления комментария. Разработчик уже написал компонент и знает, какие данные о пользователе (id, имя, текст комментария и т. д.) используются при добавлении комментария в компоненте. После этого разработчик пишет тест к этой функциональности.

В TLD подходе очень низкий порог вхождения (в отличие от TDD). Достаточно ознакомиться с инструментами, их концепциями и методами, и можно приступить к покрытию функциональности тестами. С таким подходом не нужно представлять в голове код и компоненты, которые будут реализовывать функциональность.

Ещё один большой плюс TLD — лёгкость внедрения. Вы можете длительное время разрабатывать продукт без применения автотестов, а когда возникнет потребность, — внедрить тестирование в рабочие процессы и покрыть тестами любые модули.

Также TLD хорошо показывает себя внутри небольших циклов разработки. Например, вы за 20-30 минут добавили новый небольшой компонент и сразу покрыли его тестами. В таком случае TLD-тесты получаются не очень объёмными, а на их написание уходит мало времени. А для циклов разработки побольше TLD работает хуже. Если же добавлять по 3-4 больших компонента и только потом приступать к написанию тестов, то высок риск просто перейти к работе над следующими компонентами и проигнорировать написание тестов для только что созданных.

TLD хорошо сказывается на стабильности приложения и качестве рефакторинга, но у подхода есть и недостатки. При TLD разработчик опирается на уже написанный код. Из-за этого можно не увидеть некоторые варианты развития событий и неожиданные типы данных, а это может привести к ошибкам. Но это палка о двух концах: если на руках уже есть существующий код, можно протестировать сложные краевые кейсы, которые в TDD подходе вообразить было бы намного сложнее.

**Behavior Driven Development**

BDD — это разновидность TDD подхода, которая проверяет поведение приложения, а не входные данные и особенность реализации. Разберём BDD подход на примере простого счётчика кликов: нажатие на кнопку увеличивает значение счётчика на 1.

Для разработчика, практикующего TDD подход, эта функциональность будет интерпретирована в тест так:

1. Входные данные: значение счётчика 0.
2. Клик по кнопке и увеличение значение счётчика на 1.
3. Проверка получившихся данных — ожидается значение 1.

В нашем примере поведение — увеличение счётчика на 1 по клику. Так что для разработчика, который практикует BDD подход, функциональность счётчика будет интерпретирована в тест по-другому:

1. Клик по кнопке увеличивает значение счётчика на 1.
2. Разница между предыдущим и новым значением счётчика составляет 1.

То есть BDD методология основана на проверке ожидаемого и реального пользовательского опыта. Это набор сценариев, которые пользователь проделывает в приложении.

BDD-тест можно описать так:

* исследуются пользовательские сценарии использования продукта,
* сценарии фиксируются в текстовом формате:

Скопировать код

\*\*Если\*\* пользователь нажал на значок корзины

\*\*то\*\* должен произойти переход на страницу корзины

* сценарии использования приложения эмулируются с помощью автоматических тестов.

Такой подход позволяет протестировать функциональность всего приложения. BDD максимально близок к пользователю, ведь направлен исключительно на проверку сценариев использования приложения.

У этого подхода тоже есть минусы: более сложная реализация тестов, освоение отдельного фреймворка исполнения автотестов, а также низкая скорость работы таких тестов.

К сожалению, мы не дадим однозначного ответа на вопрос, какой из методов лучше. Это большой холивар внутри сообщества разработчиков, поэтому мы выдерживаем нейтралитет. В рамках проектной кухни или проектной работы вы можете использовать тот подход, который сочтёте нужным. А внутри команды, в которой вы будете работать или уже работаете, скорее всего уже есть мнение на этот счёт.

Каждую из этих методологий можно использовать как самостоятельный способ тестирования приложения. Но в зависимости от потребностей можно применять и сразу несколько подходов. Например, особенно важную логику на бэкенде можно спроектировать с применением TDD. А функциональность, которую необходимо добавить быстро (или которая вводится в качестве эксперимента), можно протестировать после создания с использованием TLD. Общую функциональность приложения с основными сценариями использования — тестировать с помощью BDD.

В следующем уроке разберём, какие виды автоматических тестов существуют.

**Разновидности автотестов**

Существует множество видов автоматического тестирования, которые применяются в разных областях и разными специалистами. В этом уроке разберём три основных, которые будут полезны в арсенале веб-разработчика:

* модульные,
* интеграционные,
* функциональные.

**Модульные тесты**

Модульные, или юнит-тесты (англ. Unit tests), применяются для изолированного тестирования модулей программы, например функций или компонентов.

С помощью юнит-тестов можно проверить, правильно ли работает компонент при различных входных параметрах. Разработчик тестирует оторванный от остальных компонентов кусочек кода. Основная задача — проверить, может ли компонент или функция существовать изолированно.

Такие тесты помогают разработчикам получать обратную связь непосредственно в процессе написания кода: юнит-тесты можно запускать при проверке кода, например линтером.

Примером юнит-теста может служить проверка работы компонента Avatar в нашем приложении с чатом. Всё, что нам нужно проверить, — корректно ли исполняется код компонента:

Скопировать кодJSX

export const Avatar = ({ name }) => {

return (

<div className={styles.avatar}>

<div className={styles.initials}>{name[0]}</div>

</div>

);

};

В приложении может быть много юнит-тестов, и они могут запускаться очень часто. Важно, чтобы эти тесты были простыми, быстро выполнялись, и в случае неудачи были достаточно информативными. Простота тестов влияет на их поддержку и скорость выполнения. А если тесты упали (а их может быть очень много) — разработчик должен получить качественный лог выполнения с такой информацией:

* название конкретного компонента,
* ожидаемый результат,
* ожидаемое поведение,
* действительный результат.

**Интеграционные тесты**

Интеграционные тесты применяются для проверки работы нескольких связанных компонентов. Такие тесты похожи на модульные, но в интеграционных компоненты не изолированы. Причём связанными компонентами может быть как фронтенд, так и бэкенд части приложения.

Для примера рассмотрим страницу профиля пользователя. Изолированное тестирование компонента не будет достаточно информативным, ведь большая часть данных приходит с сервера. При попадании на страницу происходит запрос данных по \_id пользователя, и с сервера возвращается информация о пользователе. В этом случае проверка данных от сервера и сравнение с эталоном будет полезнее — так мы проверим обе части приложения: фронтенд и бэкенд.

Обратите внимание, что интеграционные тесты часто используют обращение к серверу, поэтому не работают так же быстро, как юнит-тесты. Использовать их каждый раз при запуске, к примеру линта приложения, — дорогая операция.

**Функциональные тесты**

Функциональные тесты, или сквозные тесты, или E2E тесты (англ. End to End), применяются для того, чтобы проверить, как ведёт себя приложение с точки зрения пользовательского опыта. Это одна из форм интеграционных тестов. E2E тест содержит входные данные, которые подаются в приложение (или его часть), и выходные данные после действий пользователя. Выходные данные сравниваются с ожидаемым результатом — это и есть результат выполнения теста.

Функциональные тесты хорошо подойдут, скажем, для проверки процесса регистрации пользователя. В тест страницы регистрации подаются ожидаемые данные, например почта, пароль и имя. После этого происходит попытка отправки формы регистрации: отправляемые данные сравниваются с эталоном.

С видами тестов разобрались. В следующем уроке разберём инструменты для тестирования.

# Инструменты тестирования

В предыдущих уроках вы узнали про виды и подходы к тестированию. Но у тестов должна быть собственная инфраструктура, которая запускает тест, визуализирует его процесс и отображает результат тестирования. Инструментов тестирования очень много. В этом уроке мы устроим кастинг и выберем подходящие для наших задач.

## Инструмент запуска тестов

Сам по себе файл теста не запустится. Для запуска теста и его исполнения существуют инструменты — тест-раннеры. Это фреймворки, которые запускают тесты в определённой среде. О них поговорим в следующем уроке. Но тест-раннер нам однозначно понадобится.

## Инструменты создания моков и рендереров

Что-то новенькое и наверняка полезное. Подробнее про моки и рендереры мы поговорим уже при написании тестов. А сейчас важно то, что это тестовые данные или эмулятор работы API.

Представьте, что вам нужно написать юнит-тест компонента, который завязан на работе с сервером. Писать для теста целый API и пытаться эмулировать работу fetch — слишком дорого. А инструменты создания моков и рендереров позволяют делать это вызовом одного метода.

Этот инструмент тоже прошёл кастинг.

## Инструмент хода выполнения теста

Здесь всё просто — красивый консольный график выполнения одного или всех тестов. Удобно, если у вас в проекте их 700 штук, можно распланировать свой день в момент запуска. Тоже берём.

## Инструмент вывода результата

Когда файл теста был запущен, не помешает узнать, чем же закончилось приключение. Для этого используются функции вывода или assert.

Assert выводит сопоставление ожидаемого и полученного результата. Если они не совпадают, возвращает исключение.

Assert — наш кандидат. Идём дальше.

## Инструмент структуры теста

Чтобы код теста был удобно и понятно оформлен, можно использовать тестовый фреймворк. Такие фреймворки ещё называют инструментами структуры теста. Они предоставляют синтаксис и набор методов для оформления кода теста.

Покупаем.

## Инструмент создания и сравнения снапшот-тестов

Снапшот-тестирование — PixelPerfect в мире тестов и разновидность интеграционного теста. В базовой конфигурации снапшот — слепок тестируемого компонента. А если ещё проще — обычная вёрстка. Сравнение двух снапшотов похоже на диффы в Git: если они хоть немного отличаются (в теге или значении HTML-атрибута), сравнение будет неудачным. Это позволяет получить информацию о том, что после изменений в компонентах мог возникнуть непредвиденный баг.

Более прокаченная версия снапшот-тестирования — создание настоящих скриншотов тестируемого компонента и их сравнение.

Снапшоты однозначно стоит попробовать в бою, поэтому запомним этот инструмент.

## Инструменты браузера

Чтобы писать полноценные функциональные тесты, нужно воспроизводить их в среде браузера. Именно там происходит взаимодействие пользователя и приложения. Для этого тоже понадобится инструмент, с которым мы познакомимся позже.

## Инструменты анализа покрытия тестами

Этот инструмент ещё называют code coverage. Он по запросу показывает то, насколько классно (объёмно) разработчик покрыл код приложения тестами. Это звучит многообещающе, но на практике мы не будем использовать этот инструмент, потому что он лишает нас возможности подходить к тестам разумно.

Инструментов тестирования намного больше, чем мы успели рассказать в этом уроке. Мы отобрали необходимые для комфортного написания и запуска тестов. В следующем уроке поговорим про конкретные фреймворки и файловую структуру.

# Инструменты тестирования

В предыдущих уроках вы узнали про виды и подходы к тестированию. Но у тестов должна быть собственная инфраструктура, которая запускает тест, визуализирует его процесс и отображает результат тестирования. Инструментов тестирования очень много. В этом уроке мы устроим кастинг и выберем подходящие для наших задач.

## Инструмент запуска тестов

Сам по себе файл теста не запустится. Для запуска теста и его исполнения существуют инструменты — тест-раннеры. Это фреймворки, которые запускают тесты в определённой среде. О них поговорим в следующем уроке. Но тест-раннер нам однозначно понадобится.

## Инструменты создания моков и рендереров

Что-то новенькое и наверняка полезное. Подробнее про моки и рендереры мы поговорим уже при написании тестов. А сейчас важно то, что это тестовые данные или эмулятор работы API.

Представьте, что вам нужно написать юнит-тест компонента, который завязан на работе с сервером. Писать для теста целый API и пытаться эмулировать работу fetch — слишком дорого. А инструменты создания моков и рендереров позволяют делать это вызовом одного метода.

Этот инструмент тоже прошёл кастинг.

## Инструмент хода выполнения теста

Здесь всё просто — красивый консольный график выполнения одного или всех тестов. Удобно, если у вас в проекте их 700 штук, можно распланировать свой день в момент запуска. Тоже берём.

## Инструмент вывода результата

Когда файл теста был запущен, не помешает узнать, чем же закончилось приключение. Для этого используются функции вывода или assert.

Assert выводит сопоставление ожидаемого и полученного результата. Если они не совпадают, возвращает исключение.

Assert — наш кандидат. Идём дальше.

## Инструмент структуры теста

Чтобы код теста был удобно и понятно оформлен, можно использовать тестовый фреймворк. Такие фреймворки ещё называют инструментами структуры теста. Они предоставляют синтаксис и набор методов для оформления кода теста.

Покупаем.

## Инструмент создания и сравнения снапшот-тестов

Снапшот-тестирование — PixelPerfect в мире тестов и разновидность интеграционного теста. В базовой конфигурации снапшот — слепок тестируемого компонента. А если ещё проще — обычная вёрстка. Сравнение двух снапшотов похоже на диффы в Git: если они хоть немного отличаются (в теге или значении HTML-атрибута), сравнение будет неудачным. Это позволяет получить информацию о том, что после изменений в компонентах мог возникнуть непредвиденный баг.

Более прокаченная версия снапшот-тестирования — создание настоящих скриншотов тестируемого компонента и их сравнение.

Снапшоты однозначно стоит попробовать в бою, поэтому запомним этот инструмент.

## Инструменты браузера

Чтобы писать полноценные функциональные тесты, нужно воспроизводить их в среде браузера. Именно там происходит взаимодействие пользователя и приложения. Для этого тоже понадобится инструмент, с которым мы познакомимся позже.

## Инструменты анализа покрытия тестами

Этот инструмент ещё называют code coverage. Он по запросу показывает то, насколько классно (объёмно) разработчик покрыл код приложения тестами. Это звучит многообещающе, но на практике мы не будем использовать этот инструмент, потому что он лишает нас возможности подходить к тестам разумно.

Инструментов тестирования намного больше, чем мы успели рассказать в этом уроке. Мы отобрали необходимые для комфортного написания и запуска тестов. В следующем уроке поговорим про конкретные фреймворки и файловую структуру.

**Выбор инструментов и файловая структура**

В этом уроке расскажем о файловой структуре и разберёмся, как выбирать инструменты тестирования. Начнём с самого простого: поговорим о том, что из себя представляет тест. Тест — обычный JS-файл, для выполнения которого необходима среда. Прежде чем углубиться в среды для разработки, определимся, где хранятся тесты.

**Файловая структура**

Мы уже приводили пример с файловой структурой [в уроке про Redux](https://praktikum.yandex.ru/learn/react/courses/61ff90eb-9909-4a41-b910-970ae7a90b38/sprints/6023/topics/951327ed-1414-4e31-89d0-70ed1e048bd9/lessons/7cc56e9f-af79-40bd-a96a-31175a5d57c3/). Посмотрим на неё ещё раз:

Скопировать кодJSX

└── src/

├── components/

│ ├── user-profile/

│ │ ├── user-profile.js

│ │ ├── user-profile.test.js

│ │ └── user-profile.module.css

│ ├── cart/

│ │ ├── cart.js

│ │ ├── cart.e2e.test.js

│ │ ├── cart.test.js

│ │ └── cart.module.css

│ └── favorites/

│ ├── favorites.js

│ ├── favorites.test.js

│ └── favorites.module.css

│

└── services/

├── actions/

│ ├── user-profile.js

│ ├── cart.js

│ └── favorites.js

└── reducers/

├── cart.js

├── user-profile.js

├── favorites.js

└── index.js

Здесь файл теста хранится в той же директории, что и тестируемый компонент. А отличается от файла компонента только пространством имён:

* test.js — для юнит тестов;
* e2e.test.js — для сквозных тестов бизнес-логики.

Альтернативный способ хранения тестов — создание директории tests в проекте, которая зеркалирует файловую структуру компонентов или других тестируемых модулей.

В проектной работе вы можете сочетать оба этих подхода: юнит-тесты компонентов хранить рядом с тестируемым компонентом, а сквозные и интеграционные тесты — в отдельной директории.

**Среда выполнения теста**

Ранее мы сказали, что файл теста — обычный JS-файл, и его необходимо выполнить. Есть две среды исполнения тестов: среда браузера и node.js. Мы будем использовать обе: каждая предназначена для определённого инструмента. Но важно обозначить их особенности.

Среда браузера:

* конечное место рендеринга кода для пользователя;
* большой объём тестов потребляет много памяти. Это замедляет процесс разработки.

Среда node.js:

* не предзначена для рендеринга кода;
* «из коробки» умеет создавать файлы;
* обладает высокой производительностью.

Со средой для разработки разобрались. Теперь перейдём к инструментам, которые будут запускать в ней тесты.

**Тест раннеры и другие инструменты**

Современный мир разработки предлагает очень много тест раннеров: Mocha, Jasmine, Jest, Karma и другие. Из длинного списка выберем несколько:

* для исполнения в node.js используем Jest;
* для исполнения в браузере — Cypress.

Jest тесно связан с React и разработан маленькой компанией Facebook. В Jest есть всё необходимое: тест раннер, тестовый фреймворк со всеми инструментами, о которых мы говорили в предыдущем уроке.

Cypress — браузерный тест раннер интеграционных и сквозных тестов.

Оружие для войны с багами мы выбрали. В следующем уроке ещё немного поговорим о том, как не надо пользоваться этим оружием, а после — перейдём к написанию первых тестов.

# Установка всего необходимого

Перед тем как приступить к написанию тестов, следует установить несколько пакетов: тест раннер, библиотеку рендеринга компонентов и удобную обёртку над react-dom.

## Тест раннер. Jest

В качестве тест раннера удобно использовать Jest. Эта библиотека уже встроена в CRA и работает из коробки.

Но если вы используете собственную сборку React-приложения, то добавить в проект Jest можно с помощью такой команды:

Скопировать кодBASH

*# NPM:*

npm install jest --save-dev

*# Yarn:*

yarn add jest --dev

Чтобы запустить Jest в CRA, достаточно воспользоваться командой:

Скопировать кодBASH

*# NPM:*

npm test

*# Yarn:*

yarn test

После этого запустится тестовое окружение, которое будет следить за изменениями файлов тестов и выполнять тесты, когда это необходимо.

## Рендеринг компонентов. Enzyme

Enzyme позволяет частично или полностью рендерить компоненты для применения в тестах.

Чтобы начать работу, необходимо установить саму библиотеку enzyme, а также специальный адаптер, который должен соответствовать вашей версии React:

Скопировать кодBASH

*# NPM:*

npm install enzyme enzyme-adapter-react-16 --save-dev

*# Yarn:*

yarn add enzyme enzyme-adapter-react-16 --dev

После этого библиотеку нужно инициализировать. Сделать это можно в файле setupTests.js:

Скопировать кодJSX

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

import Enzyme from 'enzyme';

import Adapter from 'enzyme-adapter-react-16';

Enzyme.configure({ adapter: new Adapter() });

## Рендереры и моки

Для создания снапшот-тестов вам может пригодиться библиотека react-test-renderer. Её можно установить так:

Скопировать кодBASH

*# NPM:*

npm install react-test-renderer --save-dev

*# Yarn:*

yarn add react-test-renderer --dev

Чтобы «подделывать» запросы к серверу, установим библиотеку fetch-mock:

Скопировать кодBASH

*# NPM:*

npm install fetch-mock --save-dev

*# Yarn:*

yarn add fetch-mock --dev

Ещё нам нужна библиотека для создания фальшивого хранилища. С этой задачей идеально справится redux-mock-store:

Скопировать кодBASH

*# NPM:*

npm install redux-mock-store --save-dev

*# Yarn:*

yarn add redux-mock-store --dev

## Удобная работа с DOM из среды тестирования

Чтобы ускорить разработку и написание тестов, а также для удобства доступа к DOM-иерархии тестируемых компонентов, воспользуемся библиотекой @testing-library/react. По задачам, которые может выполнять эта библиотека, она очень похожа на enzyme.

Установим её:

Скопировать кодBASH

*# NPM:*

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

*# Yarn:*

yarn add @testing-library/react --dev

Всё готово! Можно начинать работать с тестами.

# Тестирование компонентов. Библиотека Jest

В уроке [о выборе инструментов для тестирования](https://praktikum.yandex.ru/trainer/react/lesson/f54f8480-a278-4ed6-943b-1907a6d26f23) мы упоминали библиотеку Jest. Она служит для написания юнит-тестов и использует node.js в качестве среды выполнения. В этом уроке подробнее разберём эту библиотеку и рассмотрим особенности её работы.

## Среда выполнения Jest: jsdom

Когда мы работаем с React и строим пользовательские интерфейсы в браузере, мы взаимодействуем с DOM: рендерим кнопки, текст, инпуты. Затем вешаем на них обработчики событий, а React отображает всё это в браузере. Чтобы протестировать компоненты и их методы, нужно где-то отрендерить эти компоненты и вызвать их обработчики. Здесь в игру вступает jsdom — библиотека, которая реализует работу браузера в среде node.js. Это приблизительная реализация и не все браузерные API работают, как в настоящем браузере, но такой среды достаточно, чтобы протестировать React-компоненты.

Библиотека jsdom служит средой выполнения тестов, которые написаны с использованием Jest. Как и настоящий браузер, jsdom имитирует действия пользователя: можно вызывать события на DOM-узлах, наблюдать за ними и проверять предполагаемые результаты этих действий. Например, имитировать нажатие на кнопку, которая увеличивает значение на единицу, и проверить, что после нажатия значение отличается от предыдущего на эту самую единицу. Но обо всём по порядку — сначала рассмотрим, как тестировать компоненты с помощью снапшотов.

## Тестирование с помощью снапшотов

Вы уже знаете, что снапшоты представляют собой запечатлённый компонент в эталонном виде. В проекте снапшот хранится в отдельном файле и именно с этим файлом сравнивают тестируемый компонент, передавая в него заранее подготовленные пропсы. Напишем снапшот-тест для компонента без внутреннего состояния, который рендерит ссылку:

Скопировать кодJSX

*// link.jsx*

import React from 'react';

export const Link = ({ title, url }) => <a href={url}>{title}</a>;

Наша задача — протестировать компонент Link: проверить его пропсы и убедиться, что они отображаются корректно. Jest предоставляет возможность проверки с помощью снапшотов При первом запуске теста будет создан файл снапшота. Для начала напишем сам тест:

Скопировать кодJSX

*// link.test.js*

import React from 'react';

import renderer from 'react-test-renderer';

import { Link } from './link';

it('Ссылка рендерится без ошибок', () => {

const tree = renderer

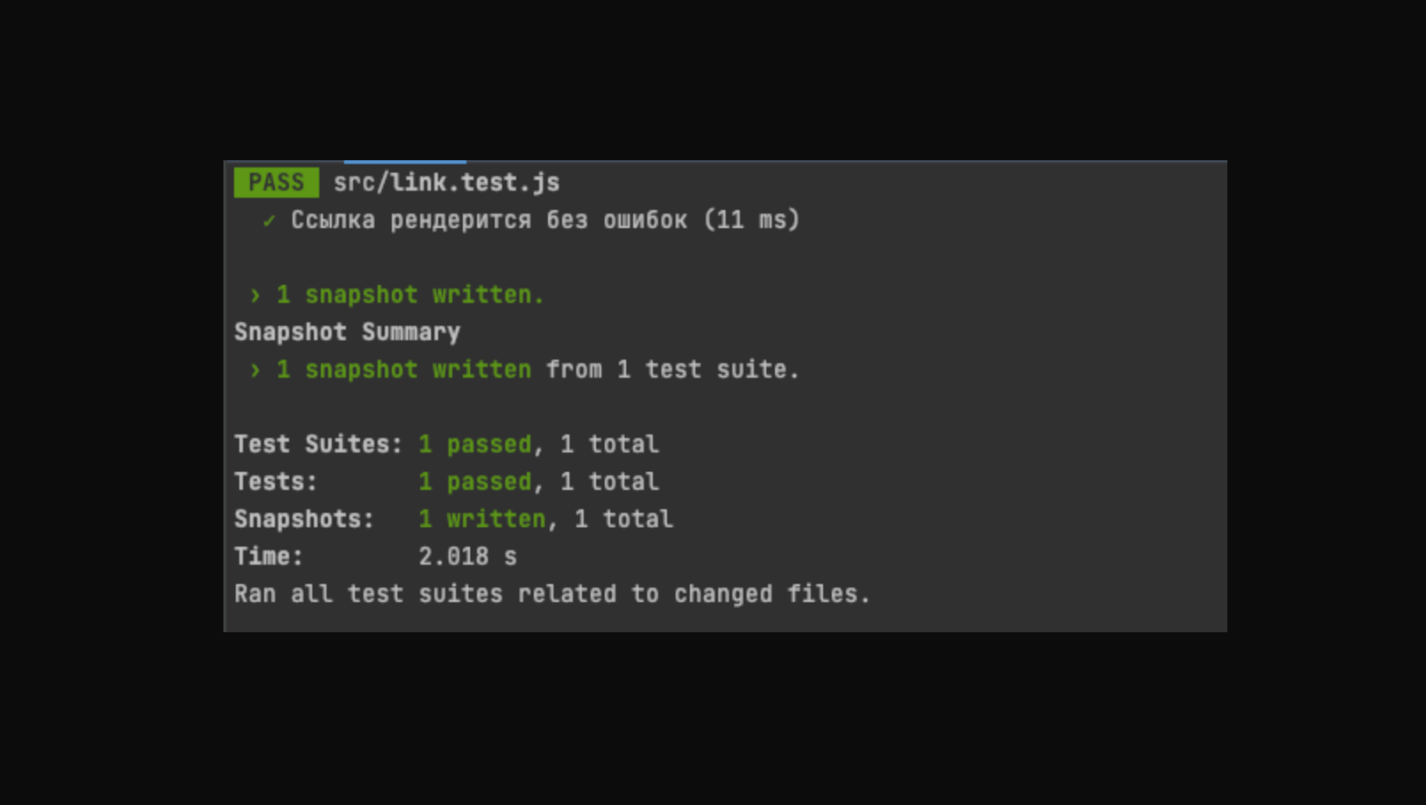
.create(<Link title="Рецепт пельменей" url="https://pelmeni.gov" />)

.toJSON();

expect(tree).toMatchSnapshot();

});

Для запуска теста следует вызвать yarn jest или yarn test. Если всё сделано правильно, в консоли отобразится сообщение об успешности прохождения теста (PASS) и информация о том, что был создан снапшот:



После выполнения тестов Jest автоматически создаст директорию \_\_snapshots\_\_ вместе с файлом снапшота, который содержит результат теста. В нашем примере появится файл link.test.js.snap с результатом рендеринга компонента Link:

Скопировать код

// Jest Snapshot v1, https://goo.gl/fbAQLP

exports[`Ссылка рендерится без ошибок 1`] = `

<a

href="https://pelmeni.gov"

>

Рецепт пельменей

</a>

`;

Если по каким-то причинам необходимо обновить снапшот, можно ещё раз выполнить тестирование со специальным флагом yarn test -- -u или yarn jest -u.

## Тестирование обработчиков событий

Зачастую React-приложения состоят из компонентов со сложной логикой внутри, обработчиков событий и прочих «фишек». Усложним задачу и добавим onClick обработчик в компонент Link:

Скопировать кодJSX

import React from 'react';

export const Link = ({ title, url }) => {

const onClick = () => {

alert('Ура! Пельмени!')

}

return <a href={url} onClick={onClick}>{title}</a>;

}

Если запустить предыдущий тест сейчас, он потерпит неудачу. Это произойдёт потому, что сохранённый снапшот не соответствует обновлённому компоненту, ведь мы добавили свойство onClick. Обновим снапшот вызовом yarn test -- -u. Теперь снапшот выглядит так:

Скопировать код

// Jest Snapshot v1, https://goo.gl/fbAQLP

exports[`Ссылка рендерится без ошибок 1`] = `

<a

href="https://pelmeni.gov"

onClick={[Function]}

>

Рецепт пельменей

</a>

`;

Затем воспользуемся функцией неглубокого рендеринга из библиотеки enzyme и проверим, правильно ли обрабатывается событие onClick на нашей ссылке. Обработчик onClick вызывает alert, поэтому применим Jest для имитации браузерного alert. Для этого обратимся к функции jest.fn(). Так мы сможем имитировать нажатие на ссылку и проверить, что alert вызывается с правильным текстом предупреждения:

Скопировать кодJSX

*// link.test.js*

*// ...*

import { shallow } from 'enzyme';

*// ...*

it('Нажатие на кнопку вызывает корректный alert', () => {

window.alert = jest.fn();

*// Рендерим ссылку в переменную*

const output = shallow(

<Link title="Рецепт пельменей" url="https://pelmeni.gov" />

);

*// Имитируем нажатие на ссылку*

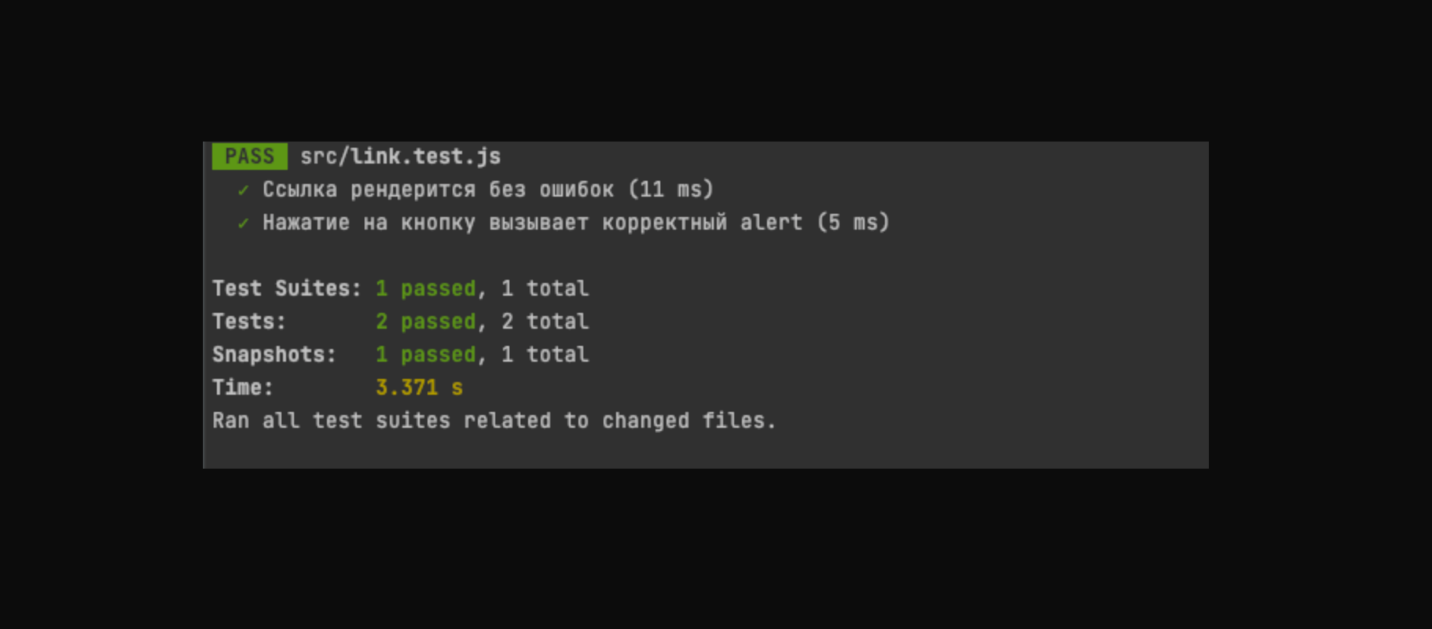
output.simulate('click');

*// Проверяем, что alert сработал с правильным текстом предупреждения*

expect(window.alert).toHaveBeenCalledWith('Ура! Пельмени!');

});

Теперь наш тест проверяет не только соответствие снапшоту, но и работоспособность обработчика onClick на компоненте. Обратите внимание, что второй тест не создаёт снимок, ведь мы не пользовались такой функциональностью в этом тесте. Вывод в консоль теперь выглядит так:



## Тестирование состояния

Тестирование состояния и хуков в функциональных компонентах — довольно сложная задача, поэтому разберём её по шагам. Для начала мы не рекомендуем проверять состояние напрямую. Вместо этого следует проверять эффект от изменения состояния, к примеру, обновление контента, увеличение значения, появление нового блока. Рассмотрим это на примере компонента со счётчиком:

Скопировать кодJSX

import React, { useState } from "react";

export const Counter = () => {

let [count, setCount] = useState(0);

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

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

return (

<div>

<p data-testid="count">{count}</p>

<button data-testid="decrement" type="button" onClick={decrement}>

-

</button>

<button data-testid="increment" type="button" onClick={increment}>

+

</button>

</div>

);

}

В этом компоненте используются data-атрибуты. Благодаря ним мы сможем обратиться к элементам с data-атрибутами и имитировать действия. В этом нам поможет библиотека @testing-library/react. С её помощью вызовем onClick на кнопках и после этого сверим значение в элементе <p data-testid="count">...</p>. Тест будет выглядеть так:

Скопировать кодJSX

import { render, getByTestId, fireEvent } from '@testing-library/react';

import { Counter } from './counter';

it("Плюс и минус в счетчике работают без ошибок", () => {

*// Рендерим компонент*

const { container } = render(<Counter />);

*// Находим элемент со значением счетчика*

const countValue = getByTestId(container, "count");

*// Находим кнопку, увеличивающую значение*

const increment = getByTestId(container, "increment");

*// Находим кнопку, уменьшающую значение*

const decrement = getByTestId(container, "decrement");

*// Проверяем, что начальное состояние счетчика равно 0*

expect(countValue.textContent).toBe("0");

*// Увеличиваем значение счетчика, симулируя нажатие на соответствующую кнопку*

fireEvent.click(increment);

*// Проверяем, что состояние счетчика теперь рано 1*

expect(countValue.textContent).toBe("1");

*// Уменьшаем значение счетчика, симулируя нажатие на соответствующую кнопку*

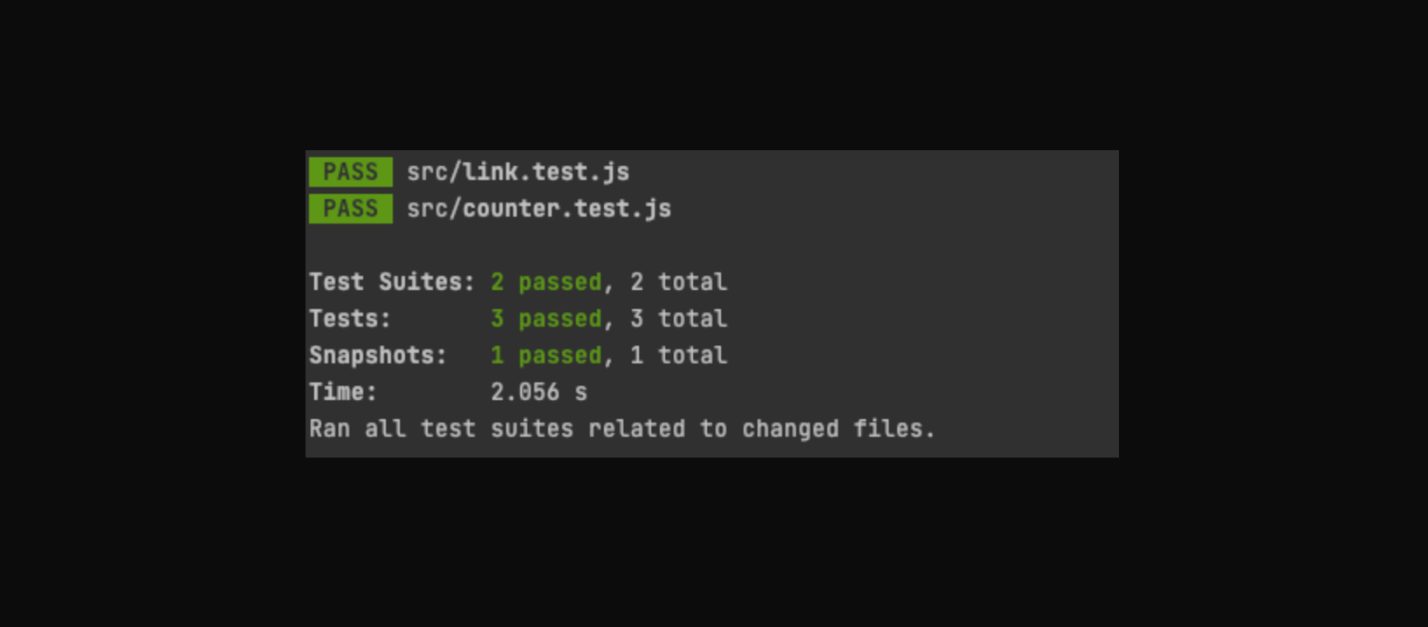
fireEvent.click(decrement);

*// Проверяем, что состояние счетчика теперь рано 0*

expect(countValue.textContent).toBe("0");

});

В примере мы воспользовались функцией fireEvent из пакета @testing-library/react. Эта функция позволяет симулировать действия. Аналогичным образом работает метод simulate в библиотеке Enzyme. Если всё сделано правильно, в консоли отобразится такой вывод:



# Подведём итоги

Библиотека Jest в связке с дополнительными инструментами позволяет писать очень детальные и точные юнит-тесты. Покрытие приложения тестами помогает держать кодовую базу в чистоте, а команду — в тонусе. Тестами важно покрывать сложные места, которые содержат много бизнес-логики. О её тестировании поговорим в следующем уроке. Важно понимать, что инструментарий для тестирования не ограничен библиотеками, которые мы перечислили. У каждого решения есть сильные и слабые стороны, но их изучение выходит за рамки этого курса.

# Тестирование Redux

В предыдущем уроке вы познакомились с библиотекой Jest и вспомогательными инструментами для тестирования компонентов. Но приложения состоят не только из компонентов, а ещё и из логики, которая управляет этими компонентами, порядком их отображения и работой с данными. В нашем приложении вся бизнес-логика хранится в Redux, так что в этом уроке мы подробно разберём его тестирование.

## Подготовка к тестированию

В этом уроке мы протестируем редьюсеры и экшены. Но сперва подготовимся — установим необходимые библиотеки и создадим фальшовое хранилище. Тестирование редьюсеров и компонентов схоже: в обоих случаях применяются юнит-тесты. Поэтому здесь нам пригодится уже знакомая библиотека Jest.

Для тестирования асинхронных экшенов нам потребуется библиотека fetch-mock, которая «фальсифицирует» запросы к серверу (моки):

Скопировать кодBASH

*# NPM:*

npm install fetch-mock --save-dev

*# Yarn:*

yarn add fetch-mock --dev

Ещё нам нужна библиотека для создания фальшивого хранилища. С этой задачей идеально справится redux-mock-store:

Скопировать кодBASH

*# NPM:*

npm install redux-mock-store --save-dev

*# Yarn:*

yarn add redux-mock-store --dev

Теперь всё готово! Можем приступать к тестированию Redux.

## Тестирование экшенов

Генераторы экшенов возвращают простые объекты. В ходе тестирования нам нужно проверить, соответствует ли эталону объект, возвращаемый генератором экшенов.

Разберёмся на примере. Создадим такой генератор экшена:

Скопировать кодJSX

export function setPrice(price) {

return {

type: 'SET\_PRICE',

price

}

}

А протестировать его можно так:

Скопировать кодJSX

import \* as actions from './actions'

import \* as types from './constants'

describe('Action creators', () => {

it('should create an action with correct price', () => {

const price = 145600;

*// Эталонный экшен*

const expectedAction = {

type: types.SET\_PRICE,

price

}

*// Проверяем экшены на равенство*

expect(actions.setPrice(price)).toEqual(expectedAction)

})

})

В этом примере мы сравниваем результат вызова генератора экшена с эталонным экшеном, который определяем выше по коду.

## Тестирование асинхронных экшенов

С генераторами экшенов разобрались, следующие на очереди — асинхронные генераторы экшенов. Для асинхронных действий в Redux обычно создают три типа экшенов: запрос отправлен, ответ получен, произошла ошибка:

Скопировать кодJSX

function fetchTodosRequest() {

return {

type: FETCH\_TODOS\_REQUEST

}

}

function fetchTodosSuccess(body) {

return {

type: FETCH\_TODOS\_SUCCESS,

body

}

}

function fetchTodosFailure(ex) {

return {

type: FETCH\_TODOS\_FAILURE,

ex

}

}

*// Асинхронный генератор экшена*

export function fetchTodos() {

return dispatch => {

dispatch(fetchTodosRequest())

return fetch('http://example.com/todos')

.then(res => res.json())

.then(body => dispatch(fetchTodosSuccess(body)))

.catch(ex => dispatch(fetchTodosFailure(ex)))

}

}

Для тестирования асинхронных генераторов экшенов нам потребуется подменить хранилище и запросы к серверу на моки. Вот как можно это сделать:

Скопировать кодJSX

import configureMockStore from 'redux-mock-store'

import thunk from 'redux-thunk'

import \* as actions from './actions'

import \* as types from './constants'

import fetchMock from 'fetch-mock'

const middlewares = [thunk]

const mockStore = configureMockStore(middlewares)

describe('async actions', () => {

afterEach(() => {

fetchMock.restore()

})

it('creates FETCH\_TODOS\_SUCCESS when fetching todos has been done', () => {

fetchMock.getOnce('/todos', {

body: { todos: ['do something'] },

headers: { 'content-type': 'application/json' }

})

const expectedActions = [

{ type: types.FETCH\_TODOS\_REQUEST },

{ type: types.FETCH\_TODOS\_SUCCESS, body: { todos: ['do something'] } }

]

const store = mockStore({ todos: [] })

return store.dispatch(actions.fetchTodos()).then(() => {

*// Возвращаем асинхронный экшен*

expect(store.getActions()).toEqual(expectedActions)

})

})

})

## Тестирование редьюсеров

В отличие от компонента, редьюсер — всегда чистая функция. Это значительно упрощает тестирование. Напишем простой редьюсер:

Скопировать кодJSX

import { ADD\_TODO } from '../constants/ActionTypes'

const initialState = [

{

text: 'Use Redux',

completed: false,

id: 0

}

]

export default function todos(state = initialState, action) {

switch (action.type) {

case ADD\_TODO:

return [

{

id: state.reduce((maxId, todo) => Math.max(todo.id, maxId), -1) + 1,

completed: false,

text: action.text

},

...state

]

default:

return state

}

}

А теперь — тест для этого редьюсера:

Скопировать кодJSX

import reducer from '../../structuring-reducers/todos'

import \* as types from '../../constants/ActionTypes'

describe('todos reducer', () => {

it('should return the initial state', () => {

expect(reducer(undefined, {})).toEqual([

{

text: 'Use Redux',

completed: false,

id: 0

}

])

})

it('should handle ADD\_TODO', () => {

expect(

reducer([], {

type: types.ADD\_TODO,

text: 'Run the tests'

})

).toEqual([

{

text: 'Run the tests',

completed: false,

id: 0

}

])

expect(

reducer(

[

{

text: 'Use Redux',

completed: false,

id: 0

}

],

{

type: types.ADD\_TODO,

text: 'Run the tests'

}

)

).toEqual([

{

text: 'Run the tests',

completed: false,

id: 1

},

{

text: 'Use Redux',

completed: false,

id: 0

}

])

})

})

Тестирование редьюсеров схоже с тестированием простых генераторов экшенов — мы сравниваем простые объекты. Разница лишь в том, что в случае с редьюсерами мы передаём экшен и проверяем, как отреагировало хранилище.

Тестирование Redux кому-то кажется увлекательным, а кому-то — скучным. И это нормально. Но стоит помнить, что иногда разработчики в Redux реализуют сложную логику и отправляют тяжеловесные экшены. И, если вы отважились писать тесты, такие редьюсеры и экшены не стоит оставлять без внимания.

# Cypress. Введение

Разработчики часто сталкиваются с необходимостью выполнять однообразные действия. Одна из подобных рутин — проверка работы программы. Например, вы написали новый компонент для приложения: его работоспособность следует проверить, а ещё удостовериться, что не пострадала остальная функциональность. Чем больше в приложении компонентов, тем сложнее выполнять все эти действия вручную. А в итоге приложение может так разрастись, что проверять всю функциональность становится настолько долго, сложно и, как следствие, дорого, что этот этап иногда просто пропускают. А это может привести к ошибкам в продакшене и негативной реакции конечных пользователей.

Но существуют инструменты, которые позволяют автоматизировать проверку работоспособности приложения и полностью избежать таких проблем. После добавления новой функциональности в проект достаточно описать его ожидаемое поведение в виде тест-кейса, после чего тест-раннер проверит, всё ли работает, как вы задумали. А если что-то пойдёт не так — вы получите подробное описание ошибки и поэтапные скриншоты произошедшего.

Одно из таких решений — Cypress — тест-раннер, который умеет выполнять тесты прямо в браузере. С помощью Cypress вы можете создавать, запускать и отлаживать функциональные, интеграционные и модульные тесты. Мы уже рассматривали модульные тесты на примере библиотеки Jest. В этой теме сосредоточимся на первых двух типах.

В следующем уроке разберём, как установить Cypress в проект и запустить тестовое окружение.

**Cypress. Установка и запуск**

Чтобы начать работать с библиотекой Cypress, её нужно установить в проект. Это можно сделать с помощью одной команды:

Скопировать кодBASH

*# NPM:*

npm install cypress --save-dev

*# Yarn:*

yarn add cypress --dev

Затем Cypress будет установлен в директорию node\_modules.

Cypress — сложная программа с большим количеством логики, и для её установки может потребоваться около 500 Мбайт дискового пространства.

**Запуск**

Для запуска тестового окружения нужно вызвать исполняемый файл, который находится в node\_modules/.bin/cypress:

Скопировать кодBASH

*# bin:*

./node\_modules/.bin/cypress open

Также запустить окружение можно с помощью npx или yarn:

Скопировать кодBASH

*# NPX:*

npx cypress open

*# Yarn:*

yarn cypress open

Для удобства вы можете создать специальную команду для запуска тестового окружения в файле package.json, к примеру:

Скопировать кодJSON

"scripts": {

"cypress:open": "./node\_modules/.bin/cypress open",

},

Запустить эту команду можно так:

Скопировать кодBASH

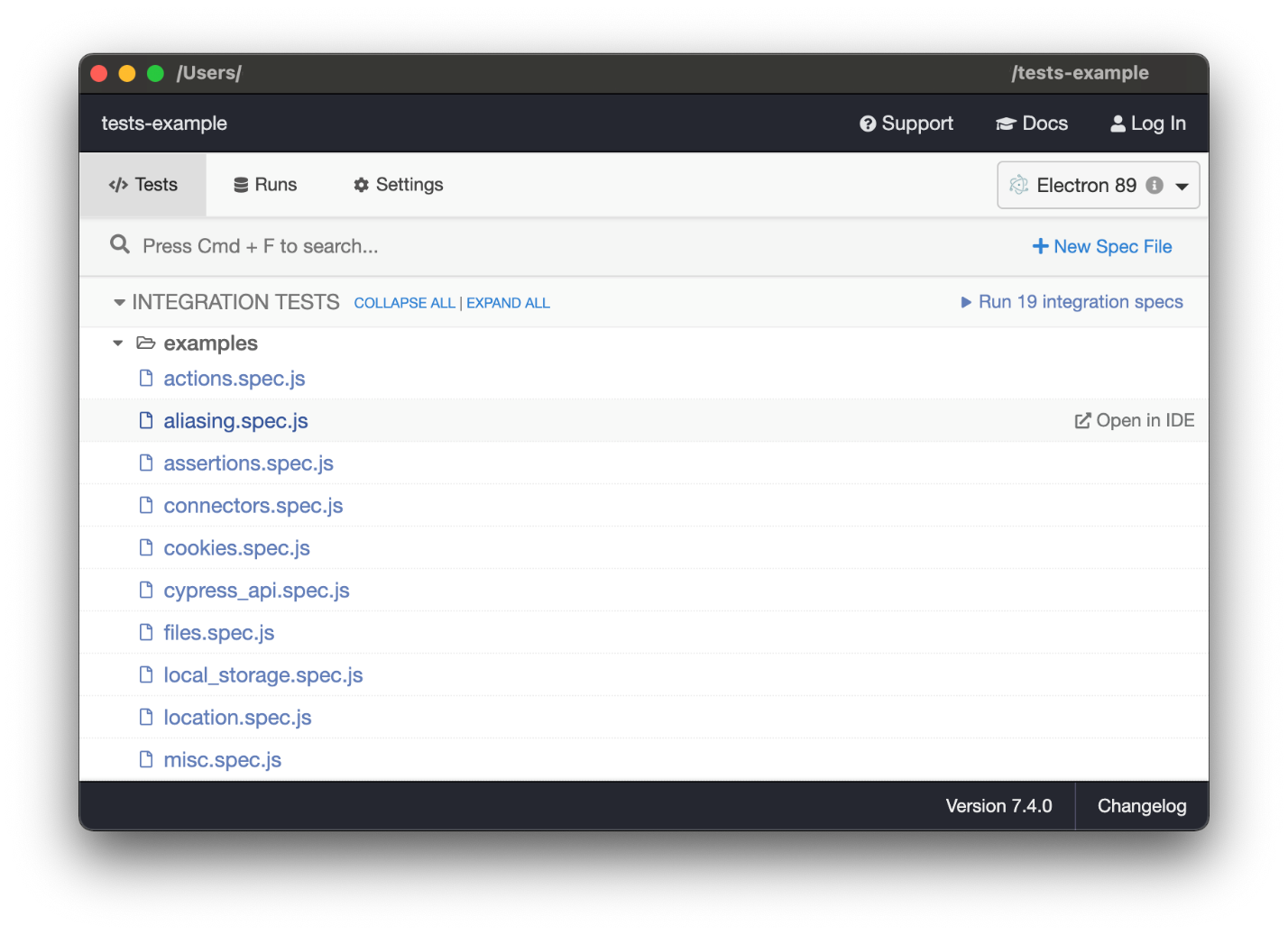
*# NPM:*

npm run cypress:open

*# Yarn:*

yarn cypress:open

При выполнении этой команды запустится тестовое окружение:



Самое время порадоваться, что у вас всё получилось, и закрыть это окно — оно пригодится на следующих этапах подготовки проекта.

**Настройка линтера**

В случае, если у вас в проекте установлен eslint, в тестовых файлах он будет подсвечивать вызовы объектов и методов как неизвестные. Чтобы разрешить эту проблему, достаточно установить плагин для eslint и настроить конфигурацию для Cypress:

* установите плагин eslint-plugin-cypress:

Скопировать кодBASH

*# NPM:*

npm install eslint-plugin-cypress --save-dev

*# Yarn*

yarn add eslint-plugin-cypress --dev

* в директории cypress, которая была создана при первом запуске тестового окружения, создайте файл .eslintrc.json;
* укажите плагин, который будет использоваться в этой директории, заодно можно добавить рекомендованный набор правил:

Скопировать кодJSON

{

"plugins": [

"cypress"

],

"extends": [

"plugin:cypress/recommended"

]

}

**Подготовка инфраструктуры**

При первом запуске тестового окружения библиотека создаёт в приложении директорию cypress с такой архитектурой:

Скопировать кодJSON

cypress

├── fixtures

├── integration

│   └── examples

├── plugins

└── support

Нас интересует только директория integration: именно в ней Cypress по умолчанию ищет файлы тестов.

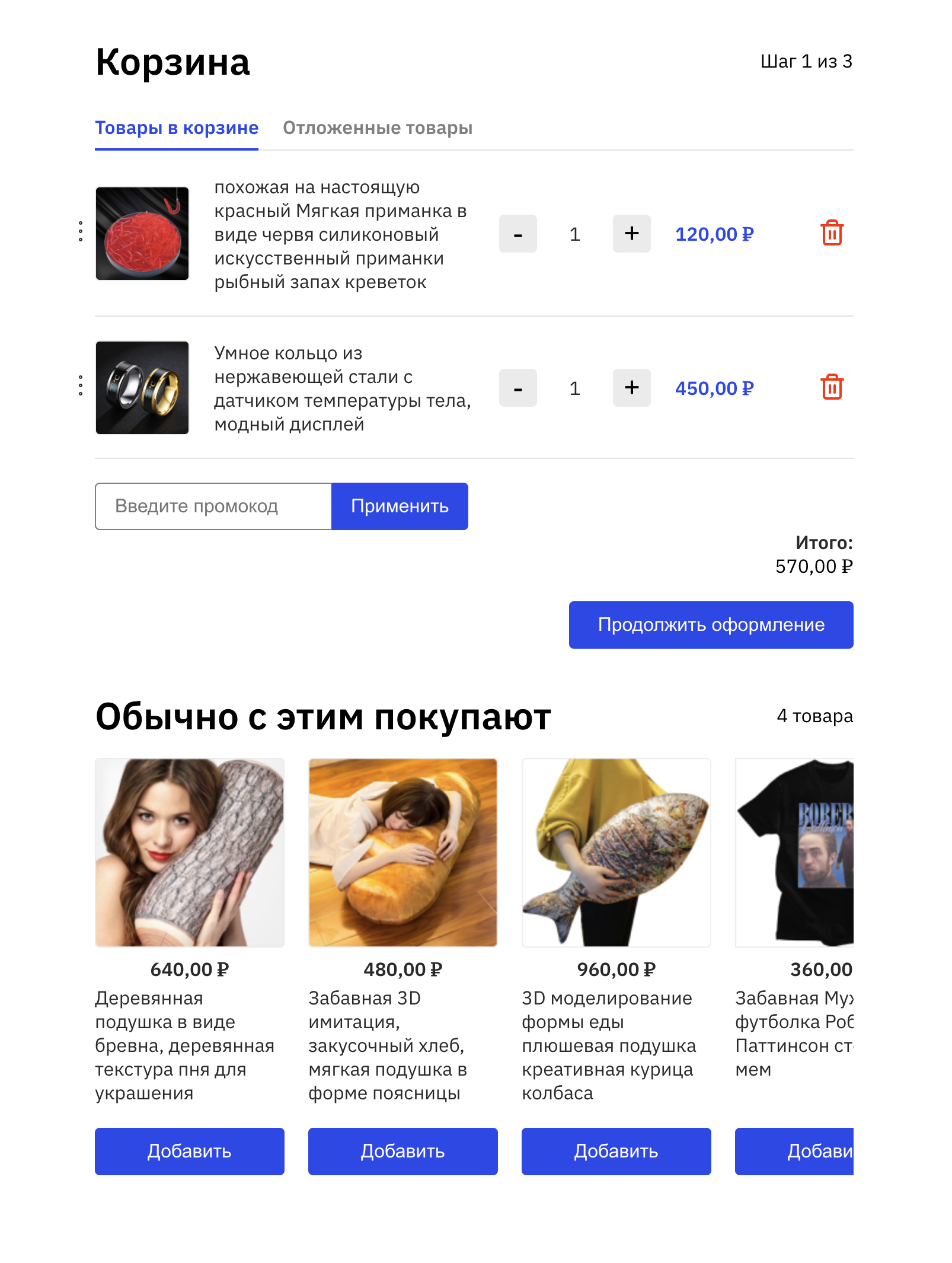
В директории integration есть поддиректория examples с примерами тестовых файлов. Сейчас все эти примеры для нас не важны и могут мешать при запуске тестов, поэтому их лучше удалить.

Теперь всё готово к написанию первых тестов. Научимся создавать их уже в следующем уроке.

**Cypress. Первый тест**

В предыдущем уроке вы установили Cypress и научились его запускать. Теперь попробуем написать первый тест.

Для примера покроем тестами проект с корзиной товаров из курса про глобальное состояние.



*Так выглядит корзина*

В корзине мы можем изменять количество товаров, применять промокоды, выбирать пункт доставки и оформлять заказы. Протестируем эту функциональность.

Чтобы приступить к написанию тестов, нужно одновременно запустить приложение и тестовое окружение:

Скопировать кодBASH

*# Эти команды выполняются в разных окнах терминала*

yarn start

yarn cypress:open

В качестве первого теста проверим, запущено ли приложение:

Скопировать кодJSX

*// {project}/cypress/integration/cart/is-available.spec.js*

describe('service is available', function() {

it('should be available on localhost:3000', function() {

cy.visit('http://localhost:3000');

});

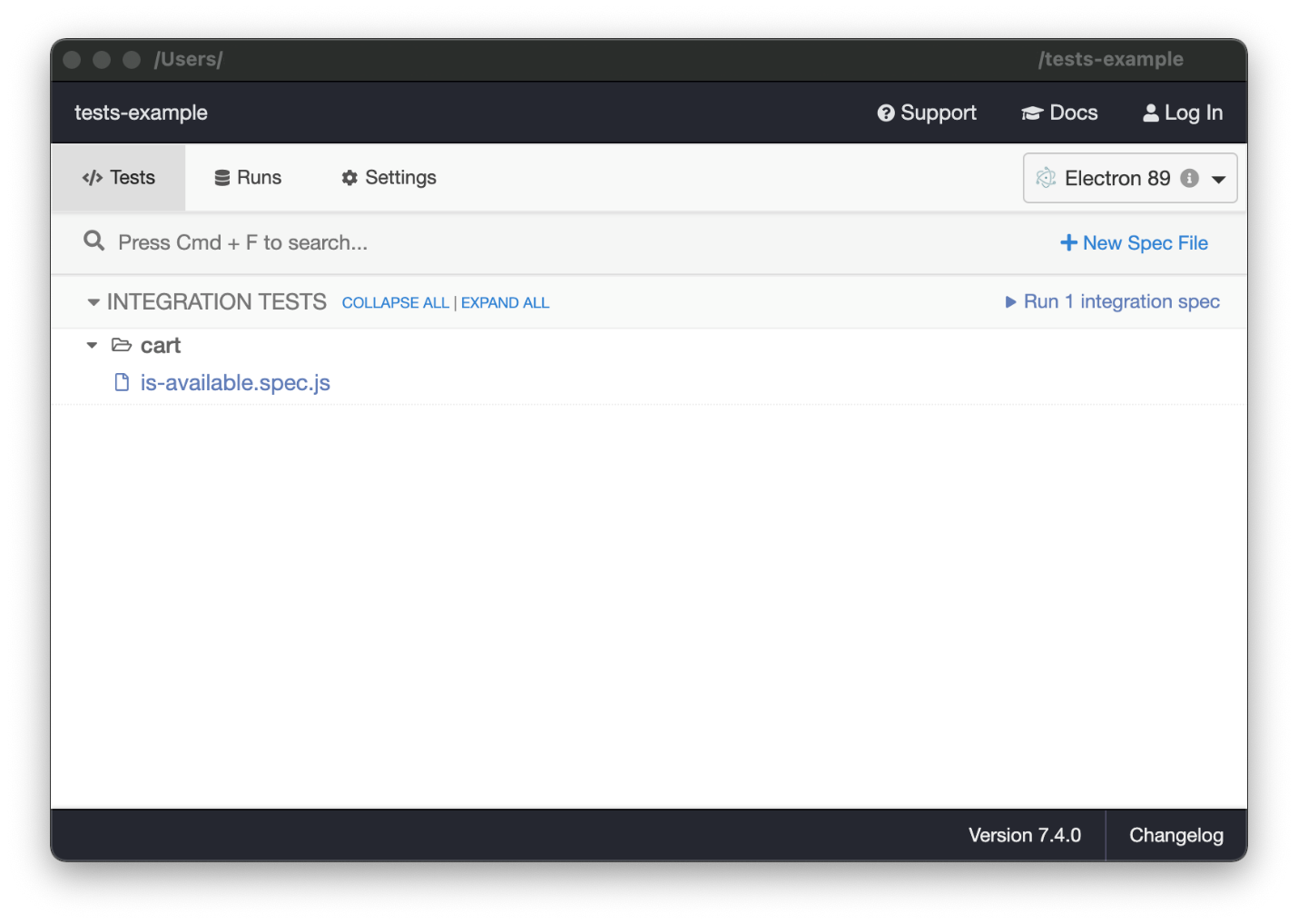
});

Чтобы тест-раннеры легко находили файлы тестов, к этим файлам принято добавлять постфикс spec или test. Cypress по умолчанию предлагает пользоваться spec, и мы не будем отходить от этой рекомендации.

Код тестов Cypress достаточно легко читать. Чтобы в этом убедиться, подробнее разберём содержимое теста is-available.spec.js:

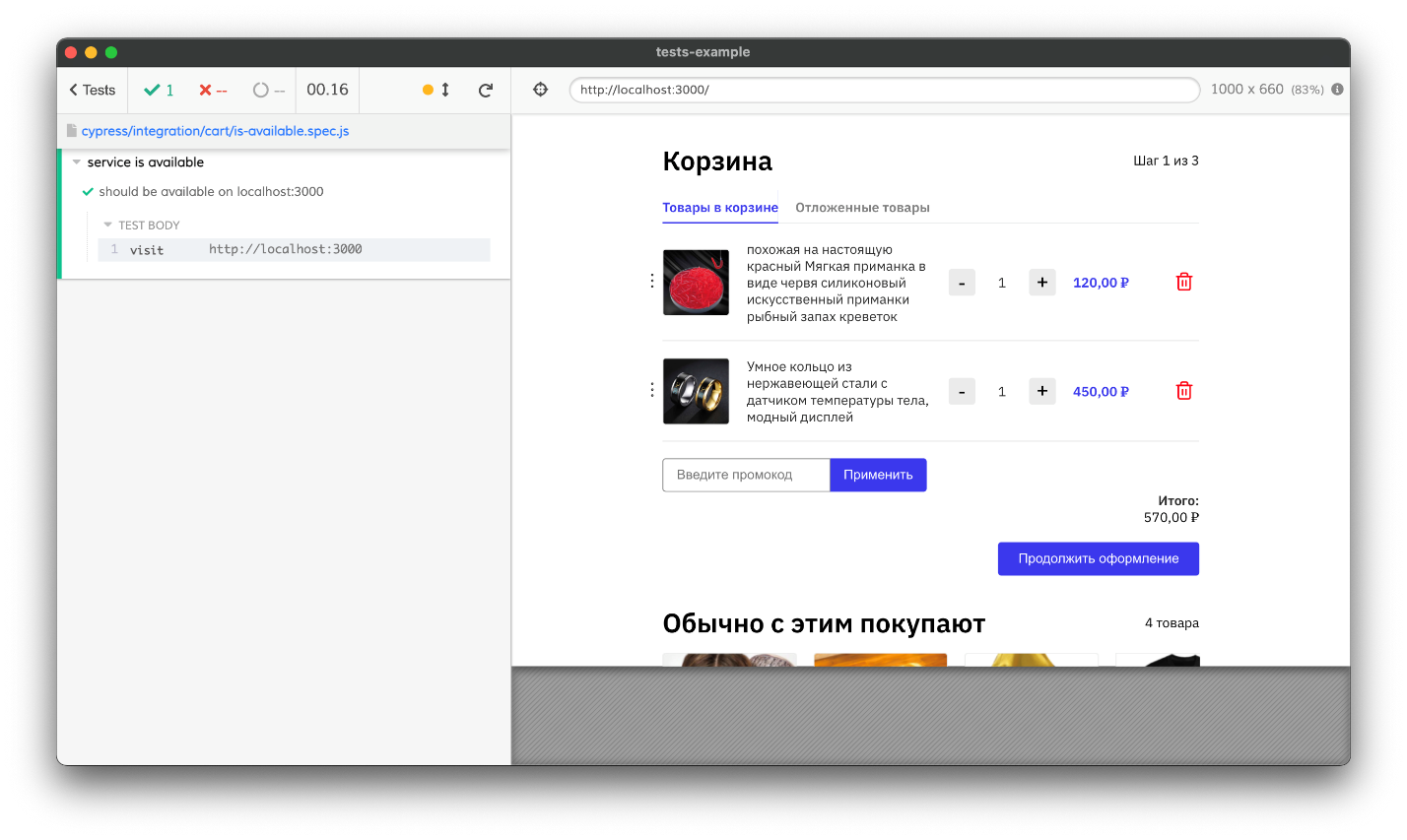
* describe — обёртка теста, с помощью которой можно кратко описать суть тест-кейсов, которые находятся внутри неё;
* it — то же, что и describe, только для конкретного тест-кейса. Обычно тест-кейсы описываются как «это (it) должно работать так-то»;
* cy.visit — метод, с помощью которого можно перейти на любую страницу приложения (или в интернете);

После создания файла теста в директории integration, Cypress находит его автоматически:



*Cypress автоматически следит за набором доступных тестов*

Для запуска теста достаточно кликнуть по его названию, после чего Cypress запустит среду выполнения (в примере это Electron, но можно выбрать и браузер), и выполнит все инструкции, указанные в тест-кейсах:



*Для выполнения тестов локально запускается настоящий браузер и эмулируются действия пользователя, описанные в тест-кейсах*

В этом уроке мы посмотрели на Cypress в бою: запустили окружение, написали пробный тест и выполнили его. А в следующем — напишем несколько настоящих тестов, с помощью которых можно проверить работу приложения.

**Cypress. Покрытие приложения тестами**

В этом уроке рассмотрим несколько примеров тестов, которые приближены к реальной работе.

**Тестирование переходов по страницам**

Роутинг — важная часть многих приложений. Например, функциональность корзины состоит из нескольких шагов: списка товаров, адреса и типа доставки и подтверждения заказа. В первую очередь важно удостовериться, что все шаги доступны пользователю и работают, как должны. Проверить эту функциональность можно так:

Скопировать кодJSX

describe('app works correctly with routes', function() {

before(function() {

cy.visit('http://localhost:3000');

});

it('should open cart page by default', function() {

cy.contains('Корзина');

});

it('should open delivery page after continue button click', function() {

cy.get('button').contains('Продолжить оформление').click();

cy.contains('Доставка');

});

it('should open agreement page after continue button click', function() {

cy.contains('Обычная доставка').click();

cy.get('button').contains('Продолжить оформление').click();

cy.contains('Подтверждение заказа');

});

});

Разберём этот пример подробнее:

* Код в блоке before выполняется в первую очередь. В этом блоке можно разместить какие-нибудь подготовительные инструкции, например переход на нужную страницу.
* Метод contains проверяет наличие текстового содержимого на странице. С помощью этого метода можно удостовериться, что первой страницей загрузилась корзина.
* Метод get работает примерно как поиск по селектору в jQuery. Здесь мы используем его, чтобы найти кнопки на странице. Так как кнопок может быть много, отфильтруем вывод get с помощью метода contains.
* С помощью метода click можно вызвать событие клика на любом элементе страницы.

Благодаря этому тесту мы можем быть уверены, что каждый из этапов корзины доступен пользователю.

**Тестирование поведения элементов на странице**

Теперь проверим функциональность самой корзины. У пользователя есть возможность увеличивать и уменьшать количество товаров. Проверить работоспособность этой логики можно так:

Скопировать кодJSX

describe('products management works correctly', function() {

before(function() {

cy.visit('http://localhost:3000');

});

it('should add and subtract products count', function() {

cy.get('[class^=product\_product\_\_]').first().as('product');

cy.get('@product').find('[class^=amount-button\_button]').first().as('minusButton');

cy.get('@product').find('[class^=amount-button\_button]').last().as('plusButton');

cy.get('@product').find('[class^=product\_amount\_\_]').as('productCount');

cy.get('@productCount').should('contain', '1');

cy.get('@plusButton').click();

cy.get('@productCount').should('contain', '2');

cy.get('@plusButton').click();

cy.get('@productCount').should('contain', '3');

cy.get('@minusButton').click();

cy.get('@productCount').should('contain', '2');

cy.get('@minusButton').click();

cy.get('@productCount').should('contain', '1');

});

});

Из-за того, что методы Cypress выполняются асинхронно, нельзя просто положить результат работы метода в переменную. Но вместо переменных можно воспользоваться механизмом псевдонимов, или алиасов (англ. alias). Метод as создаёт псевдоним, которым можно воспользоваться в других методах, например в get.

Вот, что происходит в этом примере:

* Сначала назначаются алиасы для всех компонентов, которые участвуют в тестировании.
* Затем методом click эмулируются нажатия на кнопки.
* В конце методом should проверяется, изменилось ли количество товаров.

**Тестирование стилей**

Помимо изменений состояния на странице могут присутствовать различные визуальные эффекты. Проверить их наличие тоже можно с помощью Cypress.

На странице оформления доставки есть форма выбора типа доставки. Выбранный тип должен визуально выделяться. Проверить это можно с помощью такого теста:

Скопировать кодJSX

describe('delivery page display correctly', function() {

before(function() {

cy.visit('http://localhost:3000');

cy.get('button').contains('Продолжить оформление').click();

});

it('should highlight selected type of delivery', function() {

cy.get('[class^=delivery-method-option\_option]').first().as('expressDelivery');

cy.get('[class^=delivery-method-option\_option]').last().as('defaultDelivery');

cy.get('@expressDelivery').should('have.css', 'background-color', 'rgb(250, 250, 250)');

cy.get('@defaultDelivery').should('not.have.css', 'background-color', 'rgb(250, 250, 250)');

cy.get('@defaultDelivery').contains('Обычная доставка').click();

cy.get('@expressDelivery').should('not.have.css', 'background-color', 'rgb(250, 250, 250)');

cy.get('@defaultDelivery').should('have.css', 'background-color', 'rgb(250, 250, 250)');

});

});

Метод should помогает определить наличие или отсутствие любого атрибута и значения элемента.

Вы узнали об основах работы с библиотекой Cypress. Она позволяет полностью эмулировать действия пользователя: переходить по определённому адресу, находить нужные элементы на странице, наводить курсор на элементы, кликать по ним и вызывать другие события. Все эти возможности помогают протестировать поведение страницы так, как будто с ней взаимодействует пользователь.

**Заключение**

Написать код — хорошо, но ещё лучше иметь код, который покрыт тестами. А чем выше покрытие, тем более надёжная система получается на выходе. Но во всём должен быть здравый смысл, а автоматизация — не панацея, ей не стоит увлекаться.

В этой теме вы узнали про автоматическое тестирование: виды автотестов и подходы к их написанию. А ещё познакомились с основными инструментами для написания тестов. Этих знаний достаточно, чтобы начать покрывать приложение тестами.

Но перед тем, как вы приступите к написанию собственных тестов, прочитайте о том, как тестировать не надо:

1. Тесты ради тестов — не путь воина. Не гонитесь за тем, чтобы написать тест ради code coverage. А большое количество бесполезных тестов может только усложнить рефакторинг приложения.
2. Тестирование статичной вёрстки не информативно. Такой тест лучше либо не писать совсем, либо преобразовать его в функциональный.
3. Не пишите тесты-близнецы. Если два или больше компонентов работают схожим образом, к примеру, взаимодействуют с похожими данными на входе, обладают схожей функциональностью, но являются разными элементами бизнес-логики, то лучше не писать по юнит-тесту для каждого из компонентов.

Представьте, что у вас есть компонент Product и Postponed — товары в корзине и отложенные товары. Оба компонента работают схожим образом. Если ограничиться только юнит-тестом к каждому компоненту (при этом, они будут отличаться незначительно), то такие тесты становятся бесполезными. Разработчик, который не знаком со структурой компонентов Product и Postponed, отредактирует их родительский компонент — List, и изменит только тесты для компонента Product. При этом проект продолжит проходить тесты без ошибок — тесты компонента Postponed ничего не знают об изменениях в компоненте List. Поэтому такие юнит-тесты нужно либо дополнять функциональными, либо не использовать вовсе, а применять только функциональные тесты.

Это всё. В этой теме не было заданий, но вы можете ознакомиться с полным кодом приложения и тестов к ним [по ссылке](https://github.com/Yandex-Practicum/react-developer-cart-app-tests-example). Кажется, вы готовы приступить к проектной работе. В ней вам предстоит закрепить знания на практике — написать несколько тестов.