**Что такое тестирование**

Вы уже умеете программировать сервер: подключать базу данных, настраивать роутинг, реализовывать авторизацию и обладаете навыками для разработки бэкенда сайта. Но для создания качественного веб-сервиса этого недостаточно. Кроме кода для управления запросами и ответами нужно написать тесты. Разберёмся, что это и зачем они нужны.

Разработчик Стас написал систему для компании-клиента. Всё работает, все счастливы. За это Стаса повысили, а на его место пришёл разработчик Миша.

Через год компания-клиент попросила дополнить систему новыми фичами, а старые — слегка изменить. Миша написал грамотный код, но после посыпались жалобы на то, что старые функции стали «отваливаться». Миша позвал Стаса, который уже не помнил, как именно писал код год назад. Но за несколько недель они смогли совместными усилиями заставить систему работать как надо.

Таких проблем можно избежать. Для этого при написании новой функциональности необходимо проверять, не сломалась ли старая. Стас бы потратил больше времени при разработке, зато Миша сразу увидел бы, где возникают проблемы, и исправил бы их.

Вручную такие проверки делать сложно, поэтому нужны тесты. Они позволяют автоматически прогонять программу по сценариям использования и исключить внезапные поломки отдельных функций.

Для написания тестов есть разные фреймворки. Самые распространённые — Jest и Mocha. Мы будем работать с Jest: на нём удобно писать тесты и для фронтенда, и для бэкенда.

# Тестовые фреймворки и методы

Тестовые фреймворки — набор библиотек для проверки кода. С их помощью можно узнать, что запросы по-прежнему уходят на сервер после переделывания кнопок формы, а слайдеры листаются, даже когда мы поменяли сервер с изображениями.

Разработчик Василий сидит на работе. Ему звонит сын Коля:

— Пап, у нас электричество отключили. Пытаюсь включить компьютер — не работает. Свет тоже не включается.

Василий помнит, что в квартире могла вылететь одна из пробок. Чтобы это выяснить, Василий поручает Коле выполнить тест:

— Сходи на кухню и проверь свет. Если работает, значит, пробки выбило.

Коле предстоит протестировать функцию включения света. У неё есть ожидаемый результат: загорятся лампочки на кухне. Смысл теста в том, чтобы проверить ожидаемый результат и сравнить его с реальным.

Для тестирования кода применяются те же принципы. Тесты позволяют проверить, совпадает ли результат функции с нашими ожиданиями. Правило представляет собой утверждение «если — то»: если передать этой функции вот такой аргумент, то мы получим вот это.

### Как установить фреймворк

Чтобы получить доступ к тестовым функциям, подключим фреймворк к проекту. Для этого нужно установить в dev-зависимости пакет jest:

Скопировать кодBASH

npm install --save-dev jest

Пропишем имя пакета в разделе "scripts" файла package.json:

Скопировать кодJSON

// package.json

...

"scripts": {

"test": "jest"

},

...

Готово! Теперь можно запускать тесты командой npm run test. Но сначала нужно описать тесты.

### Где писать тесты

Создадим два файла: для кода проекта и для тестов. Чтобы Jest автоматически запускал тесты, файл с ними следует назвать так же, как файл с основным кодом и приписать к имени .test. Наши файлы будут называться function.js и function.test.js.

В файле с основным кодом опишем функцию, которая создаёт приветствие, — её мы и будем тестировать:

Скопировать кодJAVASCRIPT

*// function.js*

const sayHello = (firstName, secondName) => {

return `Здравствуйте, ${firstName} ${secondName}!`;

};

module.exports = sayHello;

### Как писать тесты

Теперь создадим тест для этой функции. В Jest за это отвечает функция it. На вход она принимает два параметра: описание теста и колбэк с кодом. Всё, что происходит внутри колбэка, — и есть тест.

Скопировать кодJAVASCRIPT

it ('Что делает тестируемая функция', () => {

*// чего мы ожидаем и в каком случае*

});

Затем опишем сам тест: если функции sayHello передать аргументы "Стас" и "Басов", она должна вернуть "Здравствуйте, Стас Басов!".

Здесь поможет функция expect: она принимает на вход значение и возвращает объект с множеством методов для тестирования. Среди них метод toBe — он сравнивает свой аргумент с тем, который был передан функции expect:

Скопировать кодJAVASCRIPT

expect(1).toBe(1); *// true*

expect(1).toBe(2); *// false*

expect(1 + 2).toBe(3); *// true*

Так мы можем передать функции expect результат работы функции sayHello и сравнить его с ожидаемым результатом:

Скопировать кодJAVASCRIPT

it ('Создаёт приветствие', () => {

expect(sayHello("Стас", "Басов")).toBe("Здравствуйте, Стас Басов!");

});

### Как посмотреть результаты тестирования

Готово! Запустим тесты командой npm run test:

Скопировать кодBASH

*# Результат из консоли*

PASS ./function.test.js

✓ Создаёт приветствие (3ms)

Test Suites: 1 passed, 1 total

Tests: 1 passed, 1 total

Snapshots: 0 total

Time: 1.747s

Ran all test suites.

Слово PASS говорит, что тесты прошли успешно. Получается, Jest запущен и работает.

Разберёмся, как читать непройденные тесты. Изменим функцию приветствия:

Скопировать кодJAVASCRIPT

const sayHello = (firstName, secondName) => {

return `Здравствуйте, дорогой ${firstName} ${secondName}!`;

};

module.exports = sayHello;

И допишем для неё тест:

Скопировать кодJSX

it ('Создаёт приветствие', () => {

expect(sayHello("Стася", "Басова")).toBe("Здравствуйте, дорогая Стася Басова!");

});

Запускаем npm run test и смотрим в консоль:

Скопировать кодBASH

> jest

FAIL ./function.test.js *# Сообщает, в каком файле провалился тест*

✕ Создаёт приветствие (12ms)

● Создаёт приветствие

expect(received).toBe(expected) // Object.is equality

Expected: "Здравствуйте, дорогая Стася Басова!" *# Ожидание*

Received: "Здравствуйте, дорогой Стася Басова!" *# Реальность*

3 |

4 | it('Создаёт приветствие', () => {

> 5 | expect(sayHello("Стася", "Басова")).toBe("Здравствуйте, дорогая Стася Басова!"); *# Здесь указывается конкретное место ошибки*

| ^

6 | });

7 |

at Object.toBe (function.test.js:5:39)

Test Suites: 1 failed, 1 total

Tests: 1 failed, 1 total *# Сообщает о проваленных тестах из общего числа*

Snapshots: 0 total

Time: 1.963s

Ran all test suites.

По сообщению о результатах тестирования легко определить, какие ошибки возникли и где. А значит, проще отловить и исправить их.

# Какие есть методы тестирования

Любой метод тестирования проверяет, совпадают ли наши ожидания с результатами работы юнита.

Бóльшую часть тестов можно выполнить одним из методов:

### toBe сопоставляет примитивные значения или ссылки на объекты:

Скопировать кодJAVASCRIPT

*// Тесты пройдут*

expect('Ожидание').toBe('Ожидание');

const a = {};

const b = a;

expect(a).toBe(b);

*// Тесты не пройдут*

expect('Ожидание').toBe('Реальность');

const a = {};

const b = {};

expect(a).toBe(b);

### toEqual сопоставляет объекты или массивы. При сравнении объектов метод проверяет, что в полученном объекте есть те же ключи с теми же значениями, что и в ожидаемом. В проверяемом объекте при этом могут быть и другие свойства — их метод toEqual просто игнорирует.

При сравнении массивов метод проверяет, что в полученном результате все элементы точно совпадают. При этом просто пропущенные элементы и undefined считаются равными:

Скопировать кодJAVASCRIPT

*// Тесты пройдут*

expect({ a: undefined, b: 10, c: 'text' }).toEqual({ b: 10, c: 'text' });

expect([1, 2, 3]).toEqual([1, 2, 3]);

expect([[undefined, 1]]).toEqual([[,1]]) *// в правом массиве первый элемент пропущен*

*// Тесты не пройдут*

expect({ a: undefined, b: 10, c: 'text' }).toEqual({ a: 12, b: 10, c: 'text' });

expect([1, 2, 3, undefined]).toEqual([1, 2, 3]); *// массивы не совпадают по длине, в конце правого отсутствует ","*

### toStrictEqual тоже сопоставляет объекты или массивы, но более строго: ожидаемый объект или массив должен полностью соответствовать действительному. То есть обладать теми же свойствами или элементами. Тут undefined и пропущенный элемент не равны:

Скопировать кодJAVASCRIPT

*// Тесты пройдут*

expect({ b: 10, c: 'text' }).toStrictEqual({ b: 10, c: 'text' });

expect([3,4,undefined]).toStrictEqual([3,4,undefined]);

*// Тесты не пройдут*

expect({ a: undefined, b: 10, c: 'text' }).toStrictEqual({ b: 10, c: 'text' });

expect([[undefined, 1]]).toStrictEqual([[,1]])

### toBeTruthy и toBeFalsy сравнивают результат с true и false соответственно:

Скопировать кодJAVASCRIPT

*// Тесты пройдут*

expect(1).toBeTruthy();

expect(true).toBeTruthy();

expect(undefined).toBeFalsy();

expect(1/'string').toBeFalsy();

*// Тесты не пройдут*

expect(null).toBeTruthy();

expect(0).toBeTruthy();

expect(true).toBeFalsy();

### toBeUndefined и toBeDefined сравнивают результат со значением undefined.

Метод toBeUndefined проходит, если результат не определён, то есть равен undefined.

А метод toBeDefined наоборот — если результат не равен undefined. Этот тест пройдёт даже со значением null — оно считается определённым.

Скопировать кодJAVASCRIPT

*// Тесты пройдут*

expect(1).toBeDefined();

expect(null).toBeDefined();

expect('string').toBeDefined();

*// Тесты не пройдут*

let x;

expect(x).toBeDefined();

expect(undefined).toBeDefined();

### toBeNull сравнивает результат с null:

Скопировать кодJAVASCRIPT

*// Тесты пройдут*

const x = null;

expect(null).toBeNull();

expect(x).toBeNull();

*// Тесты не пройдут*

expect(0).toBeNull();

expect(undefined).toBeNull();

expect('string').toBeNull();

### toMatch проверяет строку на соответствие регулярному выражению:

Скопировать кодJAVASCRIPT

*// Тесты пройдут*

expect('1').toMatch(/^\d+$/);

expect('1337').toMatch(/^\d+$/);

expect('1337').toMatch(/^\d+$/);

*// Тесты не пройдут*

expect('21as1').toMatch(/^\d+$/);

expect('string').toMatch(/^\d+$/);

### toContain проверяет, есть ли в полученном массиве нужное нам значение. Со строками тоже работает:

Скопировать кодJAVASCRIPT

*// Тесты пройдут*

expect('Oh, hi Mark!').toContain('Mark');

expect(['Маша', 'Ира', 'Стас']).toContain('Стас');

*// Тесты не пройдут*

expect(['Маша', 'Ира', 'Стас']).toContain('Миша');

expect('Oh, hi Mark!').toContain('Lisa');

### toBeGreaterThan, toBeGreaterThanOrEqual, toBeLessThan и toBeLessThanOrEqual — операторы математического сравнения. Сравнивают результат с переданным числом:

Скопировать кодJAVASCRIPT

*// Тесты пройдут,*

*// потому что*

expect(2).toBeGreaterThan(1); *// 2 больше 1*

expect(2).toBeLessThan(3); *// 2 меньше 3*

expect(2).toBeGreaterThanOrEqual(2); *// 2 больше или равно 2*

expect(2).toBeLessThanOrEqual(2); *// 2 меньше или равно 2*

*// Тесты не пройдут*

*// потому что:*

expect(2).toBeGreaterThan(2); *// 2 не больше 2*

expect(2).toBeLessThan(2); *// 2 не меньше 2*

expect(2).toBeLessThanOrEqual(1); *// 2 больше и не равно 1*

expect(2).toBeGreaterThanOrEqual(3); *// 2 меньше и не равно 3*

Этих методов хватает для большинства задач. Полный список можно посмотреть в документации: <https://jestjs.io/docs/en/getting-started>.

С тестами в коде проект проще масштабировать, ведь при обновлении легко отлавливать ошибки и исправлять их. Кроме того, тесты документируют собой код: по описаниям тестов понятно, что делает приложение, и в его работе проще разобраться.

Что тесты НЕ делают?



Проверяют, что для работы функции достаточно данных.



Документируют код так, что другим разработчикам легче понять его.



Описывают работу сервера для пользователей сайта.

Правильно: пользователям своя документация, разработчикам — своя.

Для чего нужны тестовые фреймворки?



Предоставляют набор функций, который позволяет описывать поведение тестируемой части кода.

Правильно! При чтении аккуратно написанных тестов становится ясно, что делает приложение в целом и каждая его часть.



Позволяют наладить работу между отделами, занимающимися тестированием.



Автоматически тестируют типовые ситуации, которые возникают в коде.

# Юниты и методы тестирования

В предыдущем уроке мы разобрали, что представляют собой тесты и для чего они нужны. Тестами мы создаём дополнительный уровень защиты: проверяем, что результаты работы функций совпадают с нашими ожиданиями.

Ещё мы написали тест для функции, которая создаёт приветствие:

Скопировать кодJAVASCRIPT

it ('Создаёт приветствие', () => {

expect(sayHello("Стас", "Басов")).toBe("Здравствуйте, Стас Басов!");

});

Здесь всё просто, поскольку в проекте только одна функция. Но когда функций становится много, нужно определить, к каким нужно писать тесты.

### Какие функции тестировать

Желательно — все. Программу следует разбить на юниты — мельчайшие функции для выполнения конкретной задачи. Разберёмся, как это делать.

В сервисе есть функция, которая проверяет, что пароль содержит цифру и спецсимвол. Каждая проверка делается отдельной функцией:

Скопировать кодJAVASCRIPT

*// проверяем наличие цифры*

function checkNumber(pass) {

if (typeof pass === 'string') {

let regex = /\d/;

return regex.test(pass); *// метод test вернёт true, если шаблон регулярного выражения найден в строке*

}

}

*// проверяем наличие спецсимвола*

function checkSymbol(pass) {

if (typeof pass === 'string') {

let regex = /[!@#$%^&\*(),.?":{}|<>\_]/;

return regex.test(pass);

}

}

*// запускаем обе проверки*

function checkPass(pass) {

return checkNumber(pass) && checkSymbol(pass);

}

Каждая из этих функций — юнит. Поэтому тест следует написать для каждой функции:

Скопировать кодJAVASCRIPT

it('Проверяет наличие цифры', () => {

expect(checkNumber('some\_not\_so\_strong\_pass')).toBe(false);

expect(checkNumber('stronger\_pass\_123')).toBe(true);

});

it('Проверяет наличие спецсимвола', () => {

expect(checkSymbol('somePass')).toBe(false);

expect(checkSymbol('another\_pass')).toBe(true);

});

it('Проверяет пароль', () => {

expect(checkPass('somePas$')).toBe(false);

expect(checkPass('another\_pass\_123')).toBe(true);

});

Некоторые разработчики совершают грубую ошибку: пишут тесты только для самых «верхнеуровневых» функций. В нашем случае такой нерадивый разработчик протестировал бы только функцию checkPass. Не делайте так: если программа сломается, придётся потратить уйму времени, чтобы отыскать ошибку. А если тесты написаны для каждой функции, ошибку найти легко: нужно просто прочитать сообщение с результатом теста.

# Юниты и методы тестирования: практика 1

Протестируйте функцию-генератор случайных чисел от 10 до 80 включительно. Функция должна возвращать:

* число,
* больше или равно 10,
* но меньше или равно 80.

### Задание

<https://code.s3.yandex.net/web-code/randomizer.zip>

### Подсказка

Для проверки типа данных воспользуйтесь typeof.

Числа сравните методами toBeGreaterThanOrEqual и toBeLessThanOrEqual.

### По ссылке наше решение:

<https://code.s3.yandex.net/web-code/randomizer-solution.zip>

# Юниты и методы тестирования: практика 2

Протестируйте функцию-генератор url-строки. Функция должна возвращать строку, которая проходит проверку регулярным выражением из переменной urlRegEx.

### Задание

<https://code.s3.yandex.net/web-code/urlGenerator.zip>

### Подсказка

Чтобы проверить соответствие строки регулярному выражению, воспользуйтесь методом toMatch.

### По ссылке наше решение:

<https://code.s3.yandex.net/web-code/urlGenerator-solution.zip>

# Тестовые наборы

В предыдущем уроке мы рассказывали о юнит-тестах. Но уметь писать тесты недостаточно, нужно уметь их организовать.

К примеру, нужно протестировать обработчик запроса на регистрацию. Такой обработчик проверяет данные на валидность, обсчитывает хеш пароля и записывает данные в базу. Каждое из этих действий выполняет отдельная функция, а обработчик запроса по очереди их вызывает. И все функции необходимо протестировать.

Тесты юнитов, которые работают ради одной большой цели, — например, регистрации — следует объединять в тестовые наборы. Так проще изменять, дописывать и читать тесты. В этом уроке — о том, как создавать тестовые наборы.

### Как создать набор Функция describe

Функция describe создаёт набор. Её первый аргумент — описание набора, второй — колбэк с тестами:

Скопировать кодJAVASCRIPT

describe('Проверка обработчика запроса', () => {

it('должен проверять данные на валидность', () => {*/\* здесь код теста \*/*});

it('должен считать хеш пароля', () => {*/\* здесь код теста \*/*});

it('должен записывать данные в базу', () => {*/\* здесь код теста \*/*});

});

Функция describe дополняет информацию в сообщении о результате тестирования:

Скопировать кодBASH

PASS ./api.test.js

Проверка обработчика запроса *# вот эта строчка добавилась*

✓ должен проверять данные на валидность (3ms)

✓ должен считать хеш пароля (5ms)

✓ должен записывать данные в базу (7ms)

Test Suites: 1 passed, 1 total

Tests: 3 passed, 3 total

Snapshots: 0 total

Time: 1.747s

Ran all test suites.

Также у describe есть третий параметр: максимальное время исполнения тестового набора. Это ограничение защищает от зависаний тестов из-за бесконечных циклов и рекурсий.

По умолчанию это время составляет 5 секунд. Но его можно:

* увеличить, если тест может зависеть от «внешних» процессов: например, ответа сервера при обработке крупных файлов или обращении к платёжным системам;
* уменьшить: такое бывает нужно, когда мы меняем код не для новой функциональности, а для оптимизации существующей.

Таким образом, метод describe помогает аккуратно оформить тесты и ограничить время их выполнения.

## Тестовые наборы: практика

Исходный код задания: [https://code.s3.yandex.net/web-code/Тестовые-наборы.zip](https://code.s3.yandex.net/web-code/%D0%A2%D0%B5%D1%81%D1%82%D0%BE%D0%B2%D1%8B%D0%B5-%D0%BD%D0%B0%D0%B1%D0%BE%D1%80%D1%8B.zip)

В этом задании протестируем функцию валидации данных validateUserInput. При регистрации пользователь отправляет данные: email и пароль. Прежде чем записывать их в базу, нужно проверить, что данные соответствуют шаблонам: телефон состоит из цифр, в email есть «собака», имя пользователя содержит только латинские буквы.

Нужно протестировать, что функция validateUserInput правильно проверяет данные.

Работает validateUserInput так: принимает на вход объект с данными и передаёт их функциям isValidEmail и isValidPassword для проверки почты и пароля соответственно. Обе функции возвращают true, если данные прошли проверку, и false — если нет.

validateUserInput возвращает объект result с тремя свойствами:

* isValidated — true, если проверка пароля и почты прошла, false — если нет;
* message — строка 'Пользователь успешно cоздан!', если проверки пройдены, и null — если нет;
* error — null, если проверки пройдены, а если нет — одно из сообщений: 'Пароль неправильный', 'Email неправильный' или 'Данные не верны' (это если и пароль, и email не проходят проверку).

Мы подготовили тестовые данные в объектах:

* данные из dataValid должны проходить все проверки;
* данные из dataInvalidPassword не должны проходить проверку пароля;
* данные из dataInvalidEmail не должны проходить проверку email;
* данные из dataInvalidCredentials не должны проходить ни одну из проверок.

Проверьте, что функции возвращают в каждом случае верные значения: isValidEmail и isValidPassword — true или false, а validateUserInput — ответ с верными данными.

### Подсказка

Опишите тестовый набор и добавьте в него тесты:

* #isValidEmail должна проверять, что email валиден;
* #isValidPassword должна проверять, что пароль валиден;
* #validateUserInput должна возвращать message, если данные верны, и не возвращать ошибку;
* #validateUserInput должна возвращать ошибку об email, если он неверный;
* #validateUserInput должна возвращать ошибку о пароле, если он неверный;
* #validateUserInput должна возвращать общую ошибку, если все данные неверные.

### По ссылке наше решение:

[https://code.s3.yandex.net/web-code/Тестовые-наборы-решение.zip](https://code.s3.yandex.net/web-code/%D0%A2%D0%B5%D1%81%D1%82%D0%BE%D0%B2%D1%8B%D0%B5-%D0%BD%D0%B0%D0%B1%D0%BE%D1%80%D1%8B-%D1%80%D0%B5%D1%88%D0%B5%D0%BD%D0%B8%D0%B5.zip)

# Тестирование запросов

В курсе об асинхронном коде мы пользовались [kanye.rest](http://kanye.rest/" \t "_blank) — внешним API. Может наступить момент, когда разработчики kanye.rest поменяют правила обращения к своему API, и проект перестанет работать.

Чтобы такого не произошло, нужно написать тесты для обработчиков запросов. В этом уроке расскажем о библиотеке supertest. Её инструментами такие тесты удобно создавать.

## Подготовим проект для тестирования

Для тестирования сервера структуру проекта нужно немного доработать. Это связано с принципом работы тестов.

Supertest работает так: начинает тестирование → подключается к серверу → выполняет тесты → отключается от сервера → завершает тесты. Таким образом, supertest берёт на себя заботу по подключению и отключению от сервера.

Когда мы запускаем сервер из index.js, supertest не может отключить сервер, а следовательно, не может завершить тесты. Поэтому он их даже не запустит, а покажет ошибку:

Скопировать кодBASH

// Вывод в консоли

Jest did not exit one second after the test run has completed.

This usually means that there are asynchronous operations that weren't stopped in your tests. Consider running Jest with `--detectOpenHandles` to troubleshoot this issue.

Чтобы разделить подключение к серверу для тестирования и подключение к серверу для работы сервиса, файл index.js разбивают на два:

* server.js, где описывают подключение к серверу:

Скопировать кодJAVASCRIPT

*// server.js*

const app = require('./app.js');

const { PORT = 3000 } = process.env;

app.listen(PORT, () => {

console.log(`App listening on port ${PORT}`);

});

* app.js, где описывают всё остальное:

Скопировать кодJAVASCRIPT

*// app.js*

const express = require('express');

const { PORT = 3000 } = process.env;

const app = express();

app.get('/', (req, res) => {

res.status(200).send('Hello World!')

});

module.exports = app;

Затем в тестовые файлы импортируют app.js и тестируют. И ничего не конфликтует.

## Устанавливаем supertest

Вводим команду npm install --save-dev supertest.

Скопировать кодJSON

// package.json

"scripts": {

"start": "node server.js" // было "node app.js"

}

Создаём файл endpoint.test.js для тестов. К нему нужно подключить библиотеку:

Скопировать кодJAVASCRIPT

*// endpoint.test.js*

const supertest = require('supertest');

const app = require('./app.js');

В переменную supertest записана функция, которой нужно передать на вход наше приложение:

Скопировать кодJAVASCRIPT

const request = supertest(app);

Готово! Теперь мы можем обращаться к методам библиотеки через объект request. Все методы этого объекта возвращают промисы, которые нужно обработать асинхронно.

### Проверяем отправку запроса

Для каждого типа запроса есть одноимённый метод: get, post, delete, put и patch. Каждый из них принимает на вход URL, запрос на который мы хотим проверить:

Скопировать кодJAVASCRIPT

describe('Эндпоинты откликаются на запросы', () => {

it('Возвращает данные и 200-й ответ по запросу к "/"', () => {

return request.get('/').then((response) => {

expect(response.status).toBe(200);

expect(response.text).toBe('Hello World!');

});

});

});

При тестировании промисов возвращайте их: пишите return перед request.get. Так Jest автоматически дождётся, пока промис разрешится или сообщит об ошибке при отклонении.

## Настраиваем запросы

Кроме URL, у запросов есть атрибуты, которые могут содержать файлы. Разберём методы для атрибутов.

### set устанавливает атрибуты. На вход он принимает два параметра: имя атрибута и его значение:

Скопировать кодJAVASCRIPT

.set('Cookie', ['token=u1a90aw7812689adukqyw61;'])

### send задаёт тело запроса.

Имя метода сбивает с толку, но помните: методы get, post, delete, put и patch отправляют запросы. А send позволяет добавить данные в тело запроса:

Скопировать кодJAVASCRIPT

.send({ name: 'Пётр' })

### query позволяет настроить GET-запрос:

у него нет тела — только URL. Чтобы добавить в URL данные, методу query передают данные внутри объекта:

Скопировать кодJAVASCRIPT

.query({ per\_page: '50', offset: '20' })

### attach прикрепляет к запросу файл.

Первый параметр — имя файла, второй — относительный путь к нему:

Скопировать кодJAVASCRIPT

.attach('avatar', 'test/fixtures/avatar.jpg')

Обратите внимание на имя папки fixtures. Так принято называть директорию с данными, которые нужны для тестирования: картинками, аудио и видео.

 Привет! Эти методы позволяют настроить твой запрос, чтобы иметь возможность протестировать разные его варианты. Например, с разными заголовками, с query параметрами и без них. Они также добавляются в тестах в цепочку к request

describe('Эндпоинты откликаются на запросы', () => {  
 it('Возвращает данные и 200-й ответ по запросу к "/"', () => {  
 return request.get('/')  
 .query({search: "bye"})  
 .then((response) => {  
 expect(response.status).toBe(200);  
 expect(response.text).toBe('');  
 });  
 });  
});

Ещё больше методов в документации: <https://jestjs.io/docs/en/getting-started>.

## Тестирование запросов: практика

[https://code.s3.yandex.net/web-code/Тестирование-запросов.zip](https://code.s3.yandex.net/web-code/%D0%A2%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5-%D0%B7%D0%B0%D0%BF%D1%80%D0%BE%D1%81%D0%BE%D0%B2.zip)

Мы подготовили код серверного приложения. Вам нужно:

1. скачать его;
2. установить зависимости командой npm install;
3. написать тесты:
   * GET-запрос к адресу “/” должен возвращать строку “Hello World!” и корректный статус;
   * POST-запрос к адресу "/users" должен возвращать JSON с данными пользователя и корректный статус, говорящий, что пользователь был создан;
4. запустить тест командой npm run test.

## Подсказки

* Если пользователь был создан, следует вернуть статус ответа 201;
* чтобы убедиться, что пришёл JSON, нужно проверить заголовок POST-ответа content-type: он должен содержать application/json.
* //Можно по разному делать. Я обычно сравниваю ключи полученного объекта. Например:
* expect(res.body).to.contain.keys('historyId', 'accountUUID', 'boxUUID');

## По ссылке наше решение:

[https://code.s3.yandex.net/web-code/Тестирование-запросов-решение.zip](https://code.s3.yandex.net/web-code/%D0%A2%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5-%D0%B7%D0%B0%D0%BF%D1%80%D0%BE%D1%81%D0%BE%D0%B2-%D1%80%D0%B5%D1%88%D0%B5%D0%BD%D0%B8%D0%B5.zip)

# Тестирование базы данных

[В одной из предыдущих тем мы говорили, как подключать к серверу базу данных](https://praktikum.yandex.ru/trainer/web/lesson/ee86ceda-e835-4003-830e-94f953bc86f8). Теперь расскажем, как тестировать взаимодействие с базой данных.

Есть ограничение: нельзя использовать в тестах реальные данные. «Сломанная» функция может повредить данные пользователей, поэтому при тестировании в базу добавляют случайные бессмысленные данные, а потом их удаляют.

Получается алгоритм: добавляем в базу тестовые данные → тестируем → удаляем тестовые данные.

Такой же алгоритм применим к каждому тесту по отдельности. В Jest есть четыре метода:

* beforeAll и afterAll запускаются до и после выполнения всех тестов в файле;
* beforeEach и afterEach запускаются до и после каждого теста.

Перед началом всех тестов мы обычно подключаемся к базе данных, а в самом конце — отключаемся. Для этого подходят методы beforeAll и afterAll.

В начале каждого теста мы добавляем тестовые данные в базу, а в конце — удаляем их. Тут помогают beforeEach и afterEach.

Напишем эти методы кодом:

Скопировать кодJAVASCRIPT

beforeAll(() => {

*// Подключаемся к базе данных*

});

afterAll(() => {

*// Отменяем подключение к БД*

});

describe('Database tests', () => {

beforeEach(() => {

*// Перед каждым тестом добавляем нужные данные в БД*

});

afterEach(() => {

*// После каждого теста убираем данные из БД*

});

it('should do the test work...', () => {*/\* код для проверки тестов \*/*});

});

Также методы beforeAll и afterAll можно вызывать внутри describe. Тогда колбэк beforeAll и afterAll будет вызван в начале и в конце тестового набора:

Скопировать кодJAVASCRIPT

beforeAll(() => {

*// Подключаемся к базе данных*

});

afterAll(() => {

*// Отменяем подключение к БД*

});

describe('Database tests', () => {

beforeAll(() => { *// сработает перед всеми тестами внутри этого describe*

*// Перед каждым тестом добавляем нужные данные в БД*

});

afterAll(() => { *// сработает после всех тестов в этом describe*

*// После каждого теста убираем данные из БД*

});

it('should do the db test work...', () => {*/\* код для проверки тестов для бд \*/*});

it('should do the other test work...', () => {*/\* код для проверки других тестов \*/*});

});

describe('Endpoints tests', () => {

*/\**

*Здесь beforeAll и afterAll из соседнего блока describe*

*срабатывать не будут, но сработают глобальные beforeAll и afterAll*

*\*/*

it('should test endpoint', () => {*/\* код для проверки тестов \*/*});

})

Методы beforeAll, afterAll, beforeEach и afterEach позволяют «без шума» протестировать базу данных: мы добавляем информацию для тестирования, а потом стираем её. Главное — осознанно расположить методы. Это приходит с практикой.

## Тестирование базы данных: практика

Исходный код задания: [https://code.s3.yandex.net/web-code/Тестирование-базы-данных.zip](https://code.s3.yandex.net/web-code/%D0%A2%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5-%D0%B1%D0%B0%D0%B7%D1%8B-%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85.zip)

При тестировании базы данных нужно:

* подключиться к БД в начале тестов и отключиться в конце;
* перед каждым тестом добавлять данные в базу, а после — удалять их.

В Mongoose для этих задач есть методы:

* mongoose.connect — для подключения: на вход ему передают URL базы и объект опций;
* mongoose.disconnect — для отключения;
* mongoose.create — для создания записи;
* mongoose.deleteOne — для удаления записи в БД — он принимает на вход объект с данными, по которым можно найти запись для удаления.

Скопировать кодJAVASCRIPT

*// Найдёт запись, у которой поле username равно 'Вася':*

User.deleteOne({ username: 'Вася' })

Также можно использовать отдельную базу данных и полностью сбрасывать её в конце тестирования методом dropDatabase:

Скопировать кодJAVASCRIPT

mongoose.connection.db.dropDatabase();

Всё необходимое для работы с базой уже есть: модель пользователя готова, библиотеки добавлены.

Ваша задача: описать before/after тесты:

* подключиться к БД перед тестированием;
* перед каждым тестом добавить данные в БД;
* после теста удалить данные;
* отключиться от БД после всех тестов.

Этот тест должен пройти несколько раз. Запускайте его командой npm run test:

Скопировать кодJAVASCRIPT

it('Пользователь должен быть', () => {

return User.findOne({ email: fixtures.user.email })

.then((user) => {

expect(user).toBeDefined();

expect(user.email).toBe(fixtures.user.email);

expect(user.name).toBe(fixtures.user.name);

});

});

Данные для тестирования возьмите из папки fixtures.

### По ссылке наше решение:

Данное решение покрывает все условия, но не является единственно верным!

[https://code.s3.yandex.net/web-code/Базы-данных-решение.zip](https://code.s3.yandex.net/web-code/%D0%91%D0%B0%D0%B7%D1%8B-%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85-%D1%80%D0%B5%D1%88%D0%B5%D0%BD%D0%B8%D0%B5.zip)

# Заключение

Вы познакомились с основами тестирования. Теперь изменять и дополнять существующую функциональность будет проще.

Также узнали о фреймворке для тестирования Jest и библиотеке для тестирования запросов supertest. С их помощью можно писать достаточно сложные тестовые наборы, в том числе и для баз данных.

Теперь вы знаете, как лучше подходить к тестированию: код нужно разбивать на юниты, тестировать каждый и после собирать в тестовые наборы.

Знания из этой темы помогут быстро разобраться в написанных тестах, если вы столкнётесь с ними на реальной работе. В целом, тестирование тесно связано с написанием самой функциональности. Существует даже подход, при котором к ещё не готовому коду сначала пишутся тесты.

Пора перейти к завершающей теме курса. В ней расскажем про деплой написанного приложения на удалённый сервер.