

ТЕСТИРОВАНИЕ И СИ



АЛЕКСАНДР ШЛЕЙКО



АЛЕКСАНДР ШЛЕЙКО

Программист в Яндекс



a.shleyko@yandex.ru



vk.com/shleiko



[@dustyo_0](https://t.me/@dustyo_0)



ПЛАН ЗАНЯТИЯ

1. [Unit-тестирование](#)
2. [Jest](#)
3. [Покрытие кода](#)
4. [Mock'и](#)
5. [Continuous Integration](#)



UNIT-ТЕСТИРОВАНИЕ

ЗАЧЕМ НУЖНЫ АВТО-ТЕСТЫ?



Зачем нужны авто-тесты, мне проще проверить руками



Зачем нужны авто-тесты, у меня итак всё работает



ДЕЙСТВИТЕЛЬНО, ЗАЧЕМ?

Авто-тесты - это возможность обезопасить себя от потенциальных ошибок (при создании нового кода или модификации существующего).

Это некая гарантия того, что то, что работало до этого - не сломалось, и то, что мы пишем сейчас - работает так, как мы задумываем.



BEST PRACTICES

В современном мире разработки написание авто-тестов считается одной из лучших практик создания поддерживаемого и качественного кода.



КАК И ЧТО ТЕСТИРОВАТЬ?

Тестирование - отдельная большая область знаний, со своими методами, подходами и теорией.

На самом базовом уровне: запускаем программу или отдельный её кусочек (например, функцию) и сравниваем полученный результат с тем, что должен был получиться.

Результат совпадает - всё ок, нет - ошибка.

Если начнёте с этого - будет уже большой шаг вперёд, после чего нужно ознакомиться с тест-анализом, тест-дизайном и комбинаторикой.



JEST



ФРЕЙМВОРК ТЕСТИРОВАНИЯ

Чтобы нам не делать этого каждый раз вручную, нам нужен инструмент, который будет запускать наши функции, сравнивать результат и собирать статистику.

Инструментов достаточно много, мы с вами будем рассматривать [Jest](#).

УСТАНОВКА

Для установки Jest выполним следующую команду:

```
$ npm install --save-dev jest babel-jest @babel/core @babel/cli @babel/preset-env  
$ npm install core-js@3
```

Пропишем скрипт `test`:

```
"scripts": {  
  "test": "jest",  
  "lint": "eslint .",  
},
```

POLYFILL

B `.babelrc`:

```
{  
  "presets": [["@babel/preset-env", {  
    "useBuiltIns": "usage",  
    "corejs": 3  
  }]]  
}
```

ОБЩИЙ ВИД ТЕСТА

```
1  test('<описание того, что проверяем>', () => {  
2    // Функция проверки  
3  
4    // 1. Выполняем нужные нам действия  
5    TODO:  
6  
7    // 2. Проверяем результат с помощью:  
8    expect(<что получили>).toBe(<что должно быть>);  
9  })
```

ПРИМИТИВНЫЙ ТЕСТ

```
1 test('should add two numbers', () => {  
2   const received = 1 + 1;  
3   const expected = 2;  
4  
5   expect(received).toBe(expected);  
6 })
```

ESLINT & JEST

Чтобы ESLint не ругался на Jest, мы можем либо добавить каталог с тестами в `.eslintignore` (плохая идея), либо прописать Jest в секцию `env` файла `.eslintrc.json`:

```
"env": {  
  ...  
  "jest": true  
}
```

ЗАПУСК АВТО-ТЕСТОВ

```
$ npm test
```

```
PASS test/app.test.js  
✓ shoud add two numbers (3ms)
```

```
Test Suites: 1 passed, 1 total  
Tests:      1 passed, 1 total  
Snapshots:  0 total  
Time:       0.762s, estimated 1s  
Ran all test suites.
```


FAIL

В случае, если тесты завершатся с ошибкой, мы увидим:

```
FAIL test/app.test.js
  ✕ should add two numbers (9ms)

  • should add two numbers

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

    Expected: 2
    Received: 4

       1 | test('shoud add two numbers', () => {
     > 2 |   expect(2 + 2).toBe(2);
         |                   ^
       3 | });
      at Object.toBe (test/app.test.js:2:17)
```

Test Suites: 1 failed, 1 total

Tests: 1 failed, 1 total

Snapshots: 0 total

Time: 0.78s, estimated 1s

Ran all test suites.

npm ERR! Test failed. See above for more details.

ЗАДАЧА

Перед нами стоит следующая задача: написать функцию, которая рассчитывает сумму покупок в магазине.

Покупки приходят в следующем виде:

```
1  [  
2    {id: 1, name: '...', price: 100, count: 3},  
3    {id: 2, name: '...', price: 55, count: 2},  
4  ]
```

Давайте попробуем написать эту функцию и авто-тесты для неё.

ФУНКЦИЯ

Начнём с самой простой реализации (файл `script.js`):

```
1  function calculateTotal(purchases) {  
2      let result = 0;  
3      for (const purchase of purchases) {  
4          result += purchase.price * purchase.count;  
5      }  
6  
7      return result;  
8  }
```

ТЕСТ

Подумаем, как должен выглядеть тест для этой функции:

```
1  test('should calculate total for purchases', () => {
2    const input = [
3      {
4        id: 1, name: '...', price: 33, count: 3,
5      },
6      {
7        id: 2, name: '...', price: 55, count: 2,
8      },
9    ];
10   const expected = 1099;
11
12   const received = calculateTotal(input);
13
14   expect(received).toBe(expected);
15 });
```



ПОВТОРЕНИЕ

СИСТЕМЫ МОДУЛЕЙ

Как вы уже знаете, на текущий момен наиболее распространёнными являются следующие системы модулей:

- **CommonJS** – система модулей, нативно поддерживается на платформе Node.js;
- **ES Modules** – система модулей, нативно поддерживаются в браузерах (текущий статус поддержки).

Под модулем мы будем понимать js-файл (достаточно упрощённое представление, но достаточное для нас на данном этапе).

Более подробно про модули мы поговорим на лекции, посвящённой модулям, сейчас же нам нужно понять ключевые моменты.



ЗАЧЕМ НАМ ДВЕ?

Большинство инструментов для JS написаны с использованием платформы Node.js, поэтому для них придётся использовать либо CommonJS, либо Babel (который обеспечит поддержку импорта в стиле ES Modules).

Поэтому придётся научиться использовать оба.

EXPORT / MODULE.EXPORTS

Если мы хотим сделать имя (функцию, переменную либо объект) доступным из нашего модуля, то в ES Modules:

```
1 export <some_name>;  
2 export function <some_function>() { ... };
```

В CommonJS:

```
1 module.exports = {  
2   <some_name>: <some_object>,  
3   <some_function>: function() { ... }  
4 };
```

IMPORT / REQUIRE

Если мы хотим использовать имя, экспортированное из другого модуля, в своём модуле, то в ES Modules:

```
import { <name> } from '<path_to_module>';
```

В CommonJS:

```
const <name> = require('<path_to_module>').<name>;
```

DEFAULT EXPORT

Если вы экспортируете из модуля всего одно имя, то лучше использовать default export:

```
export default function <some_function>() { ... }
```

```
import <some_function> from '<path_to_module>';
```



JEST & IMPORT/EXPORT

JEST IMPORT/EXPORT

Jest в связке с Babel у нас настроен таким образом, что поддерживает `import` / `export`, но для этого нужно экспортировать нашу функцию:

```
1  export default function calculateTotal(purchases) {  
2    let result = 0;  
3    for (const purchase of purchases) {  
4      result += purchase.price * purchase.count;  
5    }  
6  
7    return result;  
8  }
```

TECT

```
1  import purchasesTotal from '../src/js/script';
2
3  test('should calculate total for purchases', () => {
4      const input = [
5          {
6              id: 1, name: '...', price: 33, count: 3,
7          },
8          {
9              id: 2, name: '...', price: 55, count: 2,
10         },
11     ];
12     const expected = 209;
13
14     const received = calculateTotal(input);
15
16     expect(received).toBe(expected);
17 });
```



TESTS & GIT

Авто-тесты должны храниться вместе с нашим приложением, как и другие исходники.

Таким образом, любой участник нашей команды, меняя что-то в нашем приложении сможет удостовериться, что ничего не сломал.

ИСПОЛЬЗУЕМ `reduce`

Попробуем воспользоваться методом массива `reduce` вместо цикла `for..of`:

```
1  return purchases.reduce(  
2    (acc, curr) => acc + curr.price * curr.count,  
3    0,  
4  );
```

У нас есть авто-тесты, мы можем их запустить, чтобы удостовериться, что всё работает.



ВАЖНО

Важно понимать, что польза от авто-тестов появляется только тогда, когда вы их регулярно пишете и используете.

А кроме того, постоянно совершенствуетесь в навыке написания тестов.



ПОКРЫТИЕ КОДА

CODE COVERAGE

Code Coverage - метрика, показывающая, насколько наш код покрыт авто-тестами.

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

```
PASS test/script.test.js  
✓ should calculate total for purchases (4ms)
```

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	100	100	100	100	
script.js	100	100	100	100	

```
Test Suites: 1 passed, 1 total  
Tests: 1 passed, 1 total  
Snapshots: 0 total  
Time: 1.408s  
Ran all test suites.
```

ДОБАВИМ ЛОГИКУ

Пришло время модифицировать нашу функцию: в зависимости от переданного флага к итоговой сумме покупок должна применяться скидка 6.1%:

```
1 export default function calculateTotal(purchases, applyDiscount = false) {  
2   ...  
3  
4   if (applyDiscount) {  
5     return result * 0.939; // bad practice  
6   }  
7 }
```

ПОСМОТРИМ НА ПОКРЫТИЕ

PASS test/script.test.js
 ✓ should calculate total for purchases (7ms)

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	80	66.67	100	80	
script.js	80	66.67	100	80	8

Test Suites: 1 passed, 1 total
 Tests: 1 passed, 1 total
 Snapshots: 0 total
 Time: 1.963s
 Ran all test suites.

В каталоге `coverage/lcov-report` расположен отчёт о покрытии:

```

1  export default function purchasesTotal(purchases, applyDiscount = false) {
2  1x    const result = purchases.reduce(
3  2x      (accumulator, current) => accumulator + current.price * current.count,
4      0,
5      );
6
7  1x    if (applyDiscount) {
8      return result * 0.03;
9    }
10
11 1x    return result;
12  }
13

```

НАПИШЕМ ТЕСТ

```
1 test('should calculate total for purchases with discount', () => {  
2     ...  
3     const expected = 196.25;  
4  
5     const received = purchasesTotal(input, true);  
6  
7     expect(received).toBe(expected);  
8 });
```

Тест упал:

Expected: 196.25
Received: 196.25099999999998

```
31 |     const received = purchasesTotal(input, true);  
32 |  
> 33 |     expect(received).toBe(expected);  
    |                        ^  
34 |  
35 | });
```

MATCHERS

Jest нам предлагает различные виды проверок (не только на точное соответствие).

Полный перечень Matcher'ов можно найти на странице:

<https://jestjs.io/docs/en/expect>

В частности, в нашем случае хорошо бы подошёл `toBeCloseTo`.

КАК ПОНЯТЬ, ЧТО ТЕСТОВ ДОСТАТОЧНО?

Тесты должны помогать в разработке а не мешать. Именно они должны показывать, какие условия не протестированы, какие участки кода никогда не используются.

Значит:

1. Либо избыточны;
2. Либо мы не можем сказать, что они работают корректно.

Используйте подход TDD, который позволит уменьшить и количество разрабатываемого кода и количество разрабатываемых тестов.



МОСК'И



MOCKS

Как протестировать функцию, которая взаимодействует с внешним миром (HTTP, файловая система и т.д.)? Неужели на каждый тест будет выполняться отдельный HTTP-запрос на сервер?

Конечно же, нет. Для этого существуют Mock'и.

MOCKS

```
1  import { httpGet } from './http';
2
3  export default function loadUser(id) {
4    // bad practice
5    const data = httpGet(`http://server:8080/users/${id}`);
6    return JSON.parse(data);
7  }
```

MOCKS

```
1  import loadUser from '../src/js/user';
2  import { httpGet } from '../src/js/http';
3
4  jest.mock('../src/js/http');
5
6  beforeEach(() => {
7    jest.resetAllMocks();
8  });
9
10 test('should call loadUser once', () => {
11   httpGet.mockReturnValue(JSON.stringify({}));
12
13   loadUser(1);
14   expect(httpGet).toBeCalledWith('http://server:8080/users/1');
15 });
```



SETUP & TEARDOWN

- `beforeEach;`
- `afterEach;`
- `beforeAll;`
- `afterAll.`

<https://jestjs.io/docs/en/setup-teardown>



MOCKS

Использование mock'ов - не всегда хорошая идея, т.к. влечёт к избыточному усложнению тестового кода.

JEST EXTENSION

Для VSCode предоставляется плагин Jest за авторством Orta, который в автоматическом режиме перезапускает ваши тесты и отображает статус:

```
●test('shoud add two numbers', () => {  
  | expect(1 + 1).toBe(2);  
});
```

А также позволяет отлаживать их:

```
Debug  
●test('shoud add two numbers', () => {  
  | expect(2 + 2).toBe(2); // Expected: 2, Received: 4  
});
```



CONTINUOUS INTEGRATION

CONTINUOUS INTEGRATION



Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible.

Martin Fowler



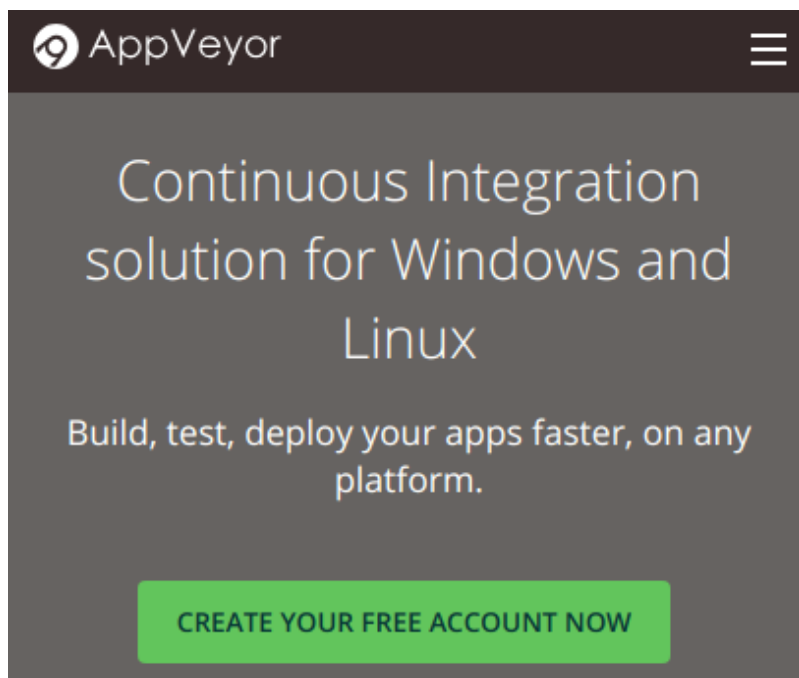
CONTINUOUS INTEGRATION (CI)

Мы будем рассматривать как практику, при которой для каждого изменения кода (`git push`) должен автоматически запускаться конвейер тестирования и сборки (automated build).

Тестирование должно запускаться автоматически.

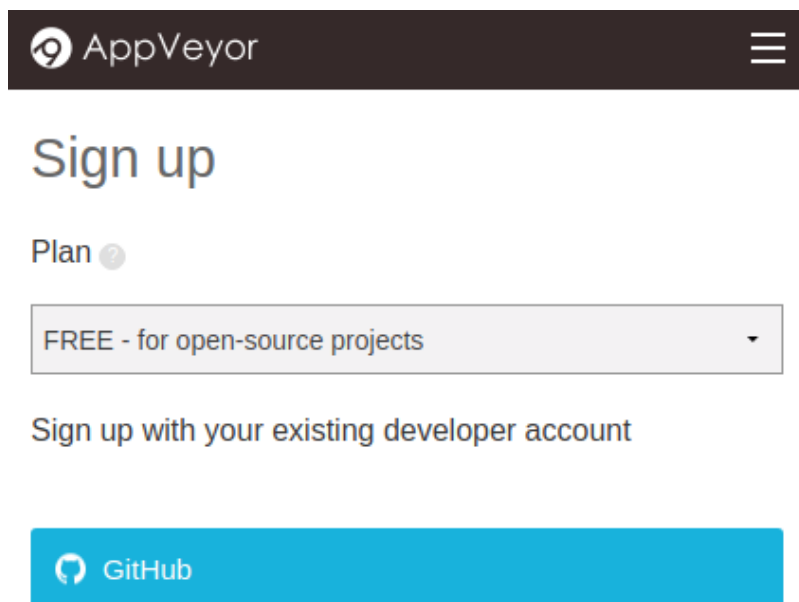
CI-СЕРВЕР

[AppVeyor](#) - одна из платформ, предоставляющих функциональность Continuous Integration. В базовом варианте - бесплатно.





APPEVEYOR + GITHUB


Бесплатный тарифный план для публичных репозиторий GitHub (авторизация - также через GitHub):




The image shows a screenshot of the AppVeyor website's sign-up page. At the top is a dark header with the AppVeyor logo and a hamburger menu icon. Below the header, the text 'Sign up' is displayed. Underneath, there is a 'Plan' label with a help icon, followed by a dropdown menu currently showing 'FREE - for open-source projects'. Below the dropdown, the text 'Sign up with your existing developer account' is visible. At the bottom of the form area is a blue button with the GitHub logo and the text 'GitHub'.


 AppVeyor 

Sign up

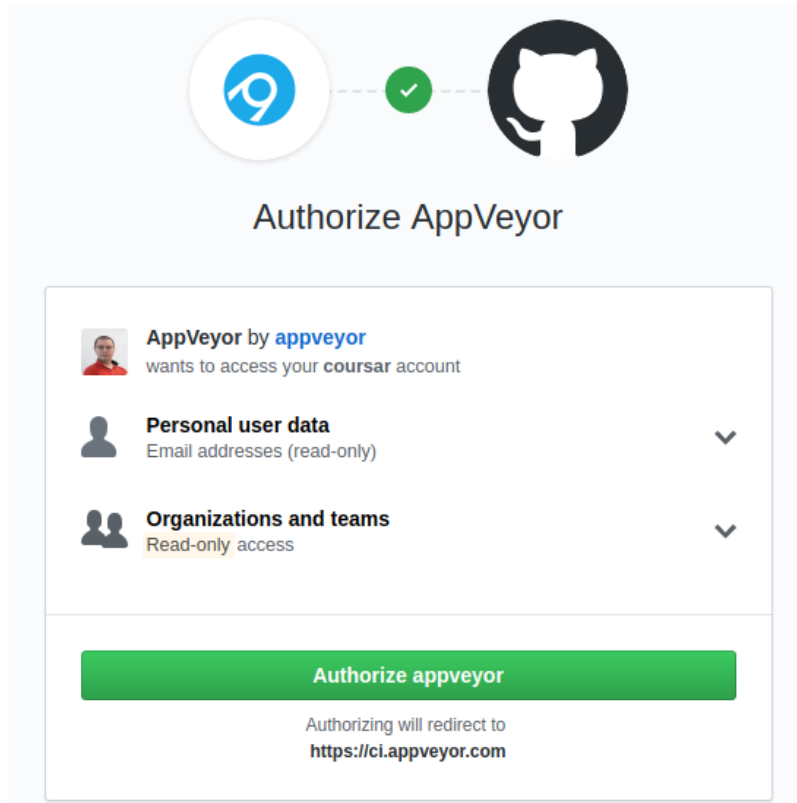
Plan 

FREE - for open-source projects 

Sign up with your existing developer account

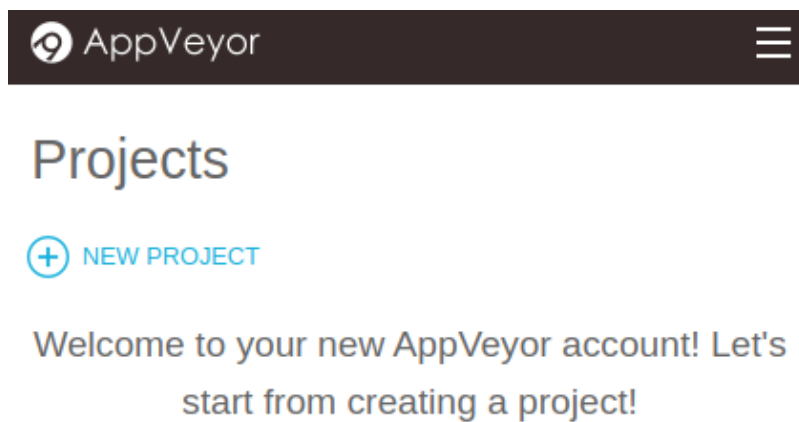
 GitHub

APPVEYOR + GITHUB



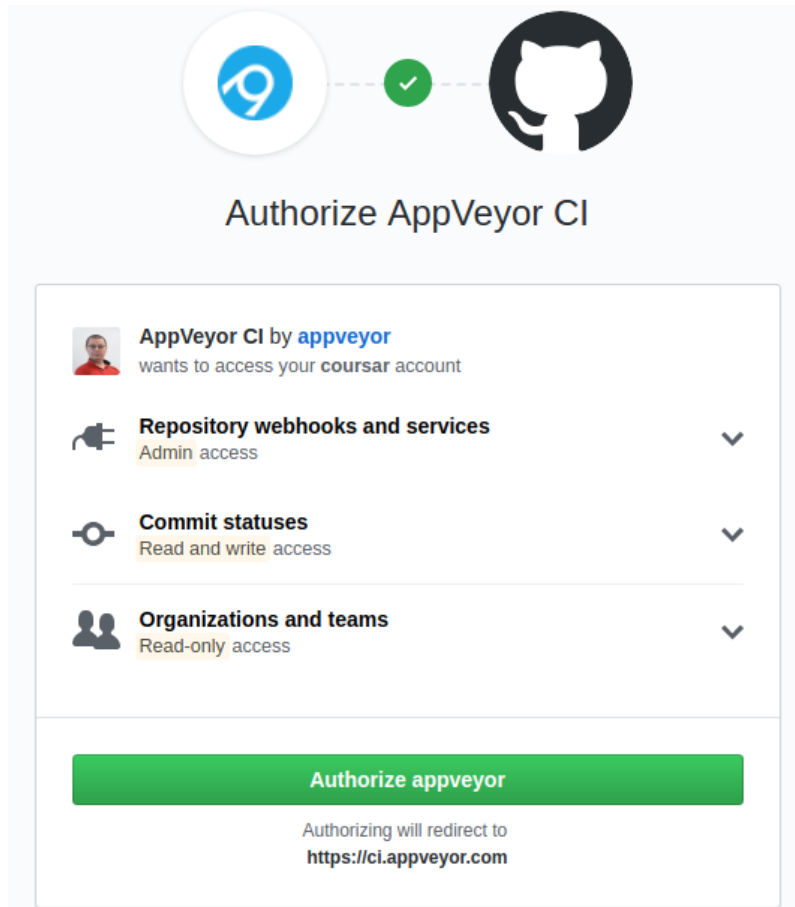
СОЗДАНИЕ ПРОЕКТА

После авторизации станет доступной панель управления, где можно создать новый проект:



Это даст возможность приложению получать уведомления о ваших `git push` в репозиторий, модификации и т.д.

OAUTH APP





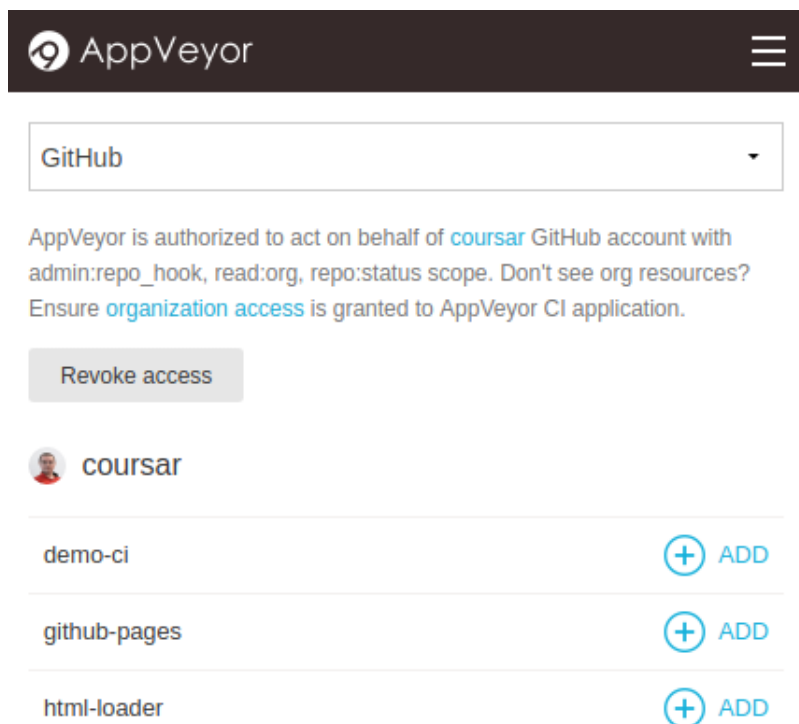
OAUTH

Детальнее об OAuth вы можете прочитать на:

- <https://oauth.net/2/>
- <https://auth0.com/docs/protocols/oauth2>

ВЫБОР РЕПОЗИТОРИЯ

После авторизации достаточно будет нажать кнопку **ADD** напротив необходимого репозитория:



The screenshot shows the AppVeyor web interface. At the top is a dark header with the AppVeyor logo and a hamburger menu icon. Below the header is a dropdown menu currently showing 'GitHub'. A text block below the dropdown states: 'AppVeyor is authorized to act on behalf of [coursar](#) GitHub account with admin:repo_hook, read:org, repo:status scope. Don't see org resources? Ensure [organization access](#) is granted to AppVeyor CI application.' Below this text is a 'Revoke access' button. Underneath is a section for the user 'coursar', which contains a list of repositories. Each repository row includes the repository name and a blue circular button with a plus sign and the word 'ADD'.

Repository Name	Action
demo-ci	+ ADD
github-pages	+ ADD
html-loader	+ ADD

GITHUB TOKEN

Для программного взаимодействия с GitHub существует возможность генерации токенов (вместо указания логина и пароля).

Токены позволяют относительно безопасно их использовать на сторонних сервисах с возможностью отзыва (и без компрометации основного пароля аккаунта).

Перейдите по адресу <https://github.com/settings/tokens> и нажмите на кнопку `Generate New Token`.

GITHUB TOKEN SCOPE

Необходимо выдать "права" только на `repo`.

Token description

appveyor

What's this token for?

Select scopes

Scopes define the access for personal tokens. [Read more about OAuth scopes.](#)

- | | |
|---|--------------------------------------|
| <input checked="" type="checkbox"/> repo | Full control of private repositories |
| <input checked="" type="checkbox"/> repo:status | Access commit status |
| <input checked="" type="checkbox"/> repo_deployment | Access deployment status |
| <input checked="" type="checkbox"/> public_repo | Access public repositories |
| <input checked="" type="checkbox"/> repo:invite | Access repository invitations |

GITHUB TOKEN

Обязательно скопируйте значение токена (GitHub больше его не покажет):

Personal access tokens

Generate new token

Revoke all

Tokens you have generated that can be used to access the [GitHub API](#).

Make sure to copy your new personal access token now. You won't be able to see it again!

[illegible]

Delete

APPEVEYOR & GITHUB TOKEN

Хранить токен мы будем в переменных окружения.

Для этого необходимо для конкретного проекта зайти на вкладку **Settings** -> **Environment** и прописать переменную окружения (**Environment Variables**) **GITHUB_TOKEN** с тем значением, что вы получили на предыдущем шаге:

The screenshot shows the Appveyor web interface. At the top, there are tabs: 'Current build', 'History', 'Deployments', 'Events', and 'Settings' (which is selected and underlined). On the left side, there is a sidebar with a list of settings categories: 'General', 'Environment' (highlighted with a blue bar), 'Build', 'Tests', 'Artifacts', 'Deployment', 'NuGet', and 'Notifications'. The main content area is titled 'Build worker image' and shows a dropdown menu with 'Ubuntu' selected. Below this is an 'Add image' button. The next section is 'Clone directory' with a text input field containing 'Optional, e.g. c:\projects\myproject'. The final section is 'Environment variables', which has a plus icon. It shows a table with one row: 'GITHUB_TOKEN' in the name column and a masked value (dots) in the value column. Below this table is an 'Add variable' button.

Environment variables	
GITHUB_TOKEN

CONFIGURATION AS CODE

Поскольку вручную настраивать каждый проект в системе CI - лишняя трата времени, мы будем хранить всю конфигурацию для AppVeyor в специальном файле с названием `.appveyor.yml`

Файл этот должен храниться в самом репозитории на GitHub, тогда AppVeyor будет автоматически подхватывать настройки из него:

 `.appveyor.yml` `.gitignore` `README.md` `package-lock.json` `package.json`



YAML

Формат сериализации данных, используемый многими системами для хранения конфигурации.

Ссылки:

- [Wikipedia](#)
- [Спецификация](#)

Странички на Wikipedia достаточно для понимания базовых конструкций языка.



CMD, PS, BASH

AppVeyor позволяет использовать все три оболочки, но мы для простоты на Linux будем использовать Bash, на Windows - CMD.

LINUX CONFIG

```
image: Ubuntu1804 # образ для сборки

stack: node 10 # окружение

branches:
  only:
    - master # ветка git

cache: node_modules # кеширование

install:
  - npm install # команда установки зависимостей

build: off

test_script:
  - npm run lint && npm test # скрипт сборки
```




WINDOWS CONFIG

```
image: Visual Studio 2015
```

```
stack: node 10
```

```
branches:
```

```
  only:
```

```
    - master
```

```
cache: node_modules
```

```
install:
```

```
  - npm install
```

```
build: off
```

```
test_script:
```

```
  - npm run lint && npm test
```



APPVEYOR.YAML

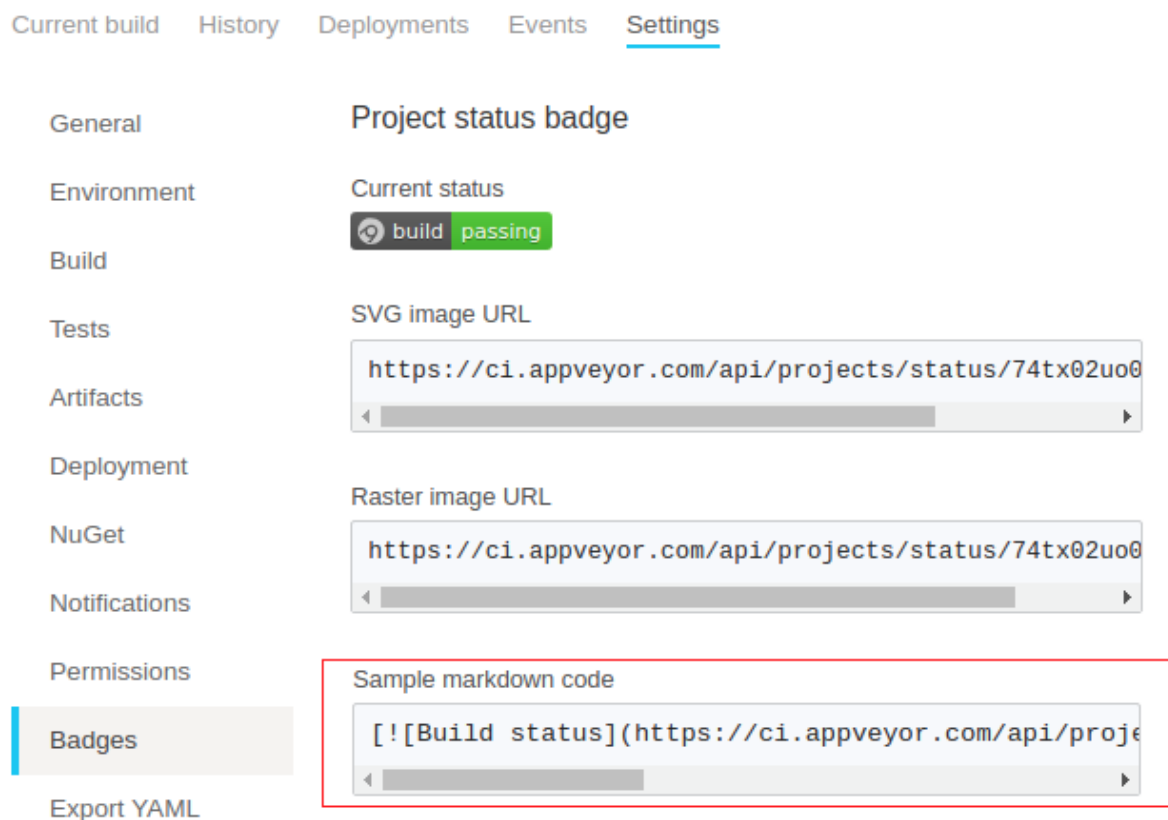
Полное описание формата находится по ссылке:

<https://www.appveyor.com/docs/appveyor-yml/>.

Для простоты мы не рассматриваем всех возможностей, в том числе сборки сразу на нескольких платформах.

STATUS BADGE

На странице **Settings** - **Badges** Appveyor предлагает код для "бейджика" статуса вашего проекта:



Current build History Deployments Events Settings

General Project status badge

Environment Current status

Build build passing

Tests SVG image URL

Artifacts `https://ci.appveyor.com/api/projects/status/74tx02uo0`

Deployment Raster image URL

NuGet `https://ci.appveyor.com/api/projects/status/74tx02uo0`

Notifications

Permissions

Badges

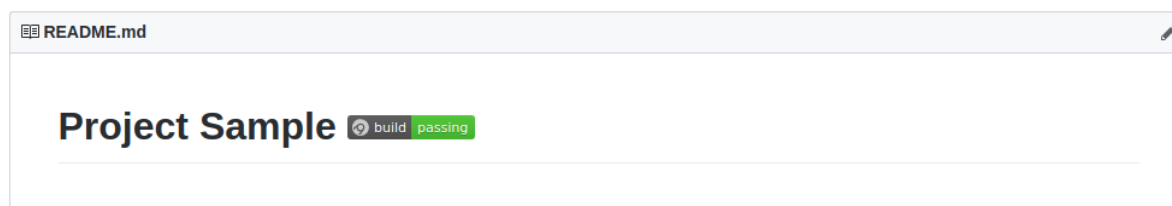
Export YAML

Sample markdown code

```
[![Build status](https://ci.appveyor.com/api/proje
```

STATUS BADGE

Этот badge необходимо разместить в файле `README.md` для отображения текущего статуса вашего проекта:





ИТОГИ



ИТОГИ

Сегодня мы с вами рассмотрели достаточно много важных вещей, а именно:

1. Jest — позволяет создавать авто-тесты.
2. CI — позволяет автоматизировать процесс тестирования и сборки.

ВАЖНО

Начиная с сегодняшнего дня во всех домашних заданиях мы будем требовать от вас:

1. Наличия авто-тестов на разрабатываемые функции
2. 100% покрытия тестируемых функций по строкам
3. Отсутствия ошибок ESLint
4. Использования CI для тестирования и сборки ваших проектов

Не забывайте выставлять Status Badge в README.md вашего проекта.



ПОЛЕЗНЫЕ ССЫЛКИ

- [Документация Jest](#)
- [Использование matcher'ов](#)
- [expect](#)



Задавайте вопросы и напишите отзыв о лекции!

АЛЕКСАНДР ШЛЕЙКО

 a.shleyko@yandex.ru

 vk.com/shleiko

 [@dustyo_0](https://t.me/@dustyo_0)