**Введение**

В этой теме вы развернёте приложение на сервере, чтобы к нему можно было обратиться из внешнего мира.

Проект разросся, и отдельную его часть трудно воспринимать в отрыве от всего кода. Поэтому рекомендуем открыть проектную работу и проходить уроки с ней перед глазами.

Вперёд.

# Сбор логов

В предыдущих спринтах вы застраховали свой бэкенд от ошибок. А позже улучшили его централизованным обработчиком. Но всех ошибок не избежать.

Представьте ситуацию: пользователь написал в поддержку небольшой социальной сети, бэкенд которой разрабатываете вы. Пользователь пожаловался, что у него не работает удаление постов. Агент поддержки описал вам проблему. Вы тут же проверили функцию удаления постов и у вас всё сработало корректно — пост удалился.

Что сделать в этой ситуации? Можно ответить пользователю фразой «у нас всё работает», но это не решит проблему. Расстроенный, он уйдёт к конкурентам.

Как проанализировать, почему в запросе пользователя происходит ошибка, если у нас нет доступа к его компьютеру? Что делать в ситуации, если сервер ведёт себя не так, как мы от него ожидаем?

В этом помогут логи.

## Что такое логи

Логи — информация о том, что происходит на сервере. В простейшем случае — это журнал записи запросов к серверу и его ответов на эти запросы. Если мы будем записывать запросы и ответы, то в случае какого-то отклонения от нормальной работы сможем проанализировать, что пошло не так. Логирование поможет лучше понимать, что происходит на сервере. Реализуем его.

## Winston

Есть много решений для сбора логов. Мы воспользуемся одним из самых популярных и гибких — библиотекой для логирования winston. Чтобы с ней было удобно работать в express, нам понадобится мидлвэр express-winston. После установки этих двух пакетов (‘npm i winston' и ‘npm i express-winston’), их можно импортировать. Всю работу будем вести в файле middlewares/logger.js:

Скопировать кодJAVASCRIPT

*// middlewares/logger.js*

*// импортируем нужные модули*

const winston = require('winston');

const expressWinston = require('express-winston');

После этого нужно создать логгер. Мы будем логировать два типа информации — запросы к серверу и ошибки, которые на нём происходят. Для создания логгера запросов воспользуемся функцией logger модуля expressWinston:

Скопировать кодJAVASCRIPT

*// middlewares/logger.js*

const winston = require('winston');

const expressWinston = require('express-winston');

*// создадим логгер запросов*

const requestLogger = expressWinston.logger({

transports: [

new winston.transports.File({ filename: 'request.log' }),

],

format: winston.format.json(),

});

Обратите внимание на опции, которые мы передаём при создании. Опция transports отвечает за то, куда нужно писать лог. В нашем случае это файл request.log. Также transports — массив, в него можно записать и другие транспорты. Например, логи можно писать в консоль или в сторонний сервис аналитики, но мы ограничимся файлом. Вторая опция — format отвечает за формат записи логов. Мы указали json, потому что его удобно анализировать.

Логгер запросов создан, теперь сделаем логгер ошибок. Он нужен, чтобы в случае возникновения ошибки в лог записывалась информация о ней — имя ошибки, сообщение и её стектрейс. Логгер ошибок создаётся методом errorLogger модуля expressWinston:

Скопировать кодJAVASCRIPT

*// middlewares/logger.js*

const winston = require('winston');

const expressWinston = require('express-winston');

const requestLogger = expressWinston.logger({

transports: [

new winston.transports.File({ filename: 'request.log' }),

],

format: winston.format.json(),

});

*// логгер ошибок*

const errorLogger = expressWinston.errorLogger({

transports: [

new winston.transports.File({ filename: 'error.log' }),

],

format: winston.format.json(),

});

Ошибки мы пишем в отдельный файл — error.log.

После создания логгеров их нужно экспортировать:

Скопировать кодJAVASCRIPT

module.exports = {

requestLogger,

errorLogger,

};

А затем импортировать в app.js:

Скопировать кодJAVASCRIPT

const { requestLogger, errorLogger } = require('./middlewares/logger');

Пришло время подключить логгеры как мидлвэр. Логгер запросов нужно подключить до всех обработчиков роутов:

Скопировать кодJAVASCRIPT

app.use(requestLogger); *// подключаем логгер запросов*

*// за ним идут все обработчики роутов*

app.post('/signup', createUser);

app.post('/signin', login);

app.use('/users', usersRouter);

app.post('/posts', postsRouter);

errorLogger нужно подключить после обработчиков роутов и до обработчиков ошибок:

Скопировать кодJAVASCRIPT

app.use(logger);

app.post('/signup', createUser);

app.post('/signin', login);

app.use('/users', usersRouter);

app.post('/posts', postsRouter);

app.use(errorLogger); *// подключаем логгер ошибок*

app.use(errors()); *// обработчик ошибок celebrate*

*// централизованный обработчик ошибок*

app.use((err, req, res, next) => {

*// ...*

});

Логгеры подключены. Теперь при запросе к серверу создастся файл request.log, в него в формате json запишется часть данных из запроса и ответа. Если на сервере произойдёт ошибка, создастся файл error.log, туда запишется часть данных запроса и ответа, и информация об ошибке. Это будет только часть данных, ведь в лог не запишется тело запроса. По умолчанию это сделано так, потому что в теле может содержаться конфиденциальная информация, например, пароль пользователя, и записывать её в файл в незашифрованном виде небезопасно. Такое поведение можно изменить. Если это потребуется в вашем проекте, разберитесь в документации самостоятельно по ссылкам в конце урока.

## gitignore

Последний момент: не забудьте добавить файлы логов в .gitignore, чтобы они не добавлялись в репозиторий. Сделать это можно с помощью одной строчки в .gitignore:

Скопировать код

\*.log

Теперь все файлы с расширением log невидимы для git.

## Ссылки

Документация express-winston: <https://www.npmjs.com/package/express-winston>.

Документация winston: <https://github.com/winstonjs/winston>.

Подробный туториал по winston: <https://stackify.com/winston-logging-tutorial/>.

**Создаём удалённый сервер**

До настоящего момента мы разрабатывали бэкенд локально: доступ к нему был только у нас. Но рано или поздно наступает момент, когда код нужно «выкатывать в продакшн», то есть загружать на публичный сервер, чтобы им могли пользоваться и другие люди.

Есть несколько вариантов, где разместить код. Первый — купить железо и поставить сервер у себя в гараже.

В этом случае придётся самостоятельно настраивать сеть, заниматься безопасностью и поддержкой стабильности сервера. Если вы не хотите этого делать, есть вторая опция — аренда облачного сервера.

Это готовый сервер, который можно создать и сконфигурировать под себя за несколько минут. При этом его поддержкой, настройкой и безопасностью занимается компания, которая сдаёт этот сервер в аренду. Вот несколько популярных решений облачных серверов:

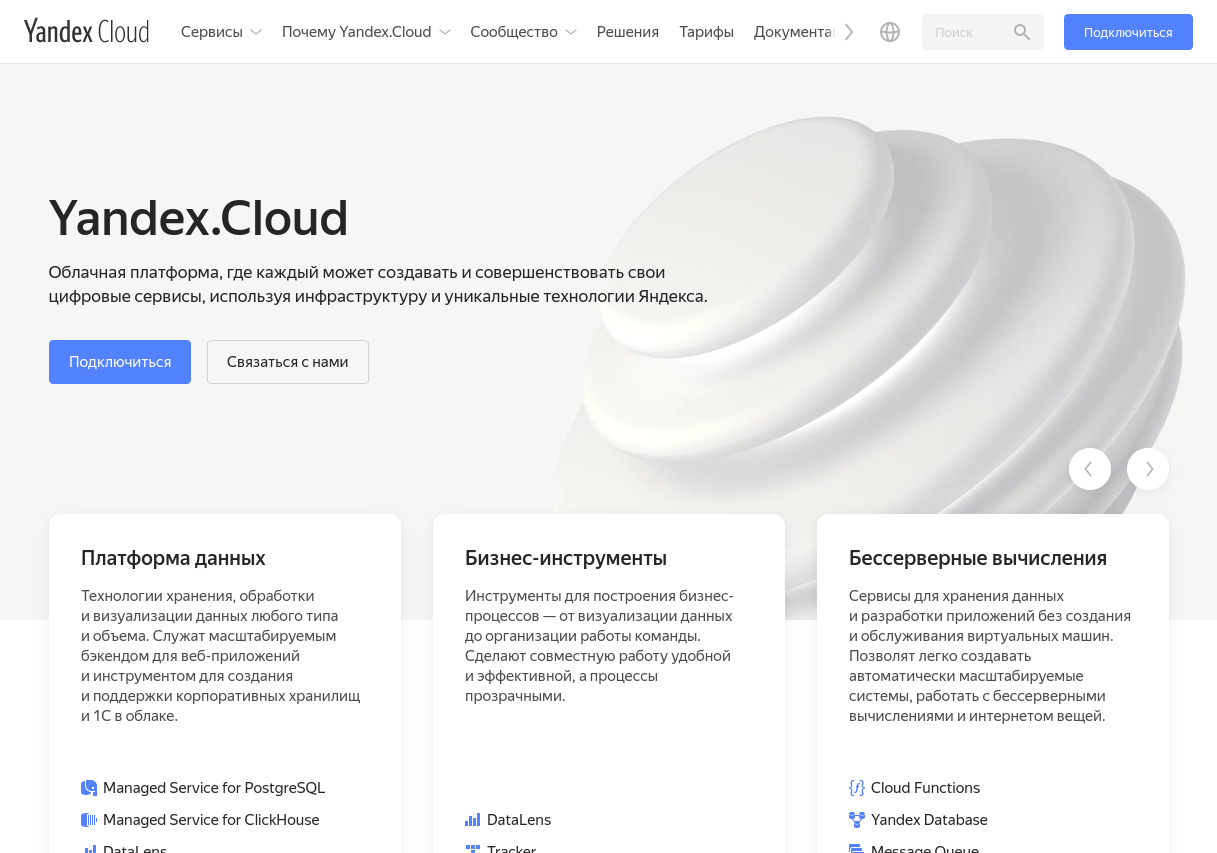
* Amazon Web Services,
* Google Compute Cloud,
* Яндекс.Облако.

Мы воспользуемся последним по двум причинам:

* Яндекс.Облако предоставляет грант на бесплатное использование сервиса. Этот грант полностью покроет нужды нашего обучения.
* Сервера Яндекс.Облака соответствуют [закону о персональных данных](http://www.consultant.ru/document/cons_doc_LAW_61801/), и нам не придётся об этом думать.

## Создайте аккаунт в Яндекс.Облаке

Зайдите на [https://cloud.yandex.ru](https://cloud.yandex.ru/) и залогиньтесь через аккаунт Яндекса. Затем [войдите в консоль](https://console.cloud.yandex.ru/):



После регистрации вы можете активировать пробный период. На это время Яндекс.Облако предоставляет вам грант на 4000 рублей. Он действует 60 дней и разбит на 2 части: 1000 и 3000 рублей. 1000 рублей можно потратить на создание виртуальных машин, а остальные 3000 — на другие сервисы облака.

Для деплоя бэкенда будем использовать виртуальные машины Яндекс.Облака, то есть на наши нужды есть 1000 рублей стартового гранта. Остальным можете распоряжаться на ваше усмотрение.

Подробная информация о предоставлении гранта: <https://cloud.yandex.ru/docs/free-trial/concepts/usage-grant>.

## Важная пометка: аккуратно прочитай про настройку ВМ и настрой корректное расходование гранта, чтобы суммы хватило на бо́льший срок. По ссылке: <https://disk.yandex.ru/i/3mRtcBD0Hkc0uA> – наши советы по настройке. Обращаем твое внимание заранее на то, что сумма гранта, а значит и работа ВМ – ограничены и по окончании этого промо тебе нужно будет самостоятельно организовать оплату сервиса или воспользоваться альтернативным, например, бесплатным сервисом heroku.Это твой промокод:dn2lu8rm23tddk9535qs

Кроме стартового гранта от Яндекс.Облака мы единоразово предоставляем промокод на 4000 рублей. Чтобы его получить, обратитесь к комьюнити-менеджеру в Slack.

Дополнительный грант можно активировать в течение двух месяцев с момента выдачи. А использовать его можно в течение 90 дней. Чтобы активировать промокод, зайдите в [список ваших платёжных аккаунтов](https://console.cloud.yandex.ru/billing) и выберите тот, который создали в этом уроке.

Мы рекомендуем активировать его немного позже. Например, в начале дипломного проекта. Главное, сделать это не позднее 60 дней с начала текущего спринта.

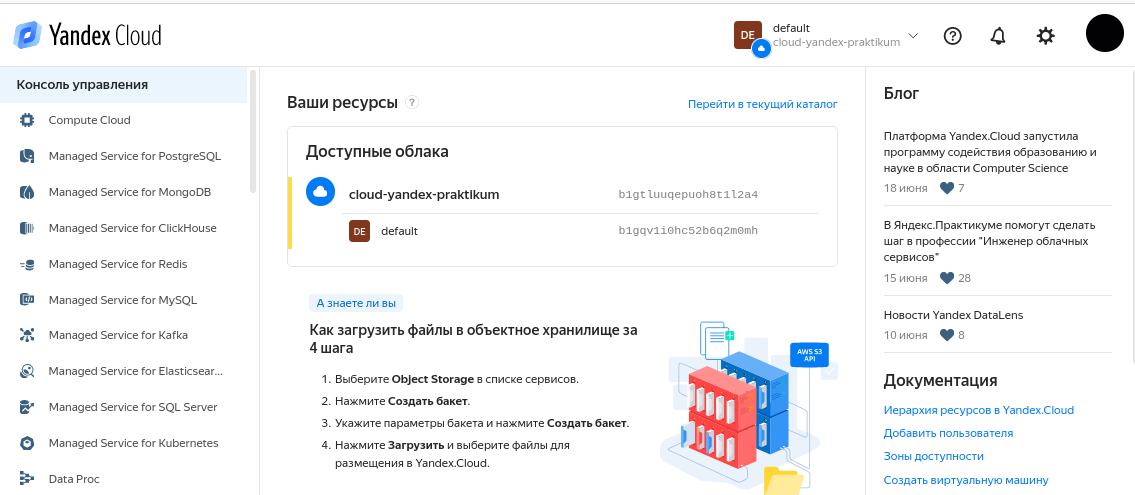
## Создайте платёжный аккаунт

Чтобы пользоваться Яндекс.Облаком, нужно создать платёжный аккаунт. Для этого перейдите [в раздел «Billing»](https://console.cloud.yandex.ru/billing) и нажмите «Создать аккаунт». В открывшемся окне введите информацию о себе, данные карты и нажмите «Активировать».

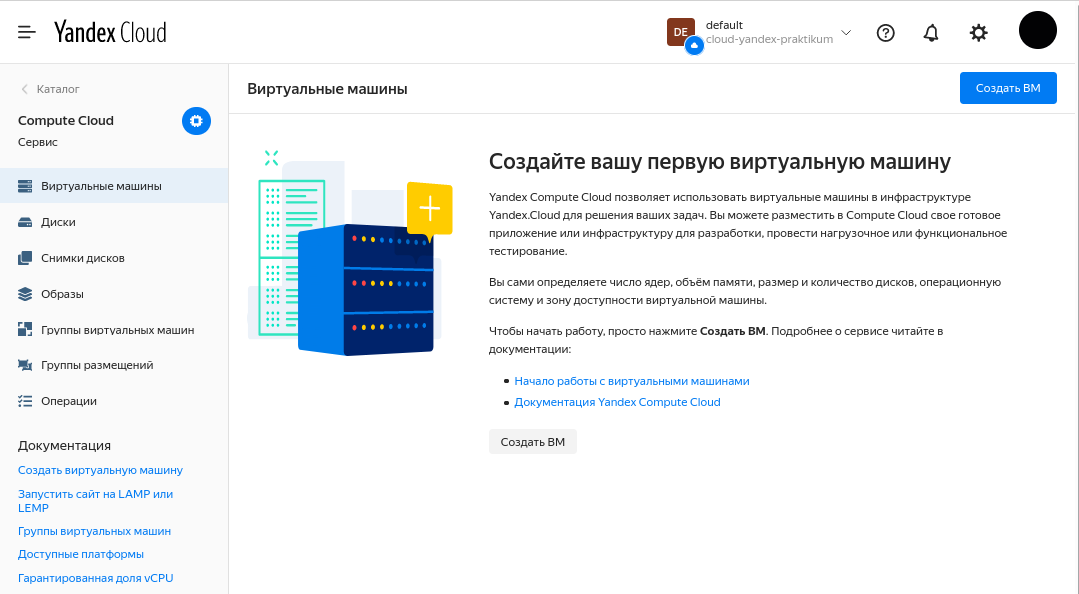
Мы понимаем скепсис по поводу ввода данных карты. Но для нас это необходимо, чтобы создать сервер и задеплоить на него бэкенд. Стартового гранта в 1000 рублей хватит примерно на 2 месяца, это полностью покрывает срок нашего обучения. Дальше вы можете остаться в Облаке или выбрать другую платформу.

## Создайте виртуальную машину

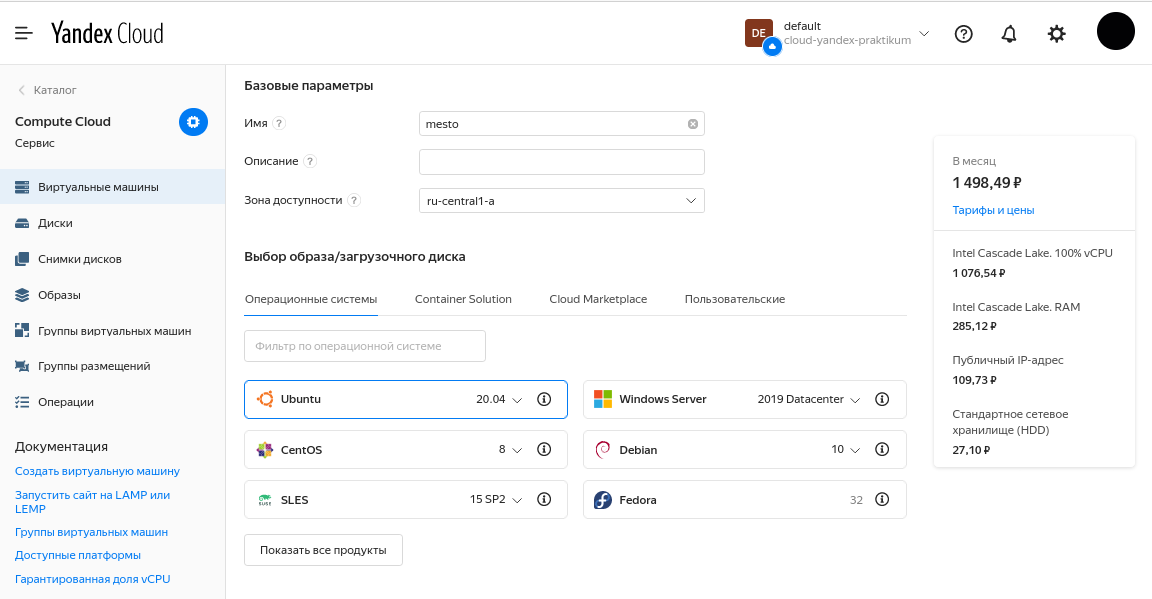
Чтобы начать создавать виртуальную машину, зайдите [в консоль Яндекс.Облака](https://console.cloud.yandex.ru/). Вам понадобится сервис Compute Cloud. В левом меню кликните на него:



Теперь нужно создать виртуальную машину. Кликните на «Создать ВМ»:



Введите название проекта на ваш выбор. В качестве публичного образа выберите Ubuntu 20.04:

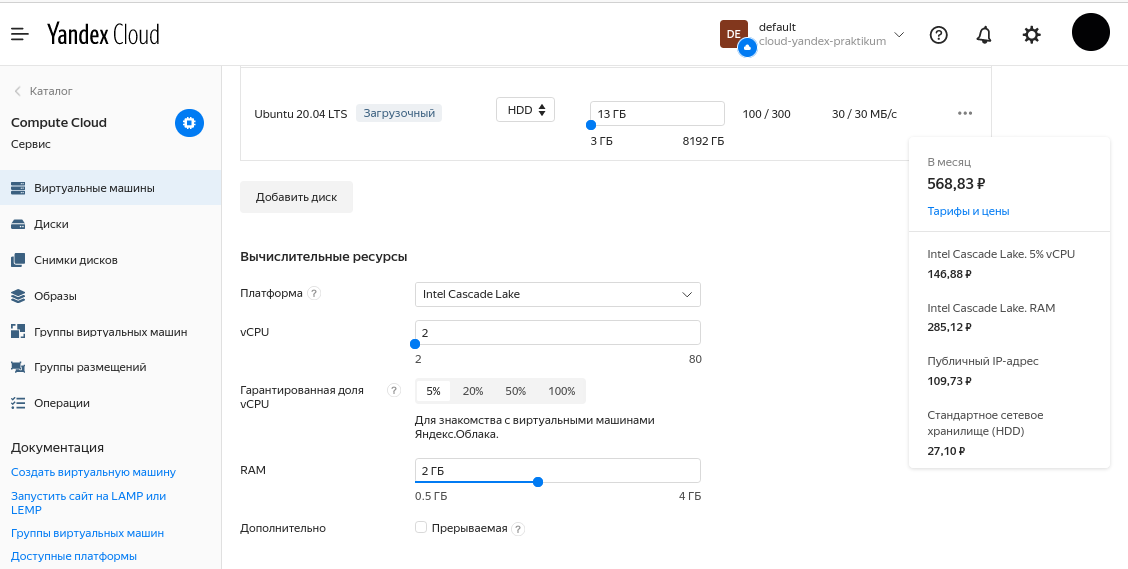


Дискам и вычислительным ресурсам оставьте настройки по умолчанию, для начала этого хватит. А вот для параметра «Гарантированная доля vCPU» необходимо проставить значение 5%.

vCPU — облачный процессор вашего сервера, а его гарантированная доля — минимальный процент мощности. Это не значит, что у сервера всегда будет мощность в 5% от полной — нередко она будет близка к максимальной по тарифу. Показатель говорит лишь о том, что в момент, когда ресурсы нашего сервера не используются на полную, Яндекс.Облако может предоставить их другому клиенту.

Это бывает нежелательно при использовании в продакшене: нагрузка на сервис может резко возрасти, и если Яндекс.Облако предоставит ресурсы с задержкой, сервис начнёт тормозить у клиентов. Но для учебных целей и экспериментальных проектов такой подход удобен: большую часть времени сервер будет использоваться только нами и можно заметно сэкономить, если не требовать у Облака полностью держать ресурсы под сервер.

После этого изменения стоимость аренды сервиса заметно снизится:



Было почти 1500 рублей, стало — чуть больше 550 рублей

Ваши настройки должны совпадать с теми, которые указаны на скриншоте. Если что-то отличается, приведите их в соответствие c ним.

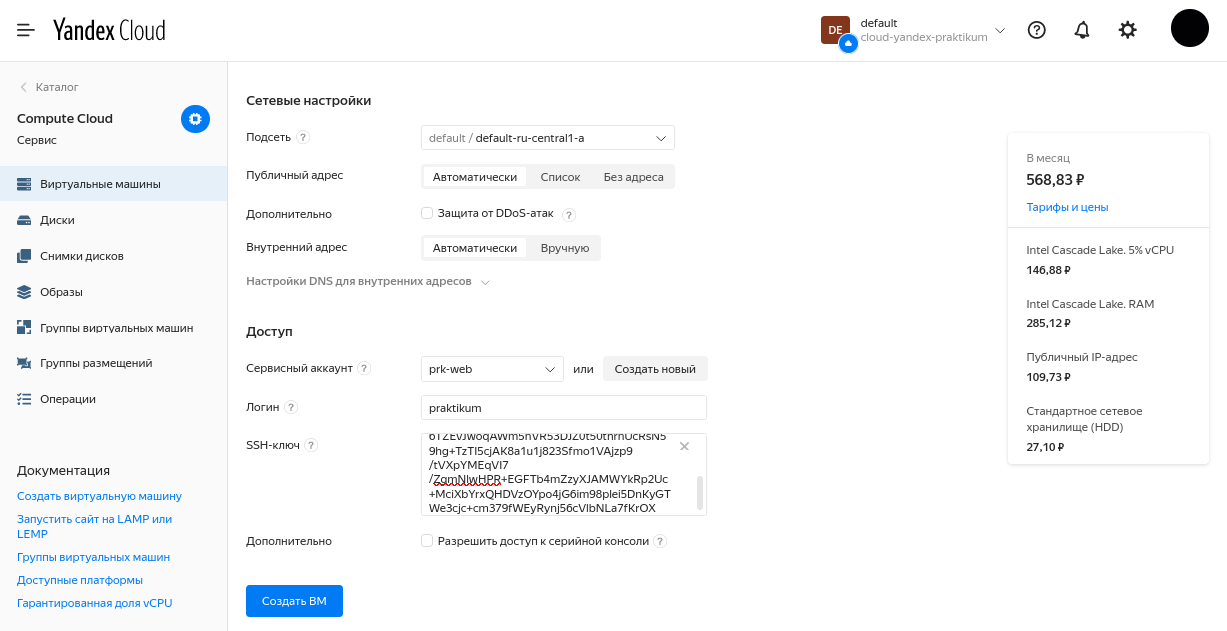
В блоке «Доступ» в поле «Логин» введите имя пользователя, по которому будете заходить на сервер. Оно может быть произвольным, но должно соответствовать правилам, которые можно прочитать, если навести курсор на знак вопроса слева от поля.

У сервера может быть несколько пользователей с разными паролями, ключами доступа и правами. Когда добавляется новый пользователь, для него автоматически создаётся специальная домашняя директория, которая предназначена для хранения личных файлов и конфигов.

Обычно она располагается по пути /home/<имя\_пользователя>. Для домашней директории также придумали специальное сокращение — ~ (тильда), которое можно использовать вместо полного пути.

Во время выполнения проектной работы вам потребуется разместить код в домашней папке, поэтому мы ещё вернёмся к этой теме.

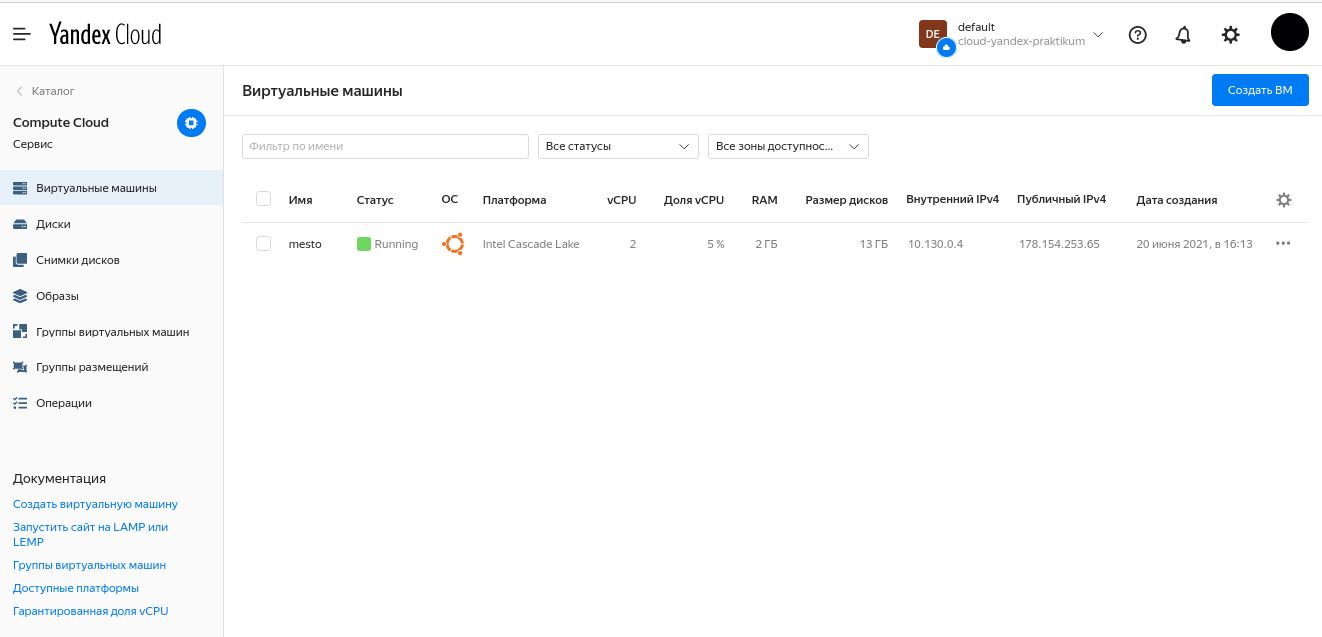
В поле «SSH-ключ» скопируйте ваш SSH-ключ, он понадобится для подключения к серверу:



Важно: SSH-ключ, который вы указываете в Яндекс.Облаке, должен совпадать с ключом доступа в GitHub-репозиторий с проектом.

Вы можете ввести SSH-ключ из темы про Github или создать новый. Если забыли, как это делается, перечитайте [этот урок](https://praktikum.yandex.ru/trainer/web/lesson/547966f0-d2df-4f29-8408-4f24fdb29918).

После того как всё сделали, нажмите «Создать BM». Облаку понадобится какое-то время, чтобы создать машину. Примерно через минуту статус изменится на “Running” — это значит, что машина создана:



## Подключение к серверу

Чтобы подключиться к серверу, понадобится публичный IP и логин, который мы указали при создании виртуальной машины.

Скопируйте публичный IP, зайдите в командную строку и введите команду (замените praktikum и 84.201.130.70 на свои логин и IP):

Скопировать кодBASH

ssh praktikum@84.201.130.70

Если это первое подключение к серверу, вы увидите фразу:

Скопировать кодBASH

The authenticity of host '84.201.130.70 (84.201.130.70)' can't be established.

ECDSA key fingerprint is SHA256:gGz1AULJpNptRRaqLz2FQTDf/IRxSGPA0vvmmXWy/6I.

Are you sure you want to continue connecting (yes/no)?

Введите “yes” и нажмите Enter. Вам будет предложено ввести пароль от приватного ключа. Введите его и снова нажмите Enter. Если пароль правильный, вы окажетесь на удалённом сервере:



Отлично, вы на сервере! Освойтесь: введите несколько команд. А в следующем уроке мы установим всё необходимое для работы нашего API. Чтобы выйти из сервера, воспользуйтесь сочетанием клавиш Ctrl+D.

# Как предоставить SSH-доступ другому разработчику

Мы часто сталкиваемся с необходимостью предоставить доступ к нашему серверу другому разработчику. Например, когда работаем в команде и хотим, чтобы каждый из её участников мог управлять приложением, читать логи или конфигурировать сервер.

Как вы знаете из предыдущих уроков, для удалённого управления сервером разработчики используют протокол SSH с аутентификацией при помощи ключей. В этом уроке научимся управлять SSH-ключами и разберёмся, как реализуется аутентификация с их использованием.

## Узнаём публичный SSH-ключ

Для начала следует узнать публичный SSH-ключ пользователя, которому мы предоставляем доступ. Ключи обычно хранятся в домашней папке пользователя директории .ssh. Для публичных ключей используются файлы с расширением .pub (по умолчанию это id\_rsa.pub).

Если владелец не менял стандартное имя ключа, его можно узнать с помощью такой команды:

Скопировать кодBASH

cat ~/.ssh/id\_rsa.pub

Если ключа пока нет, его нужно создать. Мы рассматривали процесс генерации и работы с публичными ключами [в уроке о SSH](https://praktikum.yandex.ru/trainer/web/lesson/547966f0-d2df-4f29-8408-4f24fdb29918).

После этого ключ следует сохранить в удобное место в файл your-new-key.pub.

Второй шаг — прописать SSH-ключ на удалённом сервере. Это можно сделать двумя способами: при помощи утилиты ssh-copy-id или вручную, путём прописывания ключа в конфиге.

## Добавляем SSH-ключ при помощи ssh-copy-id

ssh-copy-id — небольшая утилита, единственное назначение которой — прописывать ключ на удалённом сервере. Добавить ключ с её использованием можно так:

Скопировать кодBASH

ssh-copy-id -i <путь-до-ключа>/your-new-key.pub <имя\_пользователя>@<ip\_вашего сервера>

Здесь мы указываем путь до файла с ключом your-new-key.pub, имя пользователя и IP удалённого сервера.

Но у этой утилиты есть недостаток: чтобы добавить новый ключ к серверу, нужно подключиться с использованием уже добавленного на него ключа. Если это ваш первый ключ, следует воспользоваться вторым решением — добавить его вручную.

## Добавляем SSH-ключ вручную

Это может быть полезно в случаях, когда невозможно авторизоваться под пользователем, для которого добавляется ключ. Например, если вы только что добавили нового пользователя на сервере.

Перед началом операции скопируйте публичный ключ в буфер обмена, он нам ещё пригодится.

Чтобы добавить SSH-ключ на удалённый сервер, проделаем такие операции:

1. Подключимся к удалённому серверу при помощи SSH:

Скопировать кодBASH

ssh praktikum@<ip\_сервера>

1. Перейдём в домашнюю директорию пользователя, для которого необходимо добавить ключ. По умолчанию это /home/something, где something — имя учётной записи.

Если нужно добавить ключ текущему пользователю, можно использовать в пути символ ~ (тильда), который укажет на домашнюю директорию текущего пользователя. Перейдём в неё при помощи такой команды:

Скопировать кодBASH

cd ~

1. В случае, если мы добавляем первый ключ для данной учётной записи, нам потребуется создать соответствующие конфиги. Это можно сделать при помощи таких команд:

Скопировать кодBASH

*# создадим директорию для конфига ssh*

sudo mkdir -p -m 700 .ssh

*# создадим конфиг с ключами, для которых разрешено подключение*

sudo touch .ssh/authorized\_keys

*# проставим для него требуемые права доступа*

*# заменим something на имя пользователя, для которого добавляем ключ*

sudo chmod 664 .ssh/authorized\_keys

sudo chown -R something:something .ssh

Если для пользователя уже был задан SSH-ключ, этот этап можно пропустить и перейти к следующему.

1. Откроем конфигурацию SSH в редакторе при помощи команды:

Скопировать кодBASH

sudo nano ./.ssh/authorized\_keys



Интерфейс редактора nano

Вставим публичный ключ из буфера обмена и сохраним файл конфигурации нажатием комбинации Ctrl+O. Затем можно закрыть редактор, нажав Ctrl+X.

Если потребуется отозвать доступ для какого-то ключа, можно просто стереть соответствующую строчку из этого файла и снова его сохранить.

Всё готово — можно подключаться. Теперь разберёмся, каким образом работает аутентификация по ключу и как сервер нас узнаёт.

## **Как работает аутентификация по ключу**

Для управления сервером мы используем протокол SSH с аутентификацией по ключу. Этот механизм требует наличия пары ключей: приватного (privatekey) и публичного (publickey).

Приватный ключ сохраняется на стороне пользователя и не должен быть доступен кому-то ещё. Его утечка позволит злоумышленнику войти на сервер и выполнить действия от имени владельца ключа, поэтому при обращении с ним нужно быть очень осторожным.

Приватный ключ выглядит так:

Скопировать кодBASH

----BEGIN OPENSSH PRIVATE KEY-----

xxxx

-----END OPENSSH PRIVATE KEY-----

Он состоит из разделителей, которые помечают начало и конец ключа, и ключа шифрования между ними.

Публичный, или открытый ключ используется для шифрования сообщений, которые можно расшифровать только приватным ключом. Он генерируются автоматически при создании приватного.

Публичный ключ выглядит так:

Скопировать кодBASH

ssh-rsa xxxx denis@your-pc-name

Сначала указывается алгоритм шифрования, в нашем случае — RSA. Затем — сама ключевая последовательность. В конце записывается текстовый комментарий, который по умолчанию содержит имя пользователя и компьютера, на котором был сгенерирован этот ключ. Части в описании публичного ключа разделяются пробелом.

Запись приватного и публичного ключей выглядит по-разному, что помогает наглядно их различать. Так у разработчика меньше шансов ошибиться и скинуть не тот ключ.

Теперь, когда мы знаем какой ключ за что отвечает, разберёмся, как это работает на практике. Когда клиент пытается подключиться к серверу, последний отправляет клиенту сообщение, зашифрованное с помощью открытого ключа. Если клиент сможет расшифровать его и вернуть правильный ответ, то есть исходное сообщение, аутентификация пройдена.

Главное преимущество этого подхода — высокая безопасность. Приватный ключ не передаётся по сети и не покидает пределы компьютера пользователя, поэтому злоумышленник не может перехватить его в процессе подключения. А публичный ключ не позволяет подключиться к серверу, поэтому он бесполезен для злоумышленника.

Стойкость ключа, как и обычного пароля, зависит от длины и набора символов. Для SSH обычно используются ключи длиною в несколько тысяч символов (размер ключа в битах вы можете выбрать при его генерации), что делает его перебор «в лоб» практически бессмысленным.

## Добавляем ключ команды Практикума

Теперь, когда вы знаете, как управлять SSH-ключами, самое время закрепить это на практике — добавить ключ команды Практикума. Скопируйте ключ из блока ниже и добавьте его на сервер для вашего пользователя любым изученным способом:

Скопировать кодJSON

ssh-rsa  console@console

С помощью этого ключа вы предоставите код-ревьюеру и наставнику доступ к своей работе. Код-ревьюер проверит, что сервер настроен корректно, а наставник поможет, если возникнут трудности.

В следующем уроке настроим сервер. Поехали!

# Устанавливаем всё необходимое на сервер

В одном из предыдущих уроков вы подключились к удалённому серверу по SSH. Теперь необходимо установить на него всё, что нужно для работы API:

* Node.js, чтобы запустить приложение;
* MongoDB, чтобы работал сервер с базой данных;
* git, чтобы загружать код на сервер.

После этого мы уже к концу урока сможем запустить API, который создали на сервере.

## Установите Node.js

Сначала проверьте версию Node.js, которая установлена у вас локально. Для этого введите в командную строку команду:

Скопировать кодBASH

node -v

В командной строке появится номер версии — три числа, разделённые точками. Первое число — мажорная версия. Версия Node, которую мы установим на сервер, должна иметь такую же мажорную версию. Так инфраструктура нашего сервера будет близка к той, в которой мы разрабатывали приложение, а значит, оно будет работать без сюрпризов.

Теперь зайдите на сервер, который создали в предыдущем уроке и по очереди введите две следующие команды. В первой команде замените 14 на мажорную версию, которую узнали только что:

Скопировать кодBASH

*# в этой команде замените 14 на ту мажорную*

*# версию Node, которая стоит у вас локально*

curl -sL https://deb.nodesource.com/setup\_14.x | sudo -E bash -

*# установите Node.js*

sudo apt install -y nodejs

Первая команда добавляет официальный дистрибутив Node.js к нам в систему, а вторая его устанавливает. После того как вторая команда закончит установку, проверьте, что нужная версия Node установлена на сервере:

Скопировать кодBASH

node -v

## Установите MongoDB

Чтобы установить MongoDB на Ubuntu, воспользуйтесь [официальной инструкцией](https://docs.mongodb.com/manual/tutorial/install-mongodb-on-ubuntu/). Мы не будем объяснять, что значит каждая из этих команд, это выходит за рамки текущего курса.

Скопировать кодBASH

*# 1.*

curl -fsSL https://www.mongodb.org/static/pgp/server-4.4.asc | sudo apt-key add -

*# 2.*

echo "deb [ arch=amd64 ] https://repo.mongodb.org/apt/ubuntu bionic/mongodb-org/4.4 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-4.4.list

*# 3.*

sudo apt update

*# 4.*

sudo apt install -y mongodb-org

Когда последняя команда закончит работу, проверим, установилась ли MongoDB. Следующая команда запустит mongo-сервер:

Скопировать кодBASH

sudo service mongod start

И последнее. Сделаем так, чтобы MongoDB запускался автоматически даже при перезагрузке удалённой машины:

Скопировать кодBASH

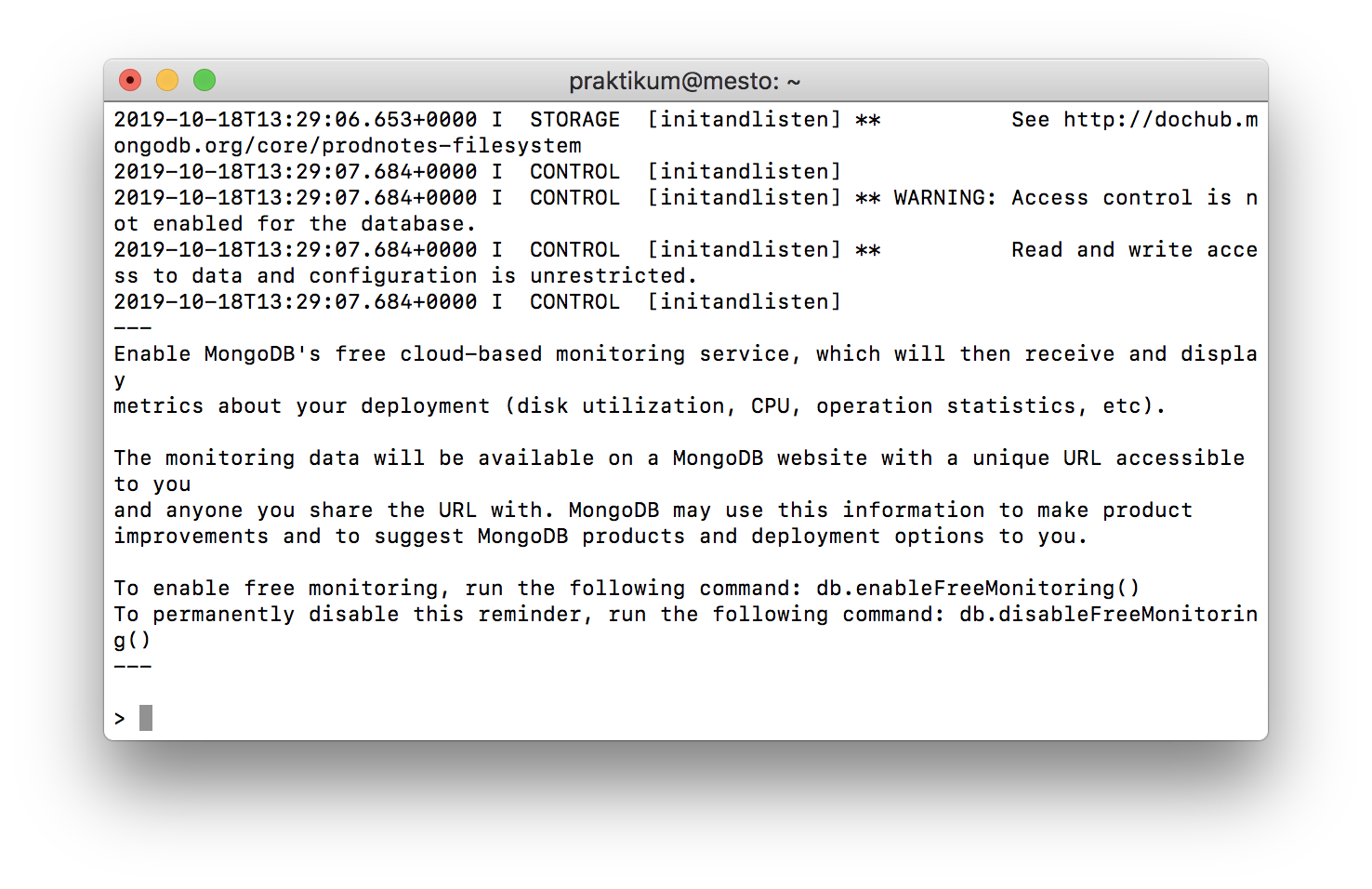
sudo systemctl enable mongod.service

Теперь у вас появилась возможность войти в mongo shell. Введите команду:

Скопировать кодBASH

mongo

Если всё в порядке, mongo shell запустится:



Чтобы выйти из него, нажмите Ctrl+C.

## Установите git

Осталось установить git, он позволит легко обновлять код на сервере. Для этого выполните следующую команду:

Скопировать кодBASH

sudo apt install -y git

После установки проверьте, всё ли в порядке, с помощью команды:

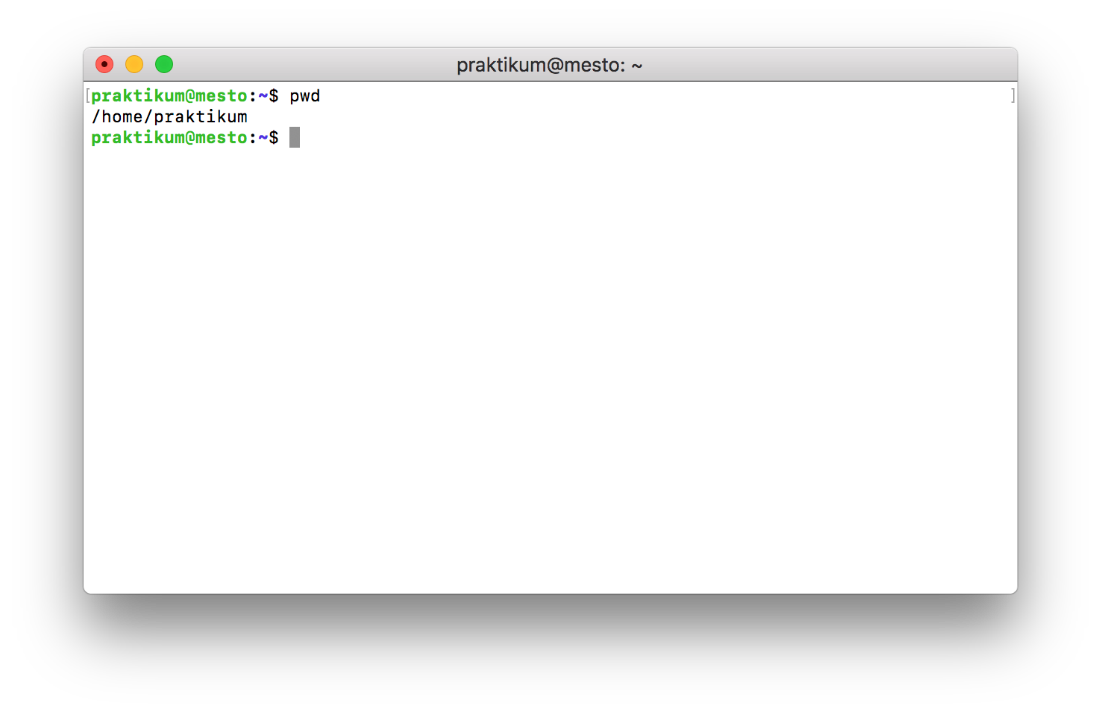
Скопировать кодBASH

git --version

Она должна вывести на экран установленную версию git.

## Запустите сервер

Перед тем как запустить сервер, нужно загрузить на него код бэкенд части приложения. Для этого склонируйте git репозиторий с кодом бэкенда приложения, которое разрабатывали в предыдущих спринтах, в папку вашего пользователя. В нашем случае это /home/praktikum:



Склонируйте репозиторий самостоятельно, вы уже умеете это делать. После этого войдите в папку проекта и установите зависимости:

Скопировать кодBASH

npm install

Затем запустите сервер:

Скопировать кодBASH

npm start

Сервер запущен, открывайте Postman.

## Проверьте, что всё работает

В Postman попробуйте сделать какой-нибудь запрос к удалённому серверу. Замените IP-адрес на свой и не забудьте добавить запросу 3000 порт:

Скопировать кодBASH

GET http://84.201.130.70:3000/users

Сервер отвечает ошибкой авторизации, но главное, что отвечает. Теперь попробуйте выйти из сервера: нажмите Ctrl+C и затем Ctrl+D.

Если вы отправите тот же запрос через Postman, то увидите, что ответа больше нет: когда вы вышли с сервера, вы остановили процесс, в котором работала Node.js. В следующем уроке разберёмся, как сделать так, чтобы при выходе приложение продолжало работать.

# Как сделать так, чтобы приложение работало всегда?

Node.js работает в определённом процессе на компьютере. Когда мы нажимаем Ctrl+C, этот процесс завершается, и приложение перестаёт работать. Чтобы этого не случалось, нужно воспользоваться менеджером процессов. Один из самых распространённых — pm2.

## Установка и запуск pm2

Зайдите на сервер и установите pm2 как глобальный модуль:

Скопировать кодBASH

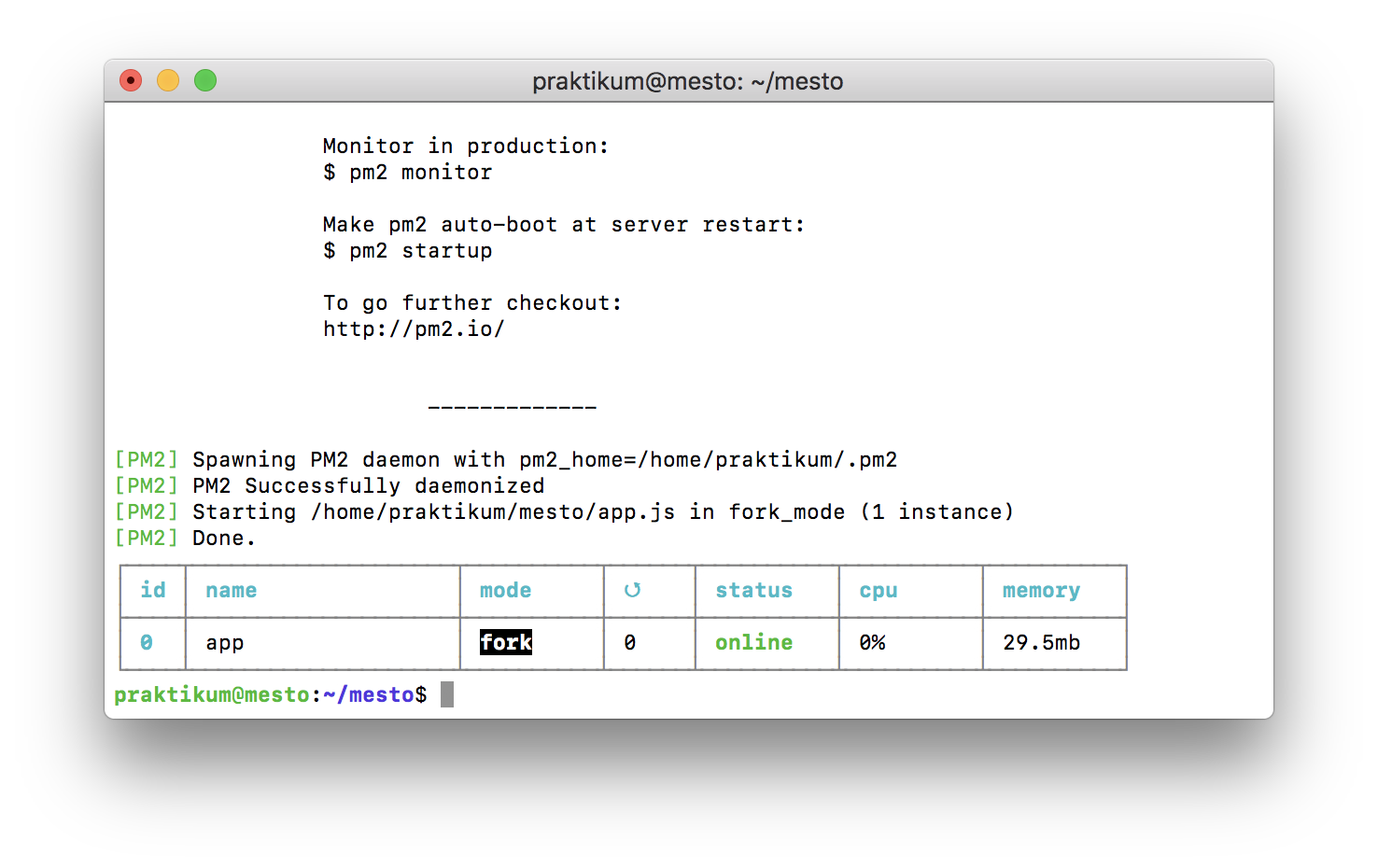
sudo npm install pm2 -g

Теперь можно запустить приложение с помощью pm2. Перейдите в папку проекта и используйте команду:

Скопировать кодBASH

pm2 start app.js

pm2 запустил приложение, по умолчанию взяв имя запускаемого файла без расширения:



Теперь, даже если мы разорвём соединение с сервером, приложение продолжит работать. А если наше приложение упадёт, pm2 его поднимет.

В случае, если с самим менеджером процессов что-то пошло не так, всегда можно воспользоваться следующими командами:

Скопировать кодBASH

pm2 kill *# Полностью отключает текущий PM2 и запущенные приложения*

pm2 start app *# Запускает приложение*

## Делаем приложение ещё стабильнее

Сделаем так, чтобы приложение запускалось даже при перезагрузке сервера. Для этого введите команду:

pm2 startup

В ответ вы получите примерно такое сообщение:

Скопировать кодBASH

To setup the Startup Script, copy/paste the following command:

sudo env PATH=$PATH:/usr/bin /usr/lib/node\_modules/pm2/bin/pm2 startup systemd -u praktikum --hp /home/praktikum

Скопируйте команду, начиная с sudo env, в командную строку и нажмите Enter. Последнее, что нужно сделать, — ввести команду:

Скопировать кодBASH

pm2 save

Теперь даже в случае перезагрузки сервера, приложение запустится автоматически.

## Ссылки

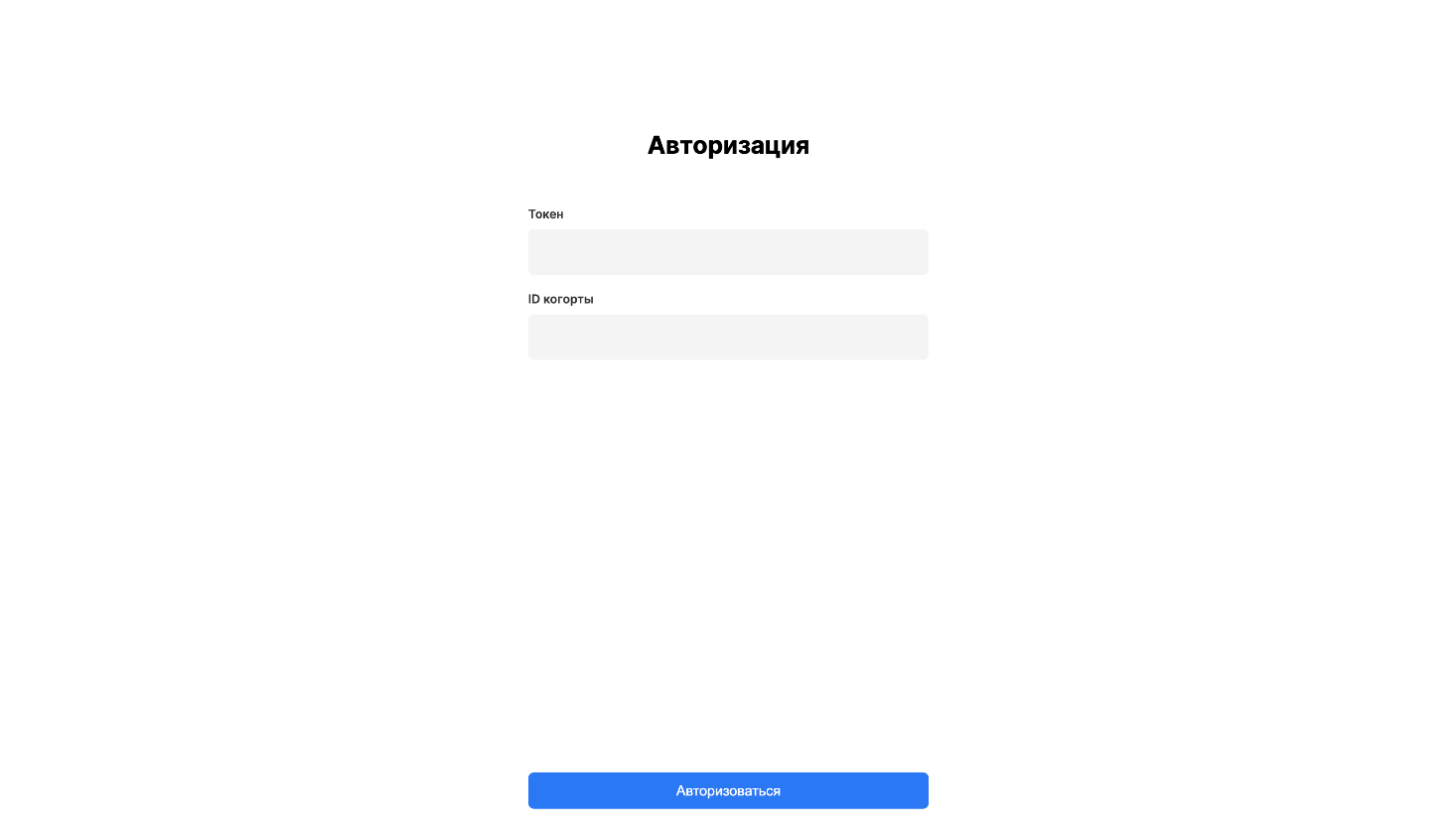
Документация модуля pm2: <http://pm2.keymetrics.io/docs/usage/pm2-doc-single-page/>.

Pm2 на npm: <https://www.npmjs.com/package/pm2>.

# Прикрепляем доменное имя

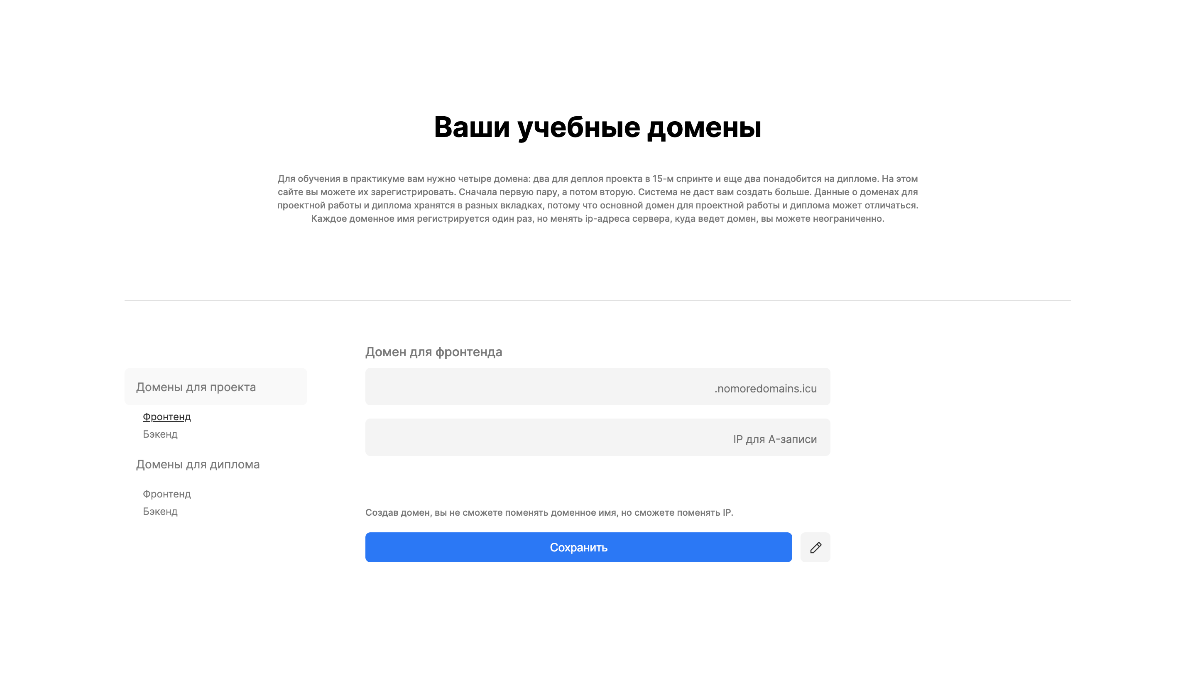
Пока мы можем обратиться к серверу только по IP-адресу, но в реальном мире обычно используется доменное имя. В этом уроке вы создадите и прикрепите его к IP-адресу сервера.

Обычно создание домена — платная услуга. А пока учитесь — [воспользуйтесь нашим сервисом по ссылке бесплатно](https://domain.nomoreparties.site/). Для начала потребуется авторизоваться по вашему токену и номеру когорты:



Интерфейс авторизации перед созданием доменов

После успешной авторизации перед вами откроется интерфейс с вашими доменами:



Интерфейс со списком доменов

Сперва нужно указать доменное имя, которое хотели бы получить. Будем регистрировать домен для фронтенда, поэтому выберите вкладку «Фронтенд» в «Домены для проекта». Введите желаемое имя в первое поле ввода. Домен может иметь такой вид:

Скопировать код

domainname.students

another.domainname.students

Домен .nomoredomains.icu со скриншота выше будет добавлен к имени домена автоматически. Другими словами, если вы введёте в поле domainname.students , будет создан такой домен:

Скопировать кодBASH

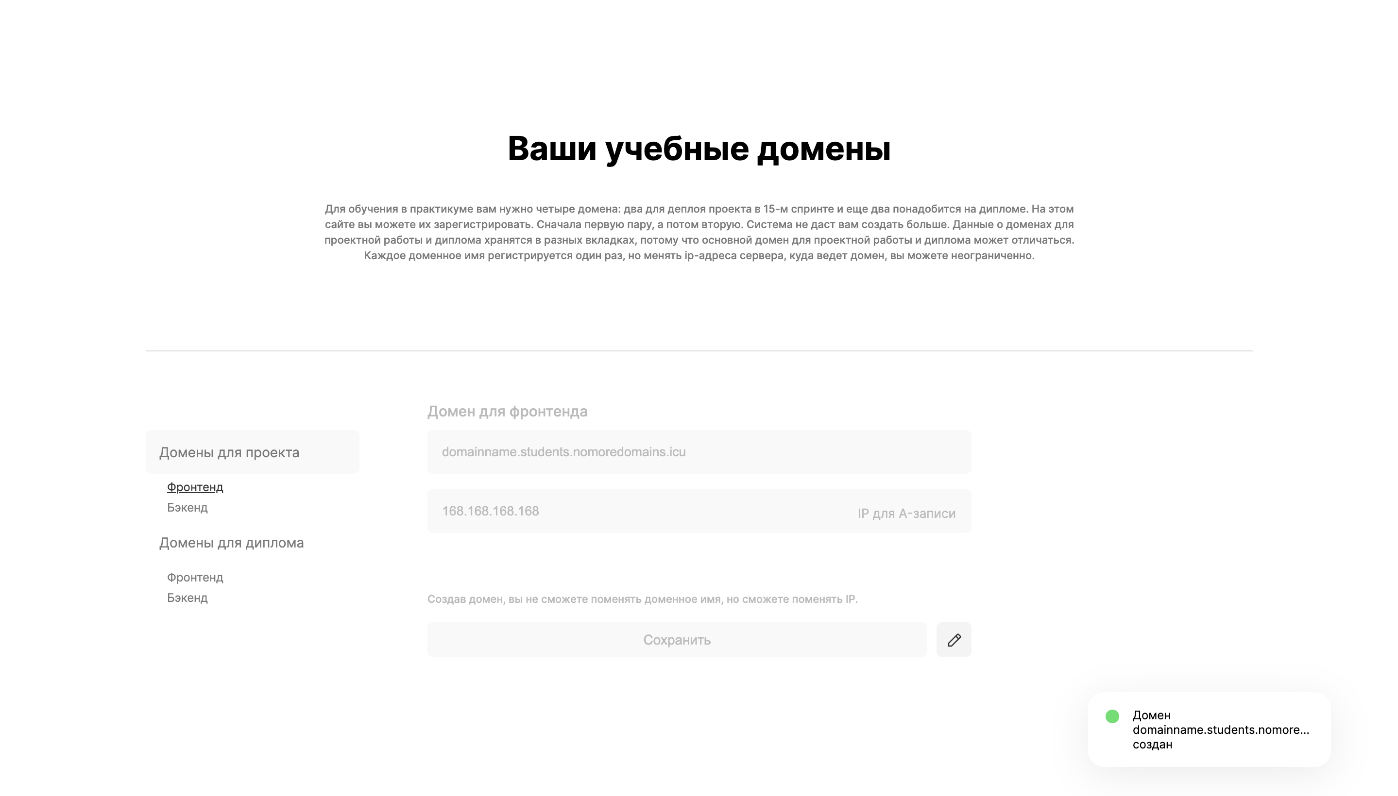
domainname.students.nomoredomains.icu

Следующим шагом нужно создать привязку домена и вашего сервера в Яндекс.Облаке. Это можно сделать с помощью A-записи. Она связывает домен и сервер по публичному IP-адресу. Введите публичный IP-адрес сервера в Яндекс.Облаке во второе поле. Используйте такой формат записи:

Скопировать код

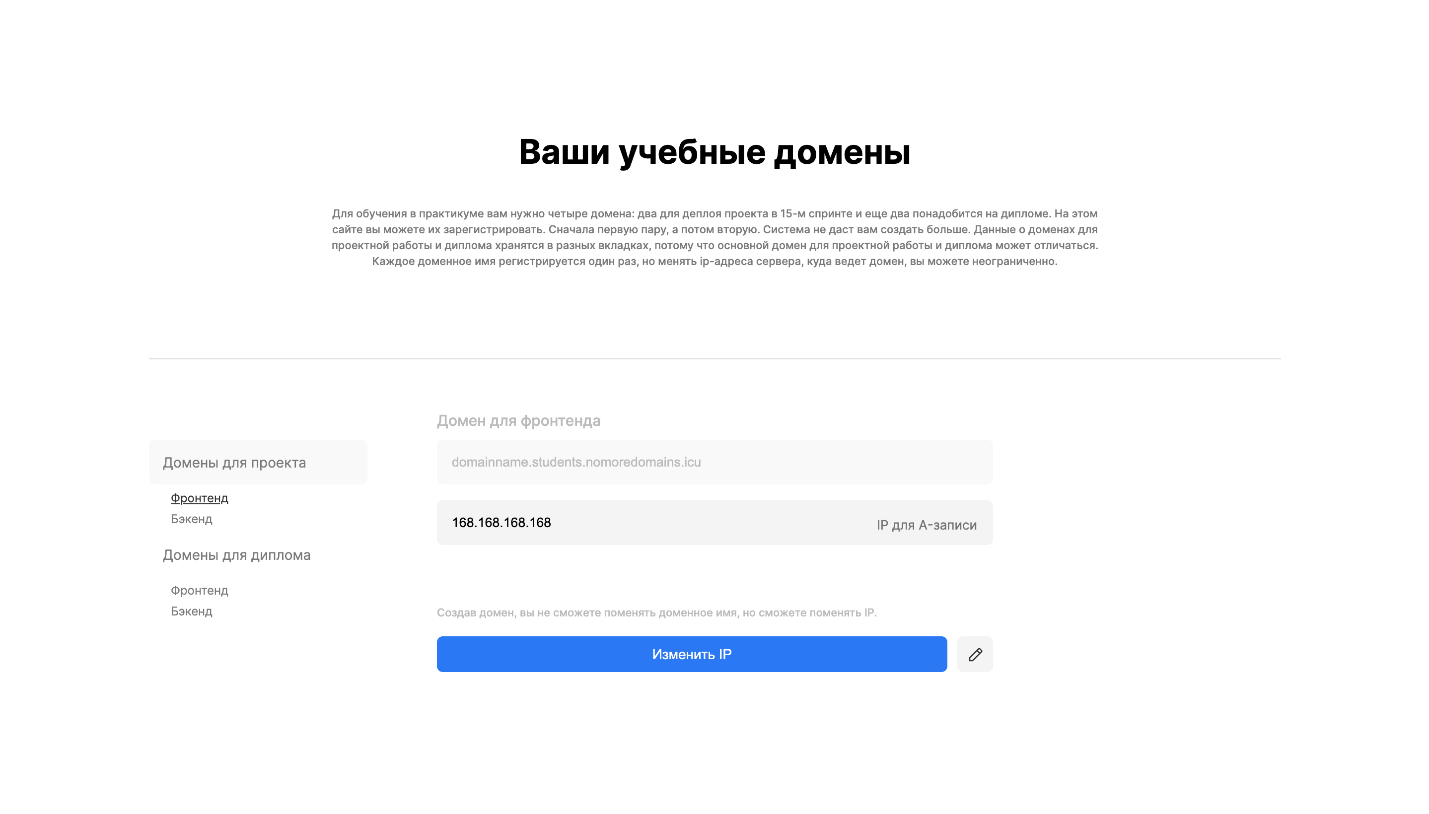
000.000.000.000

Как только все данные будут введены, нажмите «Сохранить». Если запрос на создание домена прошёл успешно, появится соответствующее сообщение:



Успешное создание домена

В дальнейшем, если публичный IP-адрес сервера в Яндекс.Облаке изменится — вы сможете его отредактировать нажатием на иконку редактирования. Доменное имя изменить нельзя.



Интерфейс редактирования IP-адреса домена

Теперь DNS службе нужно время, чтобы обновить данные о домене. Обычно это занимает до 10 минут. Поэтому через 10–15 минут зайдите в Postman и попробуйте отправить запрос:

Скопировать кодBASH

POST http://domainname.students.nomoreparties.co:3000/signup

Если домен прикрепился, вы получите ответ вашего сервера, если нет, нужно подождать ещё.

Обратите внимание, что мы по-прежнему указываем 3000 порт, так как именно на нём запущен наш Node сервер. Если убрать порт из запроса, ответа не будет. В следующем уроке разберёмся, почему так происходит и как это исправить.

# Разбираемся с портами. HTTP-сервер nginx

В предыдущем уроке мы столкнулись с проблемой: если отправить на сервер запрос без указания порта, сервер не ответит. Это происходит, потому что наше приложение запущено на 3000 порту, а запрос без указания порта по умолчанию идёт на 80-й.

Мы могли бы попробовать это исправить, изменив порт в коде приложения:

Скопировать кодBASH

app.listen(80);

Если сделать это и запустить приложение на сервере в облаке, будет ошибка:

Скопировать кодBASH

Error: listen EACCES 0.0.0.0:80

Дело в том, что Node.js по умолчанию не может взаимодействовать с портами ниже 1024. И это даже не ограничение Node.js, а ограничение Unix-систем, которое не разрешает процессу, запущенному без прав суперпользователя, взаимодействовать с такими портами. Получается, что ошибку можно исправить, если запустить приложение с правами суперпользователя. Для этого нужно добавить команду sudo:

Скопировать кодBASH

sudo node app.js

Тогда Node.js приложение сможет работать на 80 порту, но при этом станет гораздо менее безопасным. Если злоумышленник каким-то образом сумеет запустить код на вашем сервере, то у него будет полный доступ к машине. А если Node.js приложение будет работать под обычным пользователем и злоумышленник как-то запустит в нём свой код, у него будет много системных ограничений. Он уже не сможет делать с машиной всё, что пожелает. Поэтому добавление sudo — не наш вариант.

Нам нужно научиться как-то перенаправить запрос с 80 порта на 3000, где работает Node.js сервер. Это можно сделать с помощью nginx.

## Nginx

Nginx — это HTTP-сервер. Он умеет раздавать статические файлы, перенаправлять запросы, кешировать результат, и делает всё это очень быстро. Но на nginx нельзя создать сервер с какой-то сложной логикой, это задача Node.js. Поэтому они и работают в связке:

* если нужно отдать статический файл или перенаправить запрос куда-то, лучше поручить это nginx;
* если нужно исполнить какую-то логику — это сделает Node.js;

Мы будем использовать nginx как обратный прокси-сервер. Обратный прокси-сервер принимает запросы извне и перенаправляет их во внутреннюю сеть. То есть, получив ответ из внутренней сети, nginx направит его клиенту.

Наша задача сделать так, чтобы nginx слушал запросы на 80 порту и перенаправлял их на 3000 порт, где работает Node.js.

Зайдите на удалённый сервер и установите nginx:

Скопировать кодBASH

sudo apt update *# обновляем список пакетов (программ), доступных для установки*

sudo apt install -y nginx *# устанавливаем nginx*

После установки нужно настроить файрвол — прослойку между сервером и внешним миром, которая решает, какие запросы пускать дальше, а какие отклонять.

В операционной системе Ubuntu по умолчанию установлен файрвол — ufw. Сконфигурируем его, разрешив некоторые виды трафика. Для этого используйте команды:

Скопировать кодBASH

sudo ufw allow 'Nginx Full' OpenSSH либо sudo ufw allow 'Nginx Full'

sudo ufw allow OpenSSH

sudo ufw allow 'Nginx Full' открывает на текущей машине два порта: 80 и 443, на них приходят http и https запросы к серверу. OpenSSH открывает порт 22 — соединение по ssh, чтобы мы могли подключаться к серверу через консоль и при работающем файрволе.

После этого нужно включить файрвол:

Скопировать кодBASH

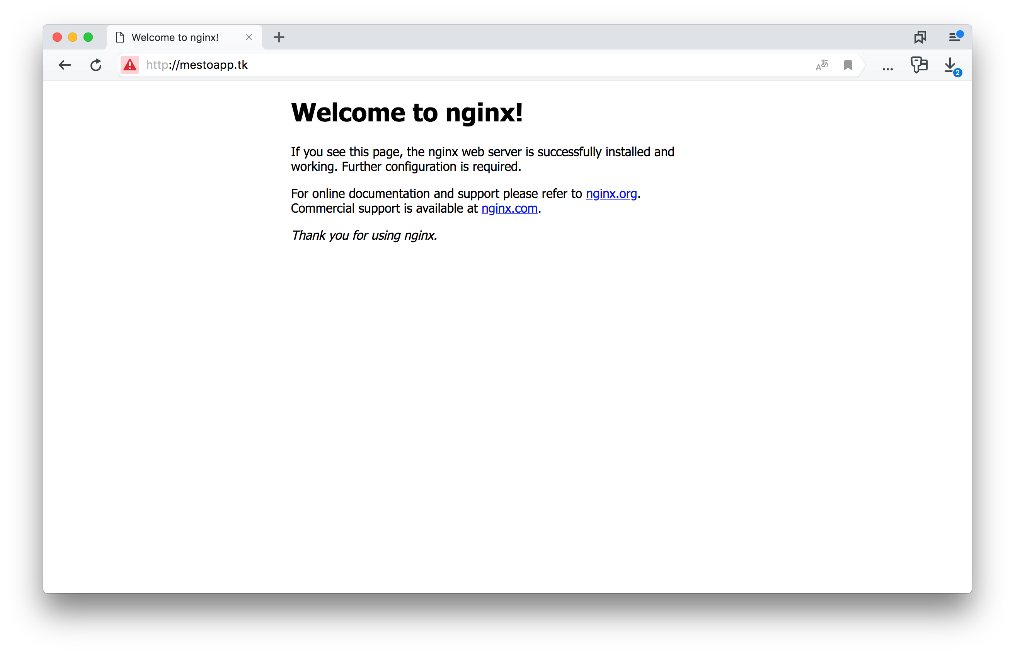
sudo ufw enable

Осталось запустить nginx и добавить его в автозапуск. Это можно сделать за один шаг при помощи следущей команды :

Скопировать кодBASH

sudo systemctl enable --now nginx

Откройте в браузере адрес вашего сервера. Вы должны увидеть:



Это значит, что nginx запустился и работает на 80 порту. Но сейчас он просто отдаёт стандартную html-страницу, а мы хотим, чтобы запросы перенаправлялись. Для этого нужно изменить конфигурационный файл nginx.

В nginx есть две директории для хранения конфигураций: sites-available/ и sites-enabled/. В первой хранятся конфигурации всех хостов, а во второй — ссылки только на активные. Перед редактированием конфигурации нужно создать символическую ссылку между sites-available и sites-enabled:

Скопировать кодBASH

sudo ln -s /etc/nginx/sites-available/<имя\_конфига> /etc/nginx/sites-enabled/<имя\_конфига>

Для конфигурации по умолчанию (default) символическая ссылка уже создана, так что этот этап можно пропустить.

Теперь переходим к редактированию файла конфигурации. В командной строке введите:

Скопировать кодBASH

sudo nano /etc/nginx/sites-available/default

*# sudo значит запустить команду как суперпользователь*

*# nano — это текстовый редактор*

*# /etc/nginx/sites-available/default — путь к конфигурационному файлу nginx*

Удалите содержимое этого файла и вставьте туда такую конфигурацию:

Скопировать кодBASH

server {

listen 80;

server\_name mestoapp.gq www.mestoapp.gq;

location / {

proxy\_pass http://localhost:3000;

proxy\_http\_version 1.1;

proxy\_set\_header Upgrade $http\_upgrade;

proxy\_set\_header Connection 'upgrade';

proxy\_set\_header Host $host;

proxy\_cache\_bypass $http\_upgrade;

}

}

Конфигурация говорит, что слушать запросы необходимо на 80 порту и перенаправлять их на http://localhost:3000, то есть на 3000 порт внутренней сети. Кроме перенаправления запроса конфигурация выставляет некоторые дополнительные заголовки. Также в конфигурационном файле есть блок server\_name. В нём пропишите доменное имя, зарегистрированное в предыдущем уроке. Тогда такая конфигурация будет перенаправлять все запросы с http://mestoapp.gq нашему Node.js приложению на 3000 порт.

Запоминать эти команды необязательно, их всегда можно подсмотреть в документации.

После того как всё сделаете, нажмите Ctrl+X. Nano спросит у вас, хотите ли вы сохранить изменения, нажимайте y и Enter.

После обновления конфигурации nginx нужно перезапустить. Но если в конфигурации окажутся ошибки, то в лучшем случае работа приложения будет остановлена. Nginx позволяет изменять и тестировать конфигурацию «на лету». Для этого до перезапуска nginx введите команду:

Скопировать кодBASH

sudo nginx -t

Она позволит проверить конфигурацию на наличие ошибок в синтаксисе команд. Если всё прошло успешно, вы получите такой ответ:

Скопировать кодBASH

nginx: the configuration file /etc/nginx/nginx.conf syntax is ok

nginx: configuration file /etc/nginx/nginx.conf test is successful

Уже после этого можно перезагрузить конфигурацию nginx:

Скопировать кодBASH

sudo systemctl reload nginx

Теперь попробуйте послать из Postman запрос без указания порта:

Скопировать кодBASH

POST http://mestoapp.gq

Пришел ответ от Node.js сервера — мы настроили проксирование запросов через nginx!

# Заливаем обновления на сервер

Мы сделали почти всё: настроили сервер, развернули бэкенд, подключили сертификаты. Но разработка проекта на этом не заканчивается. Возможно, в нём найдётся баг, который надо будет исправить, или мы решим внедрить новую функциональность, тогда код на сервере тоже придётся обновить.

Чтобы работать с git потребуется пара SSH-ключей. [В этом уроке](https://praktikum.yandex.ru/learn/web/courses/35d951a1-b62c-4a96-96ac-a8118657fad0/sprints/1702/topics/2ba3e1f9-e5e6-4dea-a218-8284951be940/lessons/547966f0-d2df-4f29-8408-4f24fdb29918/) мы рассказывали как их сгенерировать.

[В пятом уроке](https://praktikum.yandex.ru/trainer/web/lesson/fd9247e3-9de2-40da-a0a1-dac7fc72b0dc) мы установили на сервер git. Все обновления кода мы загружаем на github, в ветку main. Чтобы обновить код, зайдите в папку с проектом на сервере и заберите последние изменения с github:

Скопировать кодBASH

git pull origin main

После этого останется только перезагрузить pm2, чтобы изменения вступили в силу:

Скопировать кодBASH

pm2 restart app

Процесс перезапустится, и обновления вступят в силу.

### Из какой ветки деплоить код на ревью

Main — основная ветка проекта. В ней содержится стабильная версия кода, пригодная для деплоя. Но при отправке проектной работы на ревью, вы пишете код в отдельной ветке. Поэтому в проектной работе можно задеплоить код не из мастера, а из ветки, в которой ведёте работу.

# Кросс-доменные запросы и механизм CORS

При создании современного фронтенда разработчик регулярно сталкивается с необходимостью обмена данными с различными внешними системами. [Из урока про CORS](https://praktikum.yandex.ru/trainer/web/lesson/8f98665e-1efd-4a6c-baed-10f7ec7891b7) вы знаете, что браузер по умолчанию блокирует отправку кросс-доменных запросов.

Это сделано, чтобы обеспечить безопасность пользователя. Если бы такое ограничение отсутствовало, злоумышленник мог бы создать страницу, код которой незаметно загружал бы сайт вашего онлайн-банка и осуществлял с ним операции. При открытии такой страницы браузер автоматически подставлял бы авторизационные данные в виде cookies и отправлял запрос на сервер банка от имени пользователя. Один клик по вредоносной ссылке — и вот вы уже обыскиваете морозилку в надежде найти затерявшийся пельмешек на ужин.

Но хоть запрет кросс-доменных запросов и обеспечивает безопасность, он также доставляет и неудобства. Например, вы разрабатываете публичный API сервис, который возвращает пользователю информацию о курсах валют. Сервер API и сайты клиентов расположены на разных доменах, поэтому браузер по умолчанию заблокирует запрос к API как кросс-доменный, и воспользоваться этим сервисом с других сайтов будет невозможно.

Другой вариант — обращение к собственному API, который расположен на другом поддомене или домене. Например, сайт mestoproject.ru не сможет получить данные от API на api.mestoproject.ru, поскольку запрос к нему будет заблокирован браузером как кросс-доменный.

Разработчики — люди дальновидные, поэтому при создании ограничения они сразу придумывают способ его обойти или смягчить. В нашем случае этот способ — механизм CORS.

## Механизм CORS

CORS (Cross-Origin Resource Sharing, совместное использование ресурсов) — механизм, который позволяет серверу-обработчику управлять ограничениями на кросс-доменные запросы для определённых ресурсов и методов.

На практике это выглядит так: перед отправкой кросс-доменного запроса браузер обращается к серверу и с помощью механизма CORS уточняет, можно ли его выполнить. Для этого браузер и сервер обмениваются специальными служебными заголовками: браузер прикладывает заголовки к запросу, а сервер — к ответу.

CORS-запросы бывают двух типов: простые и сложные. Простые используют только исходный запрос для обмена информацией, а сложные дополнительно требуют отправки ещё одного запроса. То, какой подход будет использован, зависит от метода исходного запроса и «безопасности» его заголовков.

CORS-безопасными считаются такие пользовательские заголовки: Accept , Accept-Language , Content-Language , Content-Type , Origin. Для заголовка Content-Type проверяется ещё и значение: безопасными считаются только application/x-www-form-urlencoded, multipart/form-data, text/plain. Любое другое значение этого заголовка нарушает требования безопасности, а значит требует обработки при помощи сложных CORS-запросов

Другие же дополнительные пользовательские заголовки — небезопасны. Их наличие требует использования механизма сложного CORS-запроса.

Теперь разберёмся, как обработка CORS-запросов происходит на практике.

## Простые CORS-запросы

Простые CORS-запросы не требуют отправки дополнительных запросов. Такой тип запросов используется для POST-, GET-, HEAD-запросов, когда в них содержатся только CORS-безопасные пользовательские заголовки.

Для обмена информацией в CORS используют такие заголовки:

* Origin — содержит источник запроса (домен сайта, с которого осуществлён запрос). Браузер формирует его автоматически и прикладывает к каждому запросу, благодаря чему сервер знает, откуда пришёл запрос.
* Access-Control-Allow-Origin — используется сервером и прикладывается к ответу. Содержит список источников, с которых можно осуществлять кросс-доменные запросы к конкретному ресурсу. В случае, если источников несколько, они перечисляются через запятую. А если доступ разрешён с любого сайта, используют специальное значение \*.

Эти два заголовка обязательны и используются как в простых, так и в сложных CORS-запросах.

Браузер считает простой кросс-доменный запрос разрешённым, если выполняются такие условия:

1. Ответ сервера содержит заголовок Access-Control-Allow-Origin.
2. В списке разрешённых источников в заголовке Access-Control-Allow-Origin содержится адрес сайта, с которого был отправлен запрос.

Если хотя бы одно условие нарушается, браузер помечает запрос как «заблокированный» и не возвращает его результат приложению пользователя.

Рассмотрим процесс разрешения простого CORS-запроса на примере GET-запроса к api.website.com/users, отправленного с сайта mywebsite.com:

Скопировать кодBASH

GET /users HTTP/1.1

Origin: https://www.mywebsite.com

Host: api.website.com

В этом случае сервер вернёт такой ответ:

Скопировать кодBASH

HTTP/1.1 200 OK

Access-Control-Allow-Origin: https://www.mywebsite.com

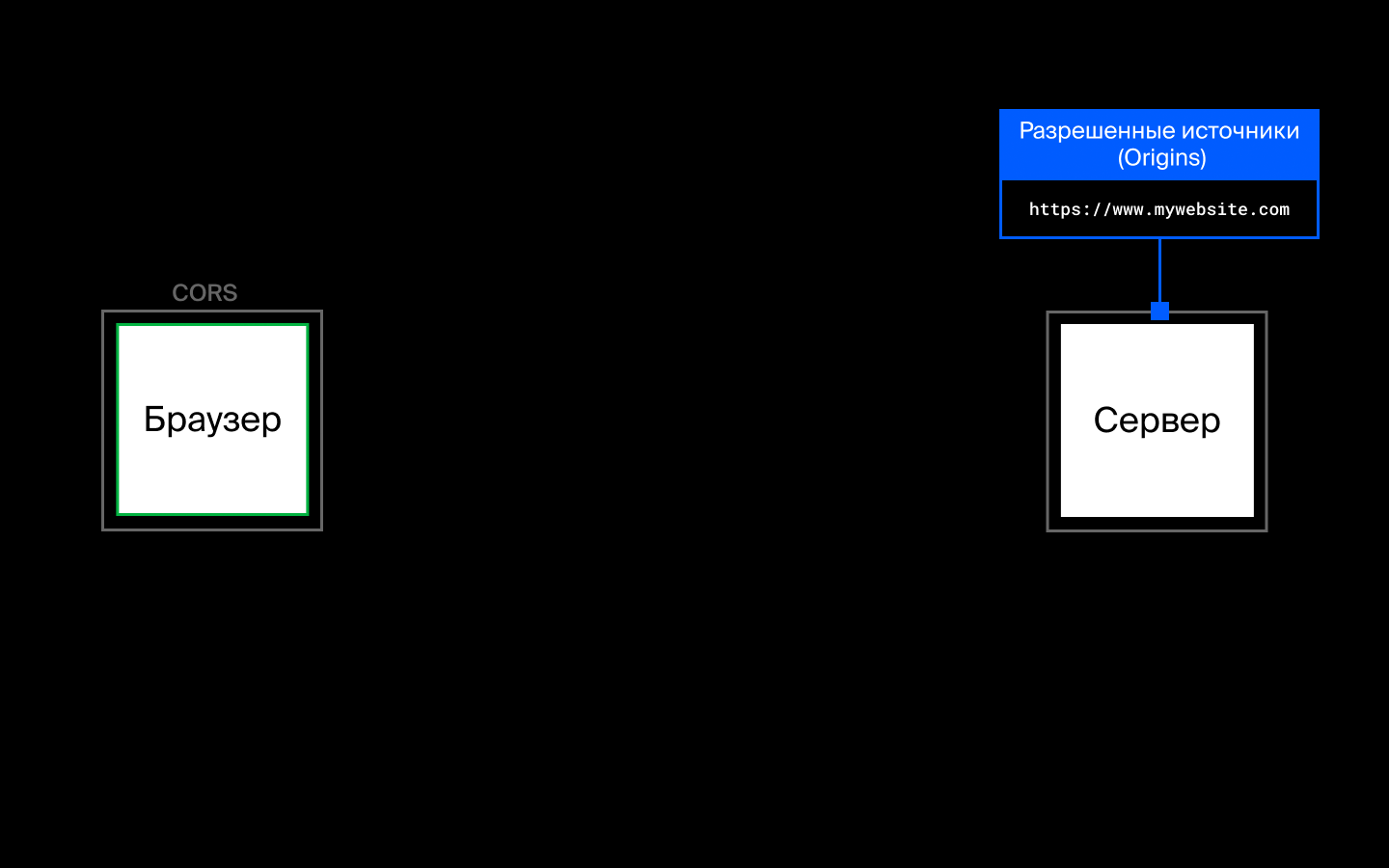
Content-Type: application/json

Server: Apache

{ users: [{...}, {...}] }

Запрос будет разрешён, ведь в Access-Control-Allow-Origin содержится источник (Origin), с которого был осуществлён запрос.

А так можно визуально представить процесс обработки простого CORS-запроса:



Механизм обработки простого CORS-запроса

Теперь рассмотрим, как формируются сложные CORS-запросы.

## Сложные CORS-запросы

Сложные CORS запросы требуют отправки предварительного (англ. preflight) запроса перед исходным. Предварительный запрос помогает браузеру узнать, можно ли осуществить запрос до его обработки. Это позволяет избежать лишних действий сервера, если запрос оказывается запрещён.

Предварительный запрос — обычный HTTP-запрос с методом OPTIONS и URL, который соответствует URL исходного запроса, и некоторыми дополнительными заголовками.

В дополнение к заголовкам из простых CORS-запросов (Origin и Access-Control-Allow-Origin) в предварительных запросах браузер прикладывает такие:

* Access-Control-Request-Method — указывает метод (тип) исходного запроса.
* Access-Control-Request-Headers — содержит список «небезопасных» заголовков основного запроса, перечисленных через запятую.

А сервер же отвечает с использованием таких заголовков:

* Access-Control-Allow-Method — возвращает методы (типы) запроса, разрешённые для данного URL.
* Access-Control-Allow-Headers — возвращает список «небезопасных» заголовков, разрешённых для данного кросс-доменного запроса. Если их несколько, они перечисляются через запятую.

Обратите внимание, что заголовки запросов и ответов в некотором роде зеркальные. Браузер использует заголовки с ключевым словом Request, когда запрашивает разрешение, а сервер — с ключевым словом Allow, когда разрешает запросы с определёнными методами и заголовками.

Если предварительный запрос оказывается разрешён, следом происходит отправка исходного запроса и обработка его результата. Таким образом, разрешение кросс-доменного доступа и обработка запроса разделяются на два этапа.

Сложные CORS-запросы используются для PUT-, PATCH- и DELETE-запросов, а также любых запросов, которые содержат «небезопасные» CORS-заголовки.

Рассмотрим механизм формирования предварительного запроса на примере PUT запроса к API [api.website.com/users/1](http://api.website.com/users/1), отправленного с сайта mywebsite.com:

Пользов

Скопировать кодJSON

PUT /users/1 HTTP/1.1

Origin: https://www.mywebsite.com

Host: api.website.com

Content-Type: application/json

Authorization: Bearer ADwHhZmGU9aklyrVpVp30mTns

{"name": "Denis", "about": "Developer"}

По этому запросу браузер построит следующий предварительный запрос:

Скопировать кодJSON

OPTIONS /users/1 HTTP/1.1

Origin: https://www.mywebsite.com

Host: api.website.com

Access-Control-Request-Method: PUT

Access-Control-Request-Headers: Content-Type, Authorization

Access-Control-Request-Method содержит тип исходного запроса — PUT, а Access-Control-Request-Headers — список заголовков исходного запроса, которые не являются CORS-безопасными, то есть Content-Type и Authorization. В этом случае заголовок Content-Type считается небезопасным, поскольку значение application/json не входит в список безопасных: application/x-www-form-urlencoded, multipart/form-data и text/plain.

Сервер получает предварительный запрос и отправляет браузеру пустой ответ, который содержит CORS-заголовки. Решение о том, допустим ли сложный кросс-доменный запрос, браузер принимает после получения ответа на предварительный запрос и до отправки исходного запроса. Браузер считает запрос разрешённым, если выполняются такие условия:

1. В заголовке Access-Controll-Allow-Methods содержится тип исходного запроса.
2. Список дополнительных заголовков исходного запроса, переданных в заголовке Access-Control-Request-Headers, соответствует списку разрешённых, которые указаны в заголовке Access-Control-Allow-Headers ответа.

Если предварительный запрос разрешён, браузер отправляет исходный.

На практике обработка сложного CORS-запроса происходит так:

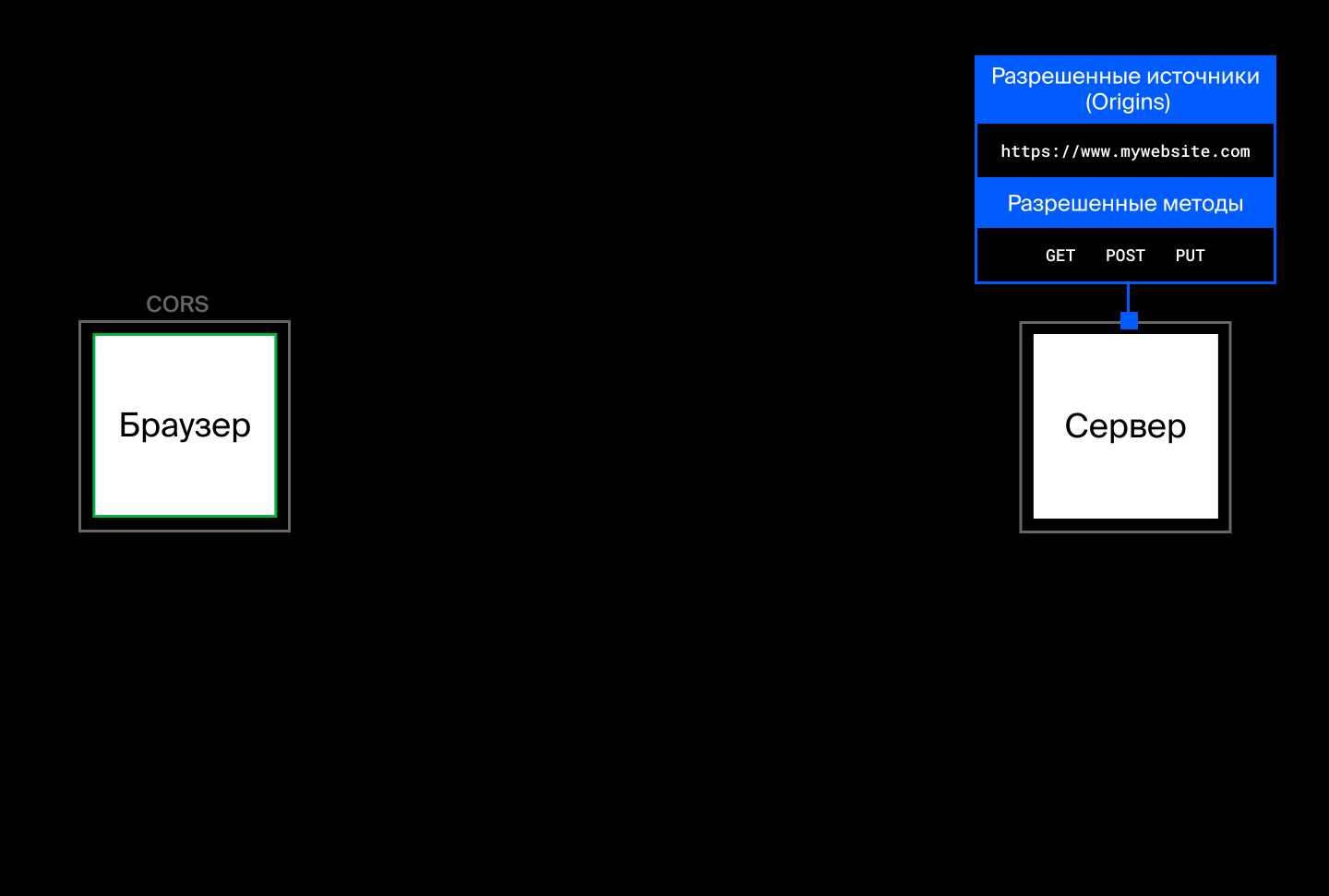


Схема взаимодействия клиента и сервера для «сложного» CORS-запроса

## Добавляем поддержку CORS в бэкенд-сервис

Теперь, когда мы разобрались с механизмом работы CORS, добавим его поддержку в наш бэкенд-сервис. Для этого нужно разобраться в паре вещей:

* как разрешать запросы из определённых источников;
* как корректно обрабатывать предварительные CORS-запросы.

Для начала следует определить список источников, с которых мы хотим разрешить кросс-доменный доступ к нашему API. В него надо включить домены сайтов, которым нужен доступ к данным нашего сервера.

Обработка CORS — типовая задача, поэтому лучше выносить реализацию во внешнюю мидлвару. Это позволит избежать дублирования кода.

### Разрешаем доступ с определённых источников

Источник запроса содержится в заголовке Origin, и чтобы его узнать, нужно обратиться к «хранилищу заголовков» запроса (req.headers) по ключу, равному имени заголовка — origin.

Затем необходимо проверить, найден ли источник в списке разрешённых:

Скопировать кодJAVASCRIPT

*// Массив доменов, с которых разрешены кросс-доменные запросы*

const allowedCors = [

'https://praktikum.tk',

'http://praktikum.tk',

'localhost:3000'

];

app.use(function(req, res, next) {

const { origin } = req.headers; *// Сохраняем источник запроса в переменную origin*

*// проверяем, что источник запроса есть среди разрешённых*

if (allowedCors.includes(origin)) {

...

}

next();

});

Если источник запроса есть среди разрешённых, прописываем его в заголовок ответа Access-Control-Allow-Origin. Это можно сделать вызовом метода header объекта ответа. Для этого передайте в метод два параметра: имя заголовка и его значение.

Скопировать кодJAVASCRIPT

...

if (allowedCors.includes(origin)) {

*// устанавливаем заголовок, который разрешает браузеру запросы с этого источника*

res.header('Access-Control-Allow-Origin', origin);

}

...

Если источник запроса не найден среди разрешённых, пропускаем обработку запроса дальше без изменений заголовка ответа. Браузер сам заблокирует запрос, поскольку заголовок не был передан.

В случае, если нам необходимо разрешить доступ для любого источника, задаём заголовку Access-Control-Allow-Origin значение \* вне зависимости от источника запроса:

Скопировать кодJAVASCRIPT

...

*// устанавливаем заголовок, который разрешает браузеру запросы из любого источника*

res.header('Access-Control-Allow-Origin', "\*");

...

Отлично! Теперь наш сервис научился обрабатывать простые CORS-запросы. Но это не всё. Если ему придёт сложный запрос, он не сможет его обработать. Чтобы это исправить, потребуется добавить поддержку предварительных запросов.

### Обрабатываем предварительные запросы

Для передачи предварительных запросов браузер использует HTTP-метод OPTIONS. Эта особенность позволяет разделять логику обработки исходных и предварительных запросов. Чтобы определить метод HTTP-запроса, нужно обратиться к полю method объекта запроса (req.method). Если метод равен OPTIONS, это предварительный CORS-запрос.

Чтобы сервер мог корректно разрешать предварительные запросы, он должен включать в ответ дополнительные заголовки: Access-Control-Allow-Methods и Access-Control-Allow-Headers.

Заголовок Access-Control-Allow-Methods содержит список HTTP-методов, разрешённых для конкретного ресурса:

Скопировать кодJAVASCRIPT

...

const { method } = req; *// Сохраняем тип запроса (HTTP-метод) в соответствующую переменную*

*// Значение для заголовка Access-Control-Allow-Methods по умолчанию (разрешены все типы запросов)*

const DEFAULT\_ALLOWED\_METHODS = "GET,HEAD,PUT,PATCH,POST,DELETE";

*// Если это предварительный запрос, добавляем нужные заголовки*

if (method === 'OPTIONS') {

*// разрешаем кросс-доменные запросы любых типов (по умолчанию)*

res.header('Access-Control-Allow-Methods', DEFAULT\_ALLOWED\_METHODS);

}

...

В большинстве случаев его значение состоит из основных типов запросов: GET, HEAD, PUT, PATCH, POST, DELETE.

Access-Control-Allow-Headers содержит перечень заголовков, которые разрешены к передаче при кросс-доменных запросах. Обычно в качестве его значения используют значение заголовка Access-Control-Request-Headers, переданного браузером вместе с запросом:

Скопировать кодJAVASCRIPT

...

*// сохраняем список заголовков исходного запроса*

const requestHeaders = req.headers['access-control-request-headers'];

if (method === 'OPTIONS') {

...

*// разрешаем кросс-доменные запросы с этими заголовками*

res.header('Access-Control-Allow-Headers', requestHeaders);

}

...

Теперь точно всё — ваш сервер полностью умеет работать с CORS.

## Заключение

CORS — один из базовых механизмов безопасности браузера. С ним выигрывают все: разработчик может обмениваться информацией с внешними сервисами, а пользователи — не переживать за свои данные.

Благодаря CORS мы можем гибко управлять нашим сервисом: разрешать обращаться к нему с одних ресурсов и запрещать делать это с других. А это жизненно необходимо при использовании сервиса в продакшене.

# Фронтенд и бэкенд на одном домене

Вы научились деплоить бэкенд на сервере и прикреплять к нему домен. Теперь, когда необходимо послать запрос к API, можно использовать доменное имя.

Вы уже умеете деплоить фронтенд на Github Pages, но фронтенд реальных проектов обычно хранят не на Github, а на каком-нибудь сервере с красивым доменным именем. Нужно научиться делать так, чтобы при заходе на сайт с использованием доменного имени пользователь видел фронтенд сайта, и при этом сохранялась возможность делать запросы к API. Приступим.

## Заливаем код фронтенда на сервер

Залить код фронтенда можно так же, как мы делали это с бэкендом — через клонирование git-репозитория. После того как репозиторий клонирован, нужно запустить на сервере процесс сборки, и когда он закончится, в папке build будут файлы, готовые к раздаче.

Но можно сделать проще. В случае фронтенда на сервере нам нужна только готовая сборка, а репозиторий с исходным кодом понадобится только для разработки. Поэтому процесс может быть таким:

* осуществить сборку локально;
* скопировать папку build на удалённый сервер.

Собирать фронтенд вы уже умеете, осталось придумать, где на сервере расположить папку с ним, и скопировать её туда. Для этого:

* на удалённом сервере, рядом с папкой, где храните бэкенд на сервере, создайте папку для фронтенда. Назовите её, например, mesto-frontend;
* соберите файлы локально;
* скопируйте файлы с локального компьютера на удалённый. Это можно сделать командой scp (замените praktikum и 84.201.172.240 на свои логин и A-запись):

Скопировать кодBASH

scp -r ./build/\* praktikum@84.201.172.240:/home/praktikum/mesto-frontend

В таком виде команда scp принимает одну опцию и несколько аргументов. Опция -r говорит, что папки нужно копировать рекурсивно. Значит, если в папке dist есть вложенные папки, их содержимое тоже необходимо скопировать. Аргументы, которые принимает scp:

* ./build/\* — путь к локальным файлам или папкам, которые нужно скопировать. Build/\* означает, что нужно взять все файлы из папки build.
* praktikum@84.201.172.240:/home/praktikum/mesto-frontend — адрес удалённого сервера и папки на нём.

Важный момент: чтобы подключение к удалённому серверу было успешным, на нём должен быть наш SSH-ключ.

Для удобства можно добавить скрипт для деплоя в package.json проекта с фронтендом:

Скопировать кодJSON

"deploy": "npm run build && scp -r ./build/\* praktikum@84.201.172.240:/home/praktikum/mesto-frontend"

Такой скрипт сначала соберёт файлы, а затем скопирует их на сервер.

Осталось научить сервер их раздавать.

## Бэкенд на /api

Первый способ организовать фронтенд и бэкенд на одном домене — разместить на самом домене фронтенд, а API разместить на специальном пути — /api.

Скопировать код

domainname.students.nomoreparties.co — фронтенд

domainname.students.nomoreparties.co/api — API

Мы могли бы решить эту задачу только с использованием express:

Скопировать кодJAVASCRIPT

*// подключаем главный роутер приложения на /api*

app.use('/api', require('../router'));

*// раздаём папку с собранным фронтендом*

app.use(express.static(path.join(\_\_dirname, 'public')));

Это будет работать, но отдавать раздачу статичных файлов Node.js — не лучшая идея. Node.js хорош в исполнении логики и в асинхронных операциях ввода/вывода. С раздачей статичных файлов лучше справляется nginx. Он не только делает это быстрее, но и позволяет относительно несложно реализовать сжатие этих файлов. Кроме того, nginx работает в отдельном процессе и не загружает единственный поток Node.

Сконфигурируем nginx так, чтобы он отвечал за раздачу статики, а когда это необходимо, перенаправлял запросы Node. Для этого нужно отредактировать уже готовый конфигурационный файл nginx. Зайдите на сервер и введите команду:

Скопировать кодBASH

sudo nano /etc/nginx/sites-available/default

В текстовом редакторе nano откроется файл конфигурации nginx. Он будет выглядеть примерно так:

Скопировать код

server {

listen 80;

server\_name domainname.students.nomoreparties.co;

location / {

proxy\_pass http://localhost:3000;

proxy\_http\_version 1.1;

proxy\_set\_header Upgrade $http\_upgrade;

proxy\_set\_header Connection 'upgrade';

proxy\_set\_header Host $host;

proxy\_cache\_bypass $http\_upgrade;

}

}

В нём нужно изменить всего несколько вещей. Первым делом разместим API на пути /api. Для этого откорректируем блок location, который перенаправляет запрос Node.js приложению. Сейчас он выглядит так:

Скопировать код

location / {

proxy\_pass http://localhost:3000;

proxy\_http\_version 1.1;

...

}

Заменим / на /api/:

Скопировать код

location /api/ {

proxy\_pass http://localhost:3000;

proxy\_http\_version 1.1;

...

}

Теперь все запросы, путь которых начинается с /api/, будут перенаправлены Node.js серверу. Обратите внимание на висячий слеш. Он страхует от того, что API будут перенаправлены запросы вида: /api-some-more-words-after-api. Например:

Скопировать кодJAVASCRIPT

/api/hello/world — будет перенаправлен Node серверу

/api-hello-world — не будет

/apii — тоже не будет

Осталось научиться раздавать фронтенд. Для этого в конфигурацию nginx нужно добавить директиву root, передав ей путь к папке со статическими файлами:

Скопировать кодJSX

root /home/praktikum/mesto-frontend;

Добавить её можно где-то в блоке server, например, после директивы server\_name:

Скопировать код

server {

listen 80;

server\_name domainname.students.nomoreparties.co;

root /home/praktikum/mesto-frontend;

location /api/ {

...

Всё, что остаётся сделать после этого — перезапустить nginx. Перед этим обязательно проверьте изменённую конфигурацию.

Скопировать кодBASH

sudo nginx -t

*# Если ошибок не появилось:*

sudo systemctl restart nginx

В следующем уроке разберём размещение бэкенда на отдельном поддомене.

# Бэкенд на поддомене

Предыдущий способ — бэкенд на API — решает задачу разделения адресов для фронтенда и бэкенда. Но он не идеален. Ведь мы уже никак не можем использовать путь /api для фронтенда — он занят. Кроме этого, фронтенд и API должны храниться на одном сервере, мы не можем разнести их на разные машины.

Второй способ доменного разделения решает эти проблемы. В нём API хранится на отдельном поддомене, например:

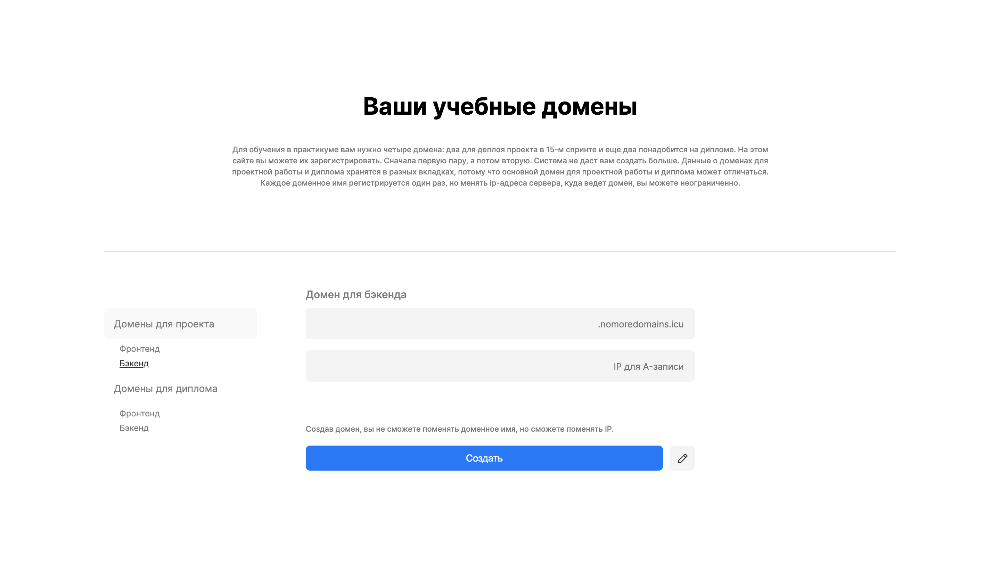
Скопировать код

domainname.students.nomoreparties.co — фронтенд

api.domainname.students.nomoreparties.co — API

Разберёмся, как это сделать.

Сначала нужно создать поддомен. Перейдите [по ссылке](http://domain.nomoreparties.site/) на уже знакомый вам сервис. Авторизуйте и перейдите во вкладку «Бэкенд» раздела «Домены для проекта»:



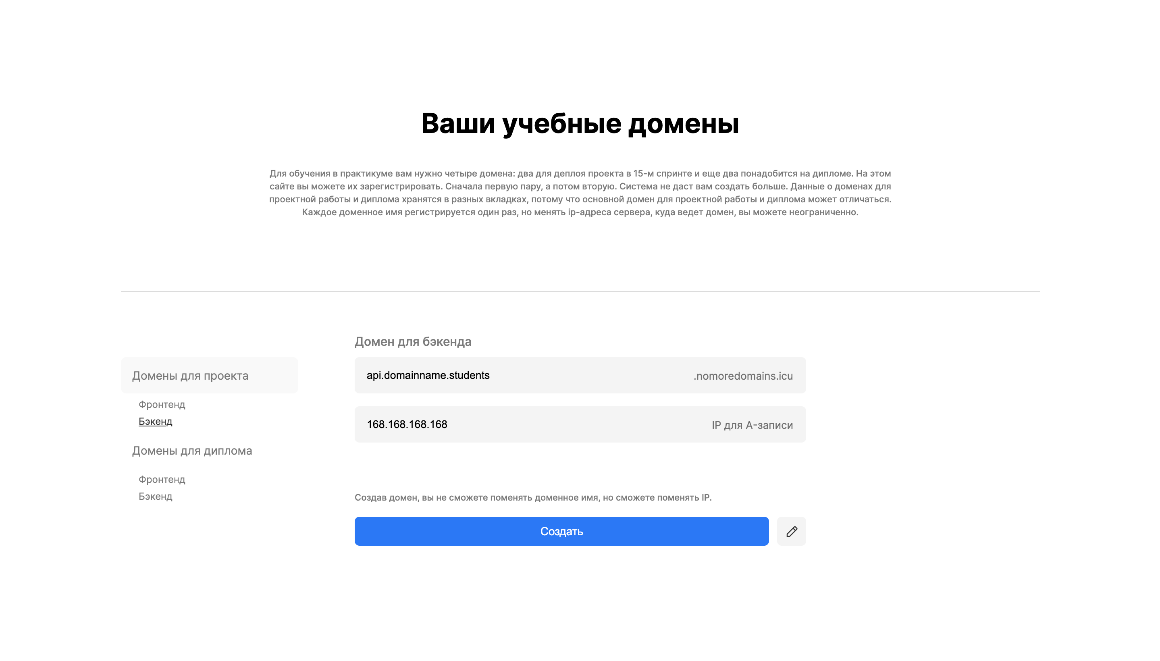
Отображение вкладки «Бэкенд»

В первое поле введите:

Скопировать код

api.domainname.students

Во второе поле введите публичный IPv4 со своего Яндекс.Облака.



Пример ввода данных при создании поддомена.

Сконфигурируем nginx. Возьмём исходный файл nginx:

Скопировать код

server {

listen 80;

server\_name domainname.students.nomoreparties.co;

location / {

proxy\_pass http://localhost:3000;

proxy\_http\_version 1.1;

proxy\_set\_header Upgrade $http\_upgrade;

proxy\_set\_header Connection 'upgrade';

proxy\_set\_header Host $host;

proxy\_cache\_bypass $http\_upgrade;

}

}

В нём нужно изменить только имя сервера, в начале добавив API:

Скопировать код

server {

listen 80;

server\_name api.domainname.students.nomoreparties.co; # изменили имя сервера

...

}

После этого полностью скопируйте уже имеющуюся конфигурацию сервера и вставьте в самом конце файла. Должно получиться два одинаковых блока server:

Скопировать код

server {

listen 80;

server\_name api.domainname.students.nomoreparties.co;

...

}

server {

listen 80;

server\_name api.domainname.students.nomoreparties.co;

...

}

Первый блок перенаправляет запросы к API, а второй должен раздавать фронтенд. В нём нужно изменить имя сервера обратно, так как фронтенд должен передаваться при заходе на domainname.students.nomoreparties.co (а не на api.domainname.students.nomoreparties.co):

Скопировать код

server {

listen 80;

server\_name domainname.students.nomoreparties.co;

...

}

Дальше укажите путь к папке с фронтендом:

Скопировать код

server {

listen 80;

server\_name domainname.students.nomoreparties.co;

root /home/praktikum/mesto-frontend;

...

}

В конце второго блока server нужно удалить блок location, перенаправляющий запросы к Node.js. В итоге полностью второй блок должен выглядеть так:

Скопировать код

server {

listen 80;

server\_name domainname.students.nomoreparties.co;

root /home/praktikum/mesto-frontend;

}

А весь файл конфигурации выглядит так:

Скопировать код

server {

listen 80;

server\_name api.domainname.students.nomoreparties.co;

location / {

proxy\_pass http://localhost:3000;

proxy\_http\_version 1.1;

proxy\_set\_header Upgrade $http\_upgrade;

proxy\_set\_header Connection 'upgrade';

proxy\_set\_header Host $host;

proxy\_cache\_bypass $http\_upgrade;

}

}

server {

listen 80;

server\_name domainname.students.nomoreparties.co;

root /home/praktikum/mesto-frontend;

}

По уже сложившейся традиции проверьте и перезапустите nginx:

Скопировать кодBASH

sudo nginx -t

*# Если ошибок не появилось:*

sudo systemctl restart nginx

Теперь при заходе на основной домен nginx будет отдавать фронтенд, а при обращении на поддомен api, запрос будет перенаправляться серверу Node.js.

## Добавляем поддержку роутинга на фронтенде

По умолчанию nginx считает, что запрос пользователя обращён к какому-либо файлу. Если этот файл не обнаружен, сервер возвращает ошибку 404. Это приводит к тому, что nginx воспринимает маршруты на фронтенде как обращения к файлам, и роутинг перестаёт работать корректно. Исправим это.

Поскольку вы размещаете фронтенд и бэкенд на разных поддоменах, значит, блок location / у вас уже есть и потребуется лишь добавить директиву try\_files $uri $uri/ /index.html; в его конец:

Скопировать кодJSON

location / {

...

try\_files $uri $uri/ /index.html;

}

Обратите внимание, что нужно добавить директиву именно для домена, на котором расположен фронтенд, а не бэкенд.

Эта директива сообщает nginx, что при обработке запроса сначала следует интерпретировать его как путь к файлу, затем к папке, а если ни того, ни другого не найдено, передавать его фронтенду в index.html. Теперь всё работает, как нужно: если по пути есть файл, nginx его отдаёт, в противном случае считает запрос маршрутом на фронтенде.

# Шифрование данных. Протокол HTTPS

Между клиентом и сервером происходит обмен информацией. Часто она конфиденциальна и нужно сделать всё, чтобы у злоумышленника не было к ней доступа. Для этого информацию между клиентом и сервером передают в зашифрованном виде. Шифрование осуществляется при помощи SSL-сертификата. Разберёмся, что это.

## Что такое сертификат

SSL расшифровывается как Secure Socket Layer (англ. «уровень защищённых сокетов»).

С помощью SSL-сертификата информация, передаваемая между клиентом и сервером, шифруется. Теперь, даже если злоумышленнику удастся её перехватить, он не сможет извлечь из этого пользы, так как информация зашифрована.

SSL-сертификат выдаётся так называемой Certificate Authority или CA (англ. «центр сертификации»). CA — третья сторона, которой клиент и сервер могут доверять. Она выдаёт сертификат на основе доменного имени и информации о сервере. Сертификат выпускается на доменное имя и его нужно создавать, зайдя на сервер. Проверив доменное имя и IP-адрес сервера, куда оно указывает, CA убедится, что сертификат выпускает именно владелец домена, и выдаст его.

После этого вся передаваемая информация может быть зашифрована и подписана с использованием выпущенного сертификата. А клиент сможет убедиться, что подписана информация именно сертификатом сервера и никаким другим. Таким образом, клиент и сервер смогут шифровать информацию и проверять, кем она зашифрована, это исключает изменение и подлог данных.

Некоторые CA выпускают сертификаты за деньги, но есть и бесплатные. Одна из самых популярных — Let's Encrypt. Её сертификатом мы и воспользуемся.

## Выпускаем сертификат

SSL-сертификат от Letsencrypt можно выпустить специальной программой — certbot.

[На сайте certbot](https://certbot.eff.org/lets-encrypt/ubuntubionic-nginx) есть описание процесса, как выпустить сертификат и сконфигурировать nginx, чтобы он его использовал. Можете сделать это самостоятельно или воспользоваться дальнейшей инструкцией.

Зайдите на сервер и по очереди введите команды:

Скопировать кодBASH

sudo apt update

sudo apt install -y certbot python3-certbot-nginx

Теперь certbot установлен.

## Подключаем сертификат

Чтобы подключить выпущенный сертификат, введите команду:

Скопировать кодBASH

sudo certbot --nginx

В процессе исполнения вам нужно несколько вопросов:

* Enter email address (англ. «введите почту»). Почта нужна для предупреждений, что сертификат пора обновить.
* Please read the Terms of Service; (A)gree/(C)ancel: (англ. «прочитайте правила сервиса; принять/отклонить»). Прочитайте правила по ссылке и введите a. Затем нажмите Enter.
* Would you be willing to share your email address with the Electronic Frontier Foundation? (англ. «хотите ли вы поделиться своей почтой с Фондом электронных рубежей»). Отметьте на своё усмотрение y (да) или n (нет) и нажмите Enter.
* Which names would you like to activate HTTPS for? (англ. «для каких доменных имён вы хотите включить https?»). Вам будет предложено несколько вариантов: domainname.students.nomoreparties.co и поддомен с api.. Это доменные имена, которые мы добавили в поле server\_name конфигурации nginx в предыдущих уроках. Ничего не вводите, просто нажмите Enter. Тогда https будет включён для обеих опций.
* Please choose whether or not to redirect HTTP traffic to HTTPS? 1: No redirect, 2: Redirect (англ. «нужно ли перенаправлять http траффик на https, 1: не перенаправлять, 2: перенаправлять»). Выберите 1 и нажмите Enter.

В итоге сертификаты будут выпущены. Также эта команда отредактирует конфигурацию nginx: добавит в неё нужные настройки и пропишет пути к сертификату.

Не забудьте перезапустить nginx:

Скопировать кодBASH

sudo systemctl reload nginx

Теперь весь трафик, который приходит на порт 443 (порт для https-запросов и ответов), будет шифроваться.

Откройте Postman и попробуйте отправить https-запрос — всё должно получиться. Это значит, что сертификат успешно подключён и информация между клиентом и сервером передаётся в зашифрованном виде.

## Обновление сертификата

Выпущенный сертификат нужно обновлять минимум раз в 3 месяца. Когда придёт время, воспользуйтесь командой:

Скопировать кодBASH

sudo certbot renew --pre-hook "service nginx stop" --post-hook "service nginx start"

Она обновит сертификат и перезапустит nginx.

# Заключение

Два месяца бэкенд-разработки на исходе. Вы узнали много нового: как работает сервер, как строить API, как делать его безопасным и как его разворачивать. Теперь, если на работе вам дадут задачу исправить что-то в серверном коде, это уже не будет казаться невозможным.

Более того, вы сможете самостоятельно сделать полноценное веб-приложение с фронтендом и бэкендом. Кроме большей свободы на работе, это и большая творческая свобода.

Впереди последняя проектная работа программы по веб-разработке.

[@Александр Катрюхин](https://yandex-students.slack.com/team/U01F0MCBYVB) привет, да, лучше удалить и залить новые.  
Команда для удаления в Linux rm -r имя\_папки - удалит саму папку все файлы в ней и подпапки (edited)

Также можно просто пересоздать виртуальную машину, чтобы не тратить время на чистку и т.п. и не наделать ошибок во время чистки))

возможно у тебя 443 порт закрыт.  
Проверь, пожалуйста, на сервере  
ufw status verbose

[@Михаил Зятьков](https://yandex-students.slack.com/team/U01EVLJ5XJM)  
Спасибо за подсказку по поводу порта, открыл командой

sudo ufw allow 443/tcp