7.4

Введение

Эта практика выполняется у вас локально.

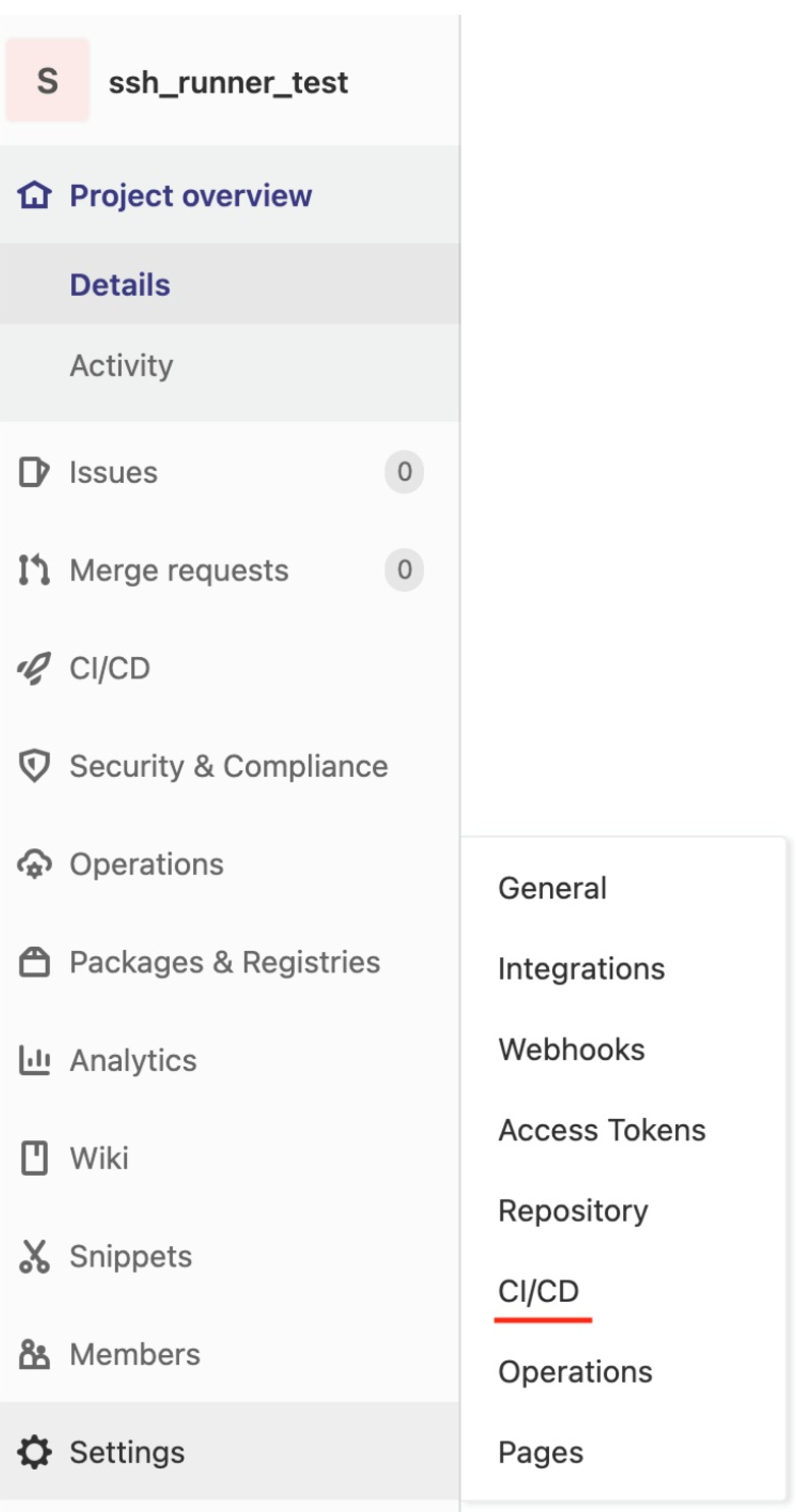
Сегодня мы с вами познакомимся с установкой и оперированием GitLab Runner вместе с Ansible.
Для этого мы установим GitLab Runner и используем его для запуска простого плейбука Ansible.

Установка раннера

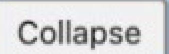
Инструкция по установке GitLab Runner находится в настройках проекта, в разделе CI/CD. Вы можете также установить раннер глобально, в настройках самого GitLab в том же разделе. Но процесс установки самого раннера не отличается никак, поэтому мы установим раннер локально для проекта.

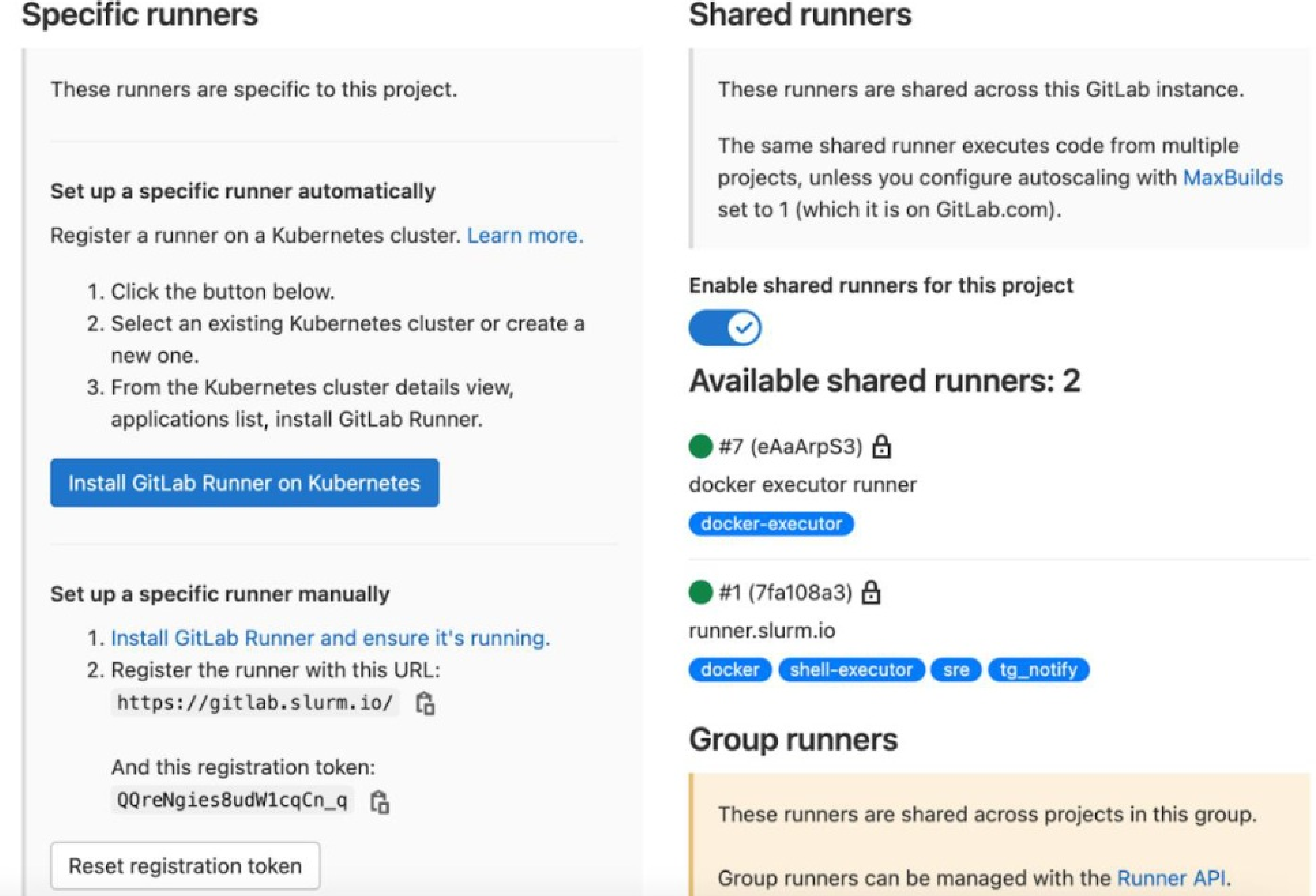
Выбираем пункт CI/CD из настроек проекта. Если вы не видите настроек, скорее всего вы не Owner проекта, поставьте права выше или зайдите от администратора.

КУПЛЕНО НА SKLADCHIK.COM



В разделе “CI/CD” мы видим раздел Gitlab runners и можем кликнуть Expand, чтобы развернуть его.





Мы будем устанавливать раннер вручную, кликнув на Install Gitlab Runner and ensure it's running.

Это откроет страницу с инструкциями которым мы будем следовать.

Install GitLab Runner ALL TIERS

GitLab Runner can be installed and used on GNU/Linux, macOS, FreeBSD, and Windows. You can install it:

- In a container.
- By downloading a binary manually.
- By using a repository for rpm/deb packages.

GitLab Runner officially supported binaries are available for the following architectures:

- x86, AMD64, ARM64, ARM, s390x

Official packages are available for the following Linux distributions:

- CentOS, Debian, Ubuntu, RHEL, Fedora, Mint

GitLab Runner officially supports the following operating systems:

- Linux, Windows, macOS, FreeBSD

You can find information on the different installation methods below. You can also view installation instructions in GitLab by going to your project's **Settings > CI / CD**, expanding the **Runners** section, and clicking **Show runner installation instructions**.

Repositories

- Install using the GitLab repository for Debian/Ubuntu/CentOS/RedHat

Binaries

- Install on GNU/Linux
- Install on macOS
- Install on Windows

Давайте залогинимся по ssh на машину, содержащую в себе установленный Ansible и установим раннер туда. Для установки раннера нам обязательно понадобится пользователь с sudo.

Мы воспользуемся самым простым путем установки раннера - просто добавим maintained пакет с их репозитория.

```
# Для Debian/Ubuntu/Mint
curl -L "https://packages.gitlab.com/install/repositories/runner/gitlab-runner/script.deb.sh" | sudo bash

# Для RHEL/CentOS/Fedora
curl -L "https://packages.gitlab.com/install/repositories/runner/gitlab-runner/script.rpm.sh" | sudo bash
```

```
[s013895@node-1.s013895.slurm.io ~]$ curl -L "https://packages.gitlab.com/install/repositories/runner/gitlab-runner/script.rpm.sh" | sudo bash
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  6904  100  6904    0     0   6966      0 --:--:-- --:--:-- --:--:--  6959
Detected operating system as centos/7.
Checking for curl...
Detected curl...
Downloading repository file: https://packages.gitlab.com/install/repositories/runner/gitlab-runner/config_file.repo?os=centos&dist=7&source=script
done.
Installing pygpgme to verify GPG signatures...
Loaded plugins: fastestmirror, protectbase
Loading mirror speeds from cached hostfile
 * epel: www.nic.funet.fi
runner_gitlab-runner-source/signature                                   |  862 B  00:00:00
Retrieving key from https://packages.gitlab.com/runner/gitlab-runner/gpgkey
Importing GPG key 0x51312F3F:
 Userid     : "GitLab B.V. (package repository signing key) <packages@gitlab.com>"
 Fingerprint: f640 3f65 44a3 8863 daa0 b6e0 3f01 618a 5131 2f3f
 From       : https://packages.gitlab.com/runner/gitlab-runner/gpgkey
Retrieving key from https://packages.gitlab.com/runner/gitlab-runner/gpgkey/runner-gitlab-runner-4C80FB51394521E9.pub.gpg
runner_gitlab-runner-source/signature                                   |  951 B  00:00:01 !!!
runner_gitlab-runner-source/primary                                     |  175 B  00:00:01
0 packages excluded due to repository protections
Package pygpgme-0.3-9.el7.x86_64 already installed and latest version
Nothing to do
Installing yum-utils...
Loaded plugins: fastestmirror, protectbase
Loading mirror speeds from cached hostfile
 * epel: www.nic.funet.fi
0 packages excluded due to repository protections
Package yum-utils-1.1.31-54.el7_8.noarch already installed and latest version
Nothing to do
Generating yum cache for runner_gitlab-runner...
Importing GPG key 0x51312F3F:
 Userid     : "GitLab B.V. (package repository signing key) <packages@gitlab.com>"
 Fingerprint: f640 3f65 44a3 8863 daa0 b6e0 3f01 618a 5131 2f3f
 From       : https://packages.gitlab.com/runner/gitlab-runner/gpgkey
Generating yum cache for runner_gitlab-runner-source...

The repository is setup! You can now install packages.
```

ВНИМАНИЕ! В официальном репозитории Ubuntu уже содержится runner старой версии, поэтому для установки раннера именно из репозитория gitlab надо запинить гитлаб с более высоким приоритетом. Делается это следующим образом.

```
cat <<EOF | sudo tee /etc/apt/preferences.d/pin-gitlab-runner.pref
Explanation: Prefer GitLab provided packages over the Debian native ones
Package: gitlab-runner
Pin: origin packages.gitlab.com
Pin-Priority: 1001
EOF
```

Таким образом apt будет ставить не раннер по умолчанию, а именно тот что вы хотите. Для RHEL, Fedora и прочих не Debian дистрибутивов ничего такого делать не надо.

После добавления репозитория мы можем просто запустить команду yum для установки раннера.

```
# Для Debian/Ubuntu/Mint
sudo apt-get install gitlab-runner

# Для RHEL/CentOS/Fedora
sudo yum install gitlab-runner
```

После установки проверим статус командой

```
ps aux | grep runner
```

Либо

```
systemctl status gitlab-runner
```

В обоих случаях runner должен отображаться как running. На моем опыте единственная ошибка, которую я получал на данном этапе - недостаточно разрешений у пользователя, либо закрытые для записи папки, в которых раннер хранит артефакты и конфигурацию. (/home/gitlab-**runner и /etc/gitlab-runner/**) Эти папки должны быть открыты для записи.

После установки раннера время настроить и зарегистрировать его в gitlab.

Регистрация раннера

Регистрация раннера необходима для установки соединения между ним и GitLab, чтобы мы могли его использовать при билде своих проектов. GitLab предоставляет подробную инструкцию и сравнения всех возможных раннеров.

Selecting the executor

The executors support different platforms and methodologies for building a project. The table below shows the key facts for each executor which will help you decide which executor to use.

Executor	SSH	Shell	VirtualBox	Parallels	Docker	Kubernetes	Custom
Clean build environment for every build	✗	✗	✓	✓	✓	✓	conditional (4)
Reuse previous clone if it exists	✓	✓	✗	✗	✓	✗	conditional (4)
Runner file system access protected (5)	✓	✗	✓	✓	✓	✓	conditional
Migrate runner machine	✗	✗	partial	partial	✓	✓	✓
Zero-configuration support for concurrent builds	✗	✗ (1)	✓	✓	✓	✓	conditional (4)
Complicated build environments	✗	✗ (2)	✓ (3)	✓ (3)	✓	✓	✓
Debugging build problems	easy	easy	hard	hard	medium	medium	medium

Если вы новичок в пайплайнах GitLab, я рекомендую установить раннеры изначально в состояние Shell или Ssh, для легкости отладки.

Мы будем использовать регистрацию раннера в режиме Shell. В этом режиме все выполнение скриптов происходит непосредственно на машине. Очень легкий для отладки режим, потому что всегда есть возможность посмотреть различные логи, и что-то подкрутить руками в машине. Не рекомендуется для продакшена, потому что какие нибудь плохо написанные пайплайны сборки могут сломать саму машину и придется начинать все заново.

Запускаем команду

```
sudo gitlab-runner register --url $REGISTRATION_URL--registration-token
$REGISTRATION_TOKEN
```

Где вам нужно заменить $REGISTRATION_URL на ваш
URL GitLab (например https://gitlab.slurm.io/) а $REGISTRATION_TOKEN на ваш токен регистрации (отображается в раделе registration token в секции CI/CD)

Команда запускается в интерактивном режиме и ожидает от вас подтверждения вопросов.

Первый вопрос — адрес GitLab для регистрации. Должен отображаться тот, который вы задали. Если это так, просто жмите Enter

```
Runtime platform                                    arch=amd64 os=linu
Running in system-mode.

Enter the GitLab instance URL (for example, https://gitlab.com/):
[https://gitlab.slurm.io/]:
```

Потом то же самое с токеном регистрации, просто Enter.

Дальше будет предложено ввести описание для раннера, здесь вы можете написать что хотите, по умолчанию там просто hostname машины. Я рекомендую писать какие-то значимые описания для продакшена, например Gitlab in cluster east-sa (east-sa connections) или Webpacker. Это поможет вам лучше ориентироваться в будущем.

```
Enter a description for the runner:
[node-1.s013895.slurm.io]:
```

Далее приложение предложит нам указать список тегов. Это те «маркеры», по которым вы можете запускать работы в Gitlab. Например, у вас три машины: на одной установлен Ansible, на другой Java компилятор, на третьей Nodejs. Таким образом, логично выполнять сборку Java на машине с Java, а сборку фронтенда на машине с nodejs, а деплой и сборку софта на машинах через машину с Ansible. И вы проставляете им соотвествующие теги, которые затем используете в своих пайплайнах.

Для нашей тестовой машины я поставлю тег test

```
Enter tags for the runner (comma-separated):
test
```

Далее раннер успешно зарегистрируется и нам как раз необходимо будет выбрать исполняющий метод для запуска работ.

```
Enter an executor: docker-ssh, shell, docker+machine, ssh, virtualbox, docker-ssh+machine, kubernetes, custom, docker, parallels:
```

Я выберу метод shell.

```
shell
Runner registered successfully. Feel free to start it, but if it's running already the config should be automatically reloaded!
```

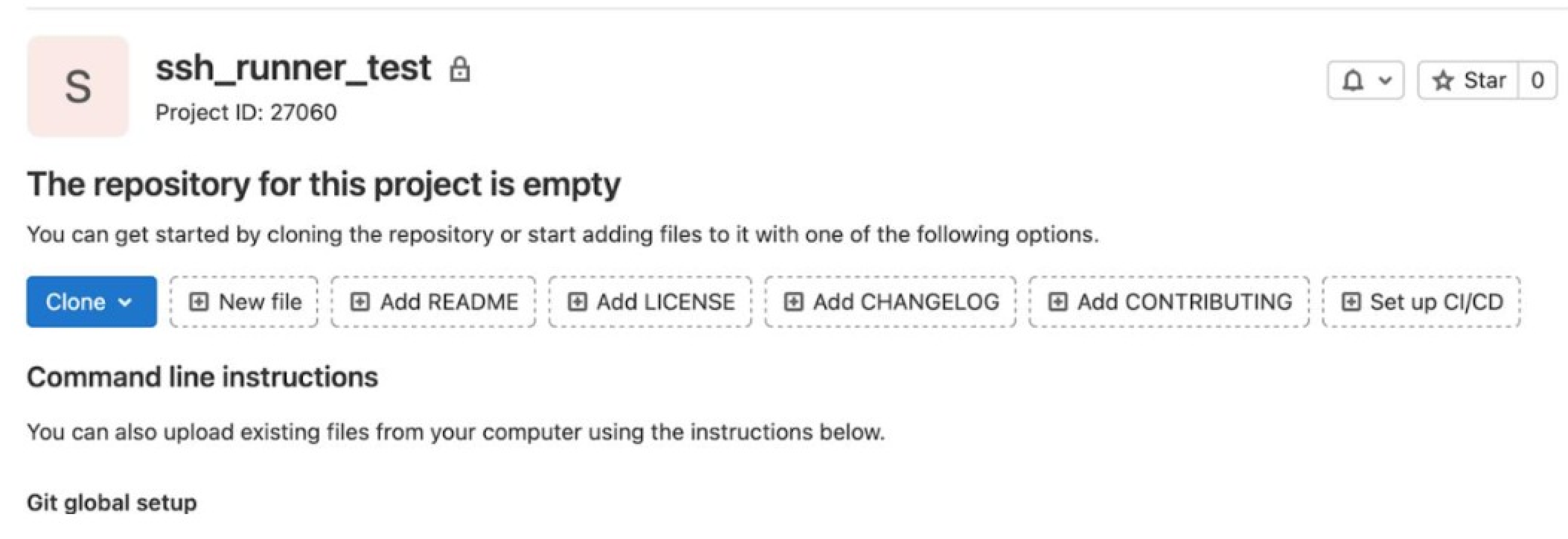
После этого мы видим жизнеутверждающую надпись, указывающую нам что раннер готов к работе, можно переходить непосредственно в Gitlab и создать работу для раннера!

Сам gitlab

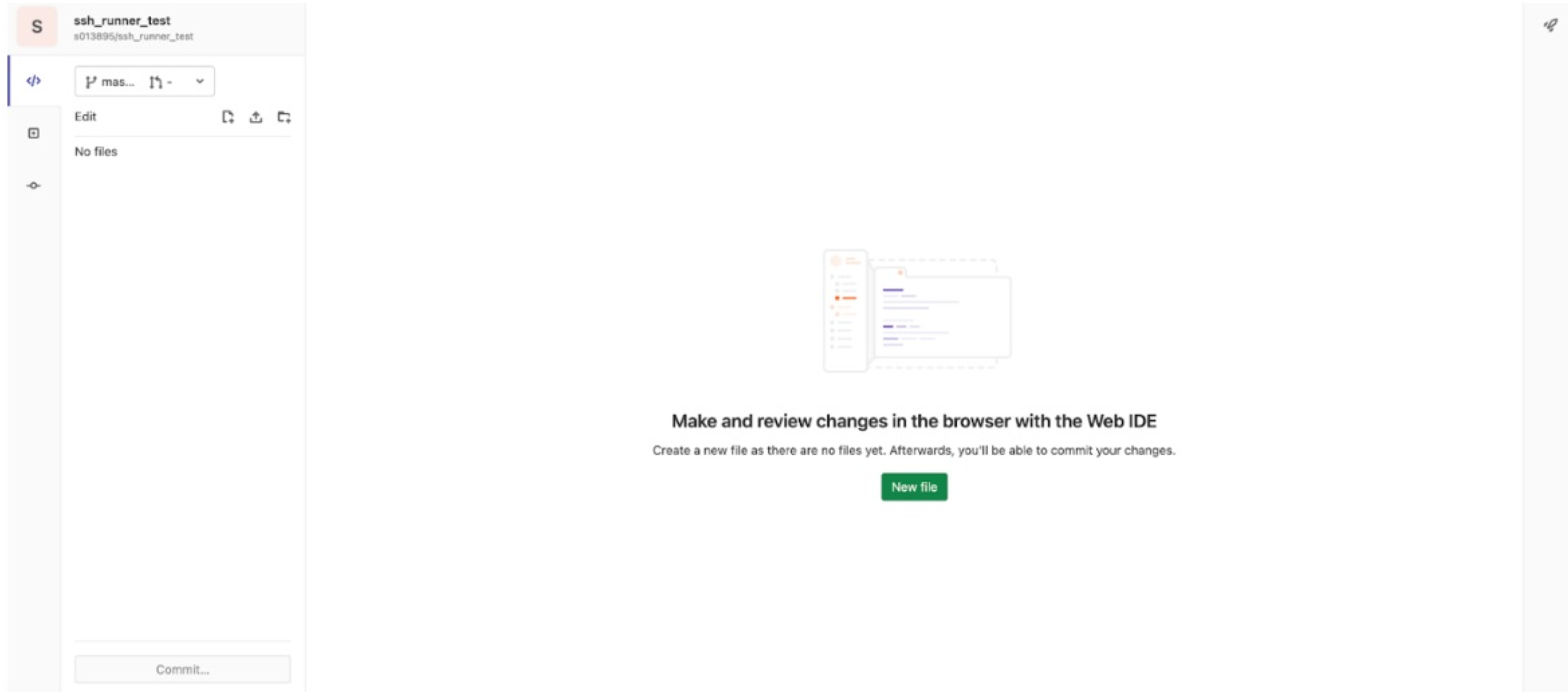
После установки и регистрации раннера вы увидите его в своем проекте в той же вкладке CI/CD.

Если его не видно, то скорее всего, ваш фаервол блокирует входящие соединения. Попробуйте открыть порты 8093 или 22. Gitlab использует файлы gitlab-ci, представляющие yml файлы, с описанием шагов пайплайна. Эти файлы пишутся на языке yaml и описывают просто шаги для запуска работы. К нашему счастью, Gitlab предоставляет шаблоны для ci, которые мы можем использовать при создании файла изнутри Gitlab. Здесь я приведу один из них и объясню что происходит.

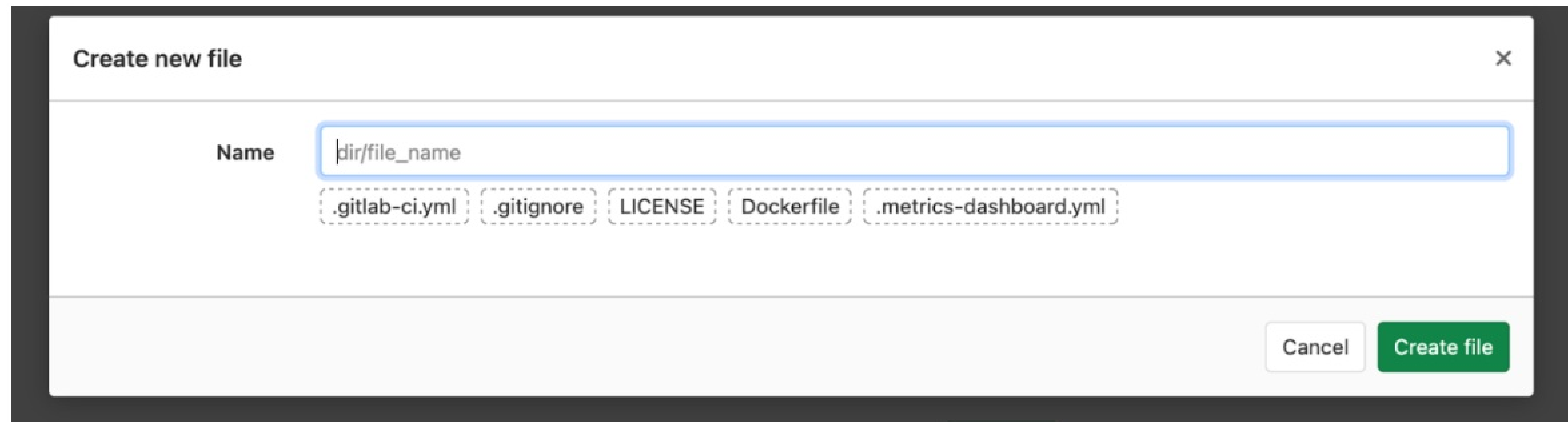
Итак, переходим на главную проекта, если вы его только создали то он пуст.



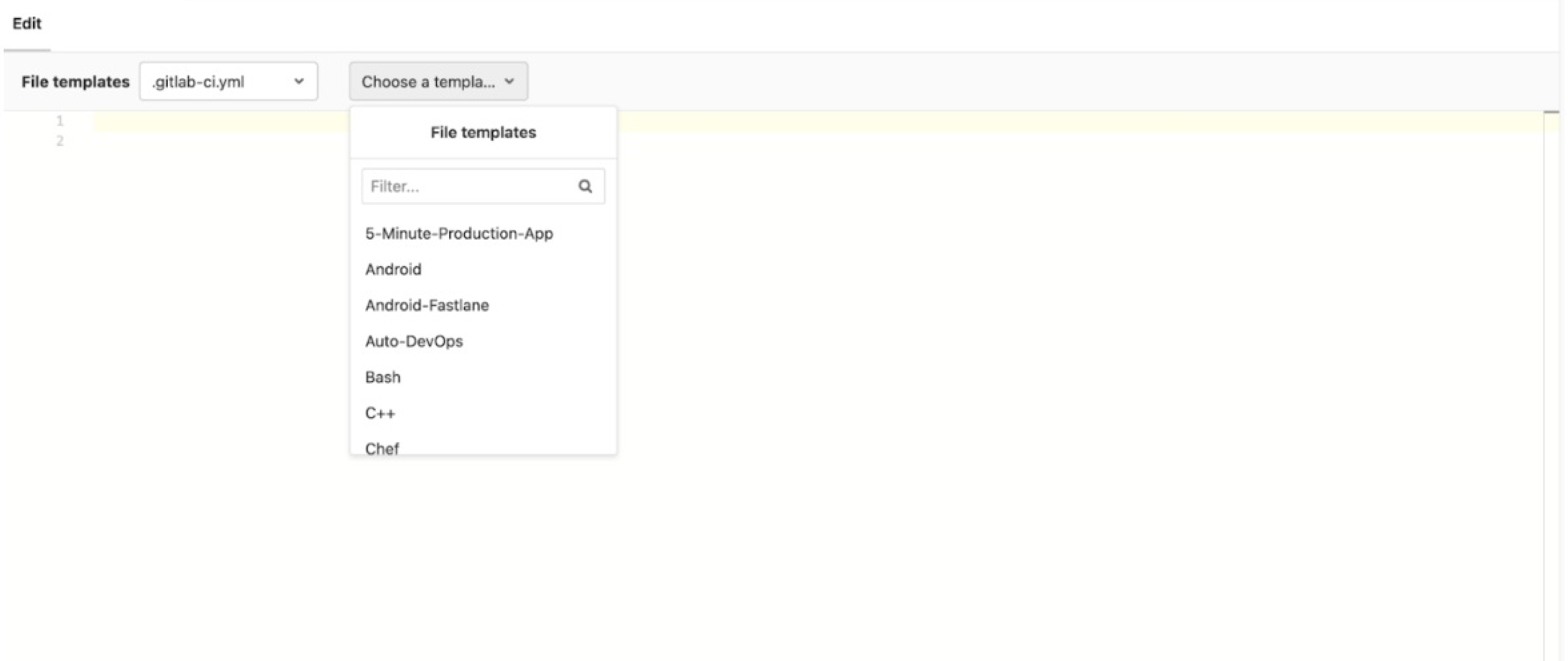
Жмякаем new file



Еще раз New file и выбираем .gitlab-ci.yml



И теперь при выборе у нас открывается окно редактирования со списком шаблонов для использования.



Выберем шаблон 5 Minutes Produiction App и посмотрим на сгенерированный шаблон

Я сосредоточусь только на описании значимых для нас частей, за более подробной инструкцией, пожалуйста, обратитесь к https://gitlab.com/gitlab-org/5-minute-production-app/deploy-template/-/blob/v2.3.0/README.md

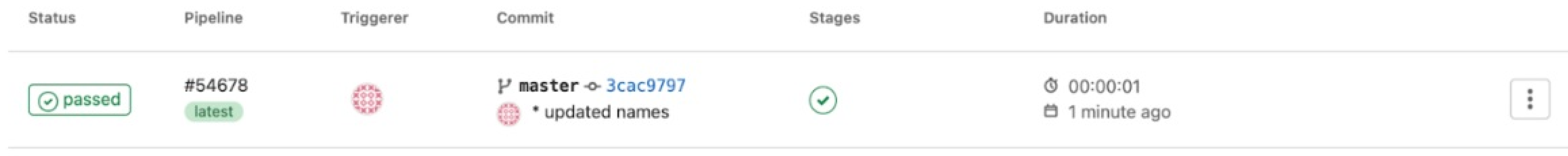
```
# Этот файл только пример, но мы рассмотрим его чтобы понять структуру пайплайнов
гитлаба

include:
 # здесь размещаются дополнительные скрипты, чтобы предотвратить дублирование
пайплайнов, они определяют дополнительные условия для скриптов и работают аналогично
include тасков в Ansible
 - template: 'Workflows/Branch-Pipelines.gitlab-ci.yml'
 - template: 'Jobs/Build.gitlab-ci.yml'

# Stages определяют стадии пайплайна, например работы по сборке, запуск тестов,
поднятие серверов, деплой, удаление серверов.
Stages:
 - build
 - test
 - provision
 - deploy
 - destroy


# В переменных содержатся значения, которые можно использовать в скриптах. Обратите
внимание на ${имя_переменной} - гитлаб подставит значения в эти переменные во время
выполнения
variables:
 TF_ADDRESS:
${CI_API_V4_URL}/projects/${CI_PROJECT_ID}/terraform/state/${CI_COMMIT_REF_SLUG}
 TF_VAR_ENVIRONMENT_NAME:
${CI_PROJECT_PATH_SLUG}_${CI_PROJECT_ID}_${CI_COMMIT_REF_SLUG}
 TF_VAR_SERVICE_DESK_EMAIL: incoming+${CI_PROJECT_PATH_SLUG}-${CI_PROJECT_ID}-issue-
@incoming.gitlab.com
 TF_VAR_SHORT_ENVIRONMENT_NAME: ${CI_PROJECT_ID}-${CI_COMMIT_REF_SLUG}
 TF_VAR_SMTP_FROM: ${SMTP_FROM}


# это непосредственно работа в пайплайне, как и таск в Ansible начинается с имени
terraform_apply:
 stage: provision # определяем на какой стадии данная работа
 image: registry.gitlab.com/gitlab-org/5-minute-production-app/deploy-template/stable
# для docker runner мы можем определить контейнер, содержащий все нужные компоненты и
софт для выполнения работы
 resource_group: terraform # jobы складывают артефакты в определенные места на
раннере, чтобы избежать коллизий мы определяем ресурсную группу для этих мест и
гарантируем что только те работы что мы хотим могут туда писать и никто ничего не
перезапишет
 Before_script: # команды делаемые перед скриптом, аналогично pre_tasks
   - cp /*.tf .
   - cp /deploy.sh .
 Script: # и сам скрипт, состоящий из набора bash команд
   - gitlab-terraform init
   - gitlab-terraform plan
   - gitlab-terraform plan-json
   - gitlab-terraform apply

deploy:
 stage: deploy
 image: registry.gitlab.com/gitlab-org/5-minute-production-app/deploy-template/stable
 resource_group: deploy
 before_script:
   - cp /*.tf .
   - cp /deploy.sh .
   - cp /conf.nginx .
 script:
```

```
    - ./deploy.sh
  artifacts:
    reports:
      dotenv: deploy.env # здесь мы определяем так называемые артефакты скрипта - 
файлы, образовавшиеся в результате работы скрипта.
  Environment: # переменные env для использования скриптом
    name: $CI_COMMIT_REF_SLUG
    url: $DYNAMIC_ENVIRONMENT_URL
    on_stop: terraform_destroy

terraform_destroy:
  variables:
    GIT_STRATEGY: none
  stage: destroy
  image: registry.gitlab.com/gitlab-org/5-minute-production-app/deploy-template/stable
  before_script:
    - cp /*.tf .
    - cp /deploy.sh .
  script:
    - gitlab-terraform destroy -auto-approve
  environment:
    name: $CI_COMMIT_REF_SLUG
    action: stop
  rules:
    - if: '$AWS_ACCESS_KEY_ID && $AWS_SECRET_ACCESS_KEY && $AWS_DEFAULT_REGION && 
$CI_COMMIT_REF_PROTECTED == "false"'
      when: manual
    - when: never
```

Мы используем знания из шаблона, чтобы сделать свой шаблон, который просто запустит скрипт, показывающий нам версию Ansible.

```
stages:
  - provision # оставим только один шаг


ansible_run:
  stage: provision # запускаем работу в provision
  tags:
    - test #этот тег нужен для того чтобы отправить работу на конкретную машину с 
ansible
  script:
    - ansible --version # сам скрипт который покажет нам версию

Если вы все сделали правильньо то после нажатия кнопки Commit в интерфейсе Гитлаба вы 
увидите следующее
```

Если вы все сделали правильньо, то после нажатия кнопки Commit в интерфейсе Гитлаба вы увидите следующее



Слева направо - статус работы, номер пайплайна, пользователь его вызвавший, коммит, и как раз эти этапы которые мы задавали, у нас один, поэтому только один кружочек, а также длительность.

Если мы кликнем по кружочку, то увидим работы которые выполняются на этапе, там будет только наша ansible_run. Если мы кликнем по ней то увидим полный лог выполнения.

```
1  Running with gitlab-runner 14.1.0 (8925d9a0)
2    on Test runner SF4FSvnj
3  Preparing the "shell" executor                                  00:00
4  Using Shell executor...
6  Preparing environment                                           00:00
7  Running on node-1.s013895.slurm.io...
9  Getting source from Git repository                              00:00
10 Fetching changes with git depth set to 50...
11 Reinitialized existing Git repository in /home/gitlab-runner/builds/SF4FSvnj/0/s013895/ssh_runner_test/.git/
12 Checking out 3cac9797 as master...
13 Skipping Git submodules setup
15 Executing "step_script" stage of the job script                 00:01
16 $ ansible --version
17 ansible 2.9.20
18   config file = None
19   configured module search path = ['/home/gitlab-runner/.ansible/plugins/modules', '/usr/share/ansible/plugins/modules']
20   ansible python module location = /usr/local/lib/python3.6/site-packages/ansible
21   executable location = /usr/local/bin/ansible
22   python version = 3.6.8 (default, Nov 16 2020, 16:55:22) [GCC 4.8.5 20150623 (Red Hat 4.8.5-44)]
24 Cleaning up file based variables                                00:00
26 Job succeeded
```

Теперь давайте попробуем написать простой плейбук и запустить его с помощью Ansible.

Наш плейбук будет состоять из файла hosts, ansible.cfg и обычного playbook.yml.

Внимание! Скорее всего раннер не будет включать в себя программу sshpass, необходимую для авторизации по паролю в ssh. Тогда ее надо будет установить:)

Hosts.ini

```
testnode ansible_host=node-2.ваш_пользователь ansible_ssh_pass=ваш_пароль
ansible_user=ваш_пользователь
```

Ansible.cfg

```
[defaults]
host_key_checking = False #отключим проверку ключей чтобы не ловить ошибки
```

Наш плейбук

```
---
- name: "Ping hosts"
  hosts: all
  gather_facts: false
  tasks:
  - name: "Ping host"
    ansible.builtin.ping:
```

И меняем файл .gitlab-ci.yml

```
stages:
  - provision


ansible_run:
```

```
  stage: provision
  tags:
    - test
  script:
    - ansible-playbook playbook.yml -i hosts.ini
```

Поздравляю! Мы освоили запуск Ansible в Gitlab CI и, как видим, здесь нет ничего сложного. Стоит понимать, что все работы запускаются из папки /home/gitlab-runner/. Таким образом пути, используемые в скриптах (например ssh ключи), надо строить оттуда, а еще лучше использовать абсолютные пути. Практически все системы CI работают схожим образом, после освоения Gitlab CI, вам необходимо будет только переехать на другую систему скриптов и управляющих команд.

SSH ключи в раннере

Реальная жизнь такова, что вы вряд ли захотите запускать работы Ansible, явно указывая пароль в hosts.ini. Мало того, что любой может их увидеть, это еще и неудобно, с точки зрения хранения.

Этот метод безусловно имеет право на жизнь, но мы с вами передадим ключ из Gitlab и попробуем использовать его для авторизации.

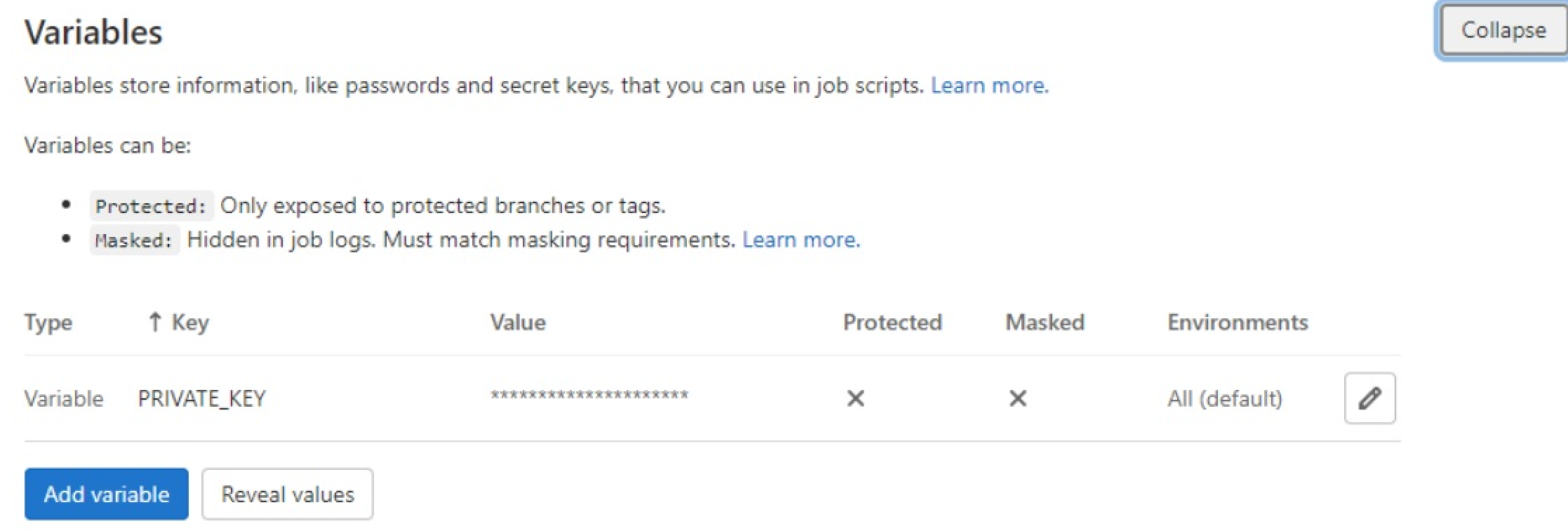
Тестовые стенды для удобства настроены так что ключи root везде авторизованы. Это позволяет ходить на стенды под рутом не указывая ключ ssh. Нам это не надо, мы хотим передать ключ напрямую, поэтому прокачиваемся до рута с помощью sudo su.

А потом забираем ключ из ~/.ssh/id_rsa.

```
[root@node-1.s013895.slurm.io /home/s013895]# cat ~/.ssh/id_rsa
-----BEGIN RSA PRIVATE KEY-----
MIIEpAIBAAKCAQEApSWRcRI+oePNEzY/dp9zzHpDiEghm4sxCfth3d/UZV2y6m7L
Q7cWNeXS6zdFh8L7Pwvz0XxYEN9AfFsfZGWQlxkP3iD+b/a/A5eirK9wrklGpKjO
41RyO9CQrfFFQExbhqJrmGcNcj8vpslKFUcgYy000jJxC54k0Zimlu/JWAg4JDYg
btw1+k16fUmVKqJSxoTMN7xOLHcsguy+aSlZqApY19NIoW6rwIYeucPNDPj4ml3Y
DJX074GFmLb0dydrHBglFMYvPGXOms4suqOkQ8ljipDyeMTxOxXG5Ghs8hr1azR5
p5zlrlkl3gpjVCKy80BlCqs0q465KWJKIT6FPwIDAQABAoIBAQCia5NiWGY/A/Mi
TP0XpnTWCv02IHLlLn0ZayHye/vBgejABh1lSMDQW2sYKPD6N7qwZAn04s0eMuOS
KnByZi1RXbWsalizN0LfvpegLJ+u2aRftyJalCgRSeuxFxRby31MqZsWdWvVC+3K
pDd3O8/PmptTi4aAvZVKik51Lfmg2Ec4ntD0oio5r3iWU+uUznaY5DpDl99c9aRM
hatCyAjKED8FK2UKxzPpUb4NAqbTC4zqV8GoSLIthdk8LpcYWQwZalHj93rdgP2E
XMSPJzOwQZtN/pxH+V9kctGjB03EMduQ4itnjryOORIHJBnDTQ9Vu1PUX0KpE5d8
Md4vyIhJAoGBANv32KW8jCq3m//lmkf1Z6WV5vAqXbrFpvIbt83AsLxnH17LJEvM
/pI5X6Lqaydt37ngPoOr5/z/Rrrm7R2Nds7CQvYVZXV3dLAW8yIPM+DMnHULwE+5
0//eOUVDYGc2IL98XB3pmmOveayImnH2IK6/0r8bP/GctzANYSCUAN8bAoGBAMAy
1sdTcVHqMRIYok1oWHqhxdrIL0vMoShh8sRzWP8sTOFhGAIVc2GyX1eIaGVJvnJb
PgLvvbb3qSTqvHw+5UM8EA5J+VlqFZqOSMsxegHhRaVLG2ARzjJGI1eo9pNEHndD
+x8/15q3TJ303wygt+OMwzav9TJ6ft4iPyr/s0CtAoGAKy/O/kM1ddGUtDoc/1HE
JrN2ouJ+goP50gD1JlVVnbfVZeQWXTeDJ6xSBYdmwFgHG9RWEb33jymDdoCOP0Yd
9FwZj30F7XkllMj8tUws0E77zNMgz8ZZRBwaPGoNDFaJOW8uGsVZhYbXKbOsNwGu
ywBoXRL5i/NL2AhYcG/+d88CgYAzeWosWByxY5dg0VgmgxuvPERRI+wG0PcYaaMr
Gflr3eF5OxfC9xu/vpXqS44qEmX/GpCxGKQmrI2QZY21eOLN7vgd25H8cHz4t8Kn
KMkUV9DfRA5A3IPs0vwELixbpJLEPmCzLdYAibbrVvml2CVnu/tMWjPBIS15MJLa
YUEQwQKBgQDILukmq3F9PbQb0Eho1mTbUZFzMGjZOLXHgkqKEaHTkMWVwGZZmLo7
5IXISjo5Ae9DH4ukYB/0efXXcBjtN3pJtwg9GVp5bvUaZQVqMQPziC9of3blVRqW
73RoJvRiNYg3l55VaVSHHxbHo9kewuWwKKPYrI8XeNoipMp5LU185Q==
-----END RSA PRIVATE KEY-----
[root@node-1.s013895.slurm.io /home/s013895]#
```


Скопируйте значение ключа куда-нибудь, мы его будем использовать внутри Gitlab.

Внутри самого Gitlab в той же секции CI/CD мы можем добавить наши переменные



Здесь уже есть небольшой спойлер к тому что произойдет дальше: мы будем добавлять наш ключ в специальную переменную.

Из интересного здесь — переменная может иметь тип variable, тогда при обращении в нее из пайплайна мы получим ее значение, либо FILE — тогда мы получим ее путь. К сожалению, более подходящий здесь FILE мы использовать не можем, по причине проверки ssh клиентом пермишенов ключей. По умолчанию

они могут читаться другими пользователями, и это слишком небезопасно по мнению Ansible. Поэтому мы пойдем другим путем.

Ставим variable, сохраняем, и меняем наш пайплайн.

```
stages:
  - provision


ansible_run:
  stage: provision
  tags:
    - test
  script:
    - echo "${PRIVATE_KEY}" > "$(pwd)/private" # сохраняем ключ в файл private
    - chmod -v 700 $(pwd)/private # ставим нужные пермишены
    - ansible-playbook playbook.yml -i hosts.ini --private-key $(pwd)/private #
подсовываем ключ в ansible
```

И наслаждаемся результатом!

```
1  Running with gitlab-runner 14.1.0 (8925d9a0)
2    on node-1.s013895.slurm.io kz-sHk4H
3  Preparing the "shell" executor                                    00:00
4  Using Shell executor...
6  Preparing environment                                             00:00
7  Running on node-1.s013895.slurm.io...
9  Getting source from Git repository                                00:01
10 Fetching changes with git depth set to 50...
11 Reinitialized existing Git repository in /home/gitlab-runner/builds/kz-sHk4H/0/s013895/ssh_runner_test/.git/
12 Checking out 89327cf8 as master...
13 Removing private
14 Skipping Git submodules setup
16 Executing "step_script" stage of the job script                   00:01
17 $ echo "${PRIVATE_KEY}" > "$(pwd)/private"
18 $ chmod -v 700 $(pwd)/private
19 mode of '/home/gitlab-runner/builds/kz-sHk4H/0/s013895/ssh_runner_test/private' changed from 0664 (rw-rw-r--) to 0700 (rwx------)
20 $ ansible-playbook playbook.yml -i hosts.ini --private-key $(pwd)/private
21 PLAY [Ping hosts] *************************************************************
22 TASK [Ping] *******************************************************************
23 ok: [testnode]
24 PLAY RECAP ********************************************************************
25 testnode                   : ok=1    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
27 Cleaning up file based variables                                  00:00
29 Job succeeded
```

Поздравляю! Вы восхитительны!

Послесловие

Само собой, это базовый пример, если вам нужно управлять кластером машин и вы хотите использовать несколько ключей, то будет посложнее, поскольку придется создавать несколько переменных.

Есть два других способа, которыми можно пользоваться:

1. Хранить приватные ключи в репо;
2. Использовать https://keepass.info/ и делать авторизацию по паролям. Ansible имеет специальный плагин для работы с keepass. А также можно изготовить свой модуль для работы с плагином самого

keepass, https://community.chocolatey.org/packages/keepass-plugin-keeagent/0.6.3 и получать ключи оттуда, все дороги открыты здесь :)

Спасибо за внимание!

7.5

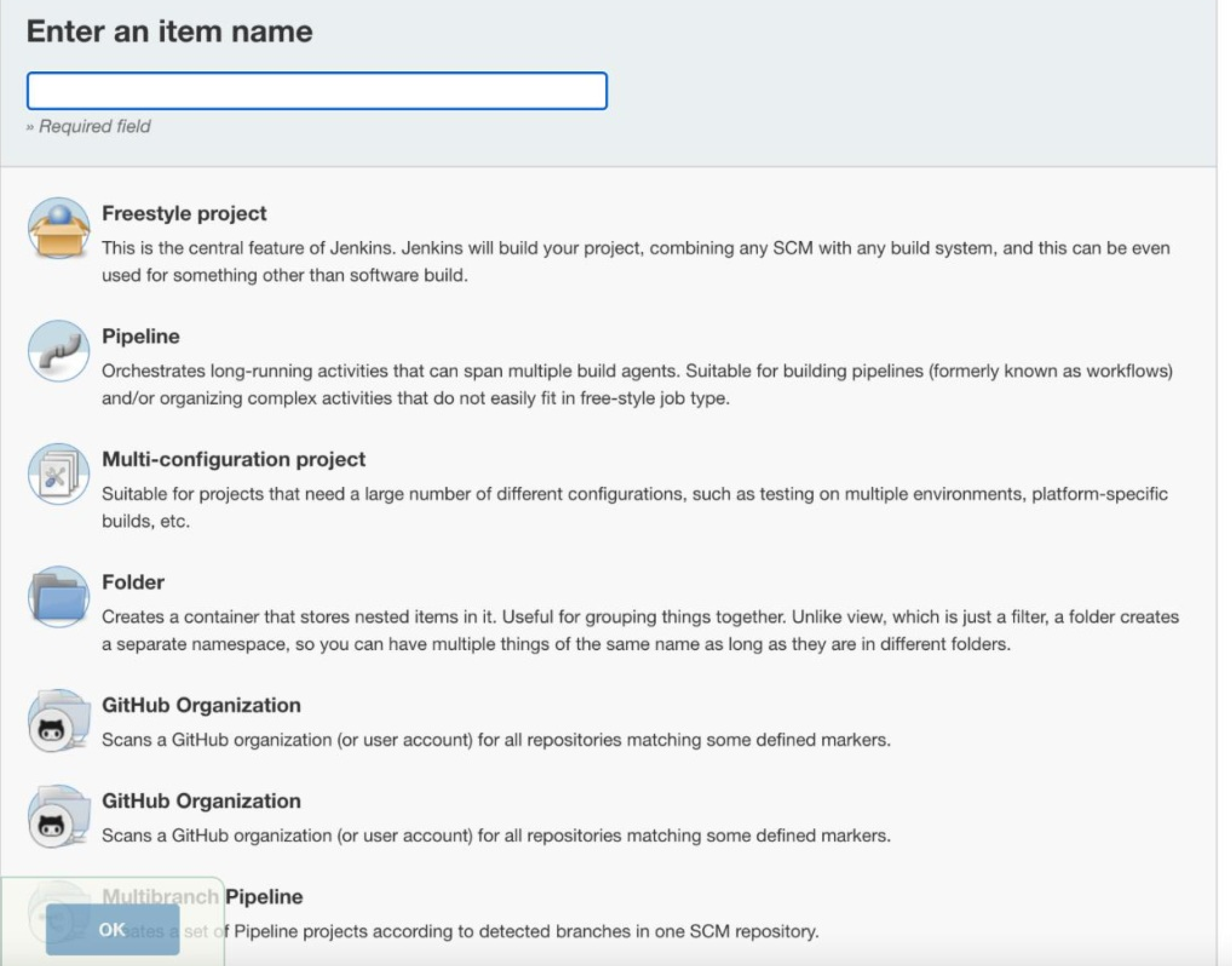
Данная статья для ознакомления и не является практикой, которую нужно выполнять.

Цель данной статьи - показать как настраивается Jenkins в связке с Ansible.
Предположим что у нас есть система с установленным Jenkins и мы хотели привязать IaaC в виде Ansible к ней.
После изначальной установки Jenkins нам необходимо настроить связку с Ansible.

Для дальнейшей работы необходимо договориться об условиях:

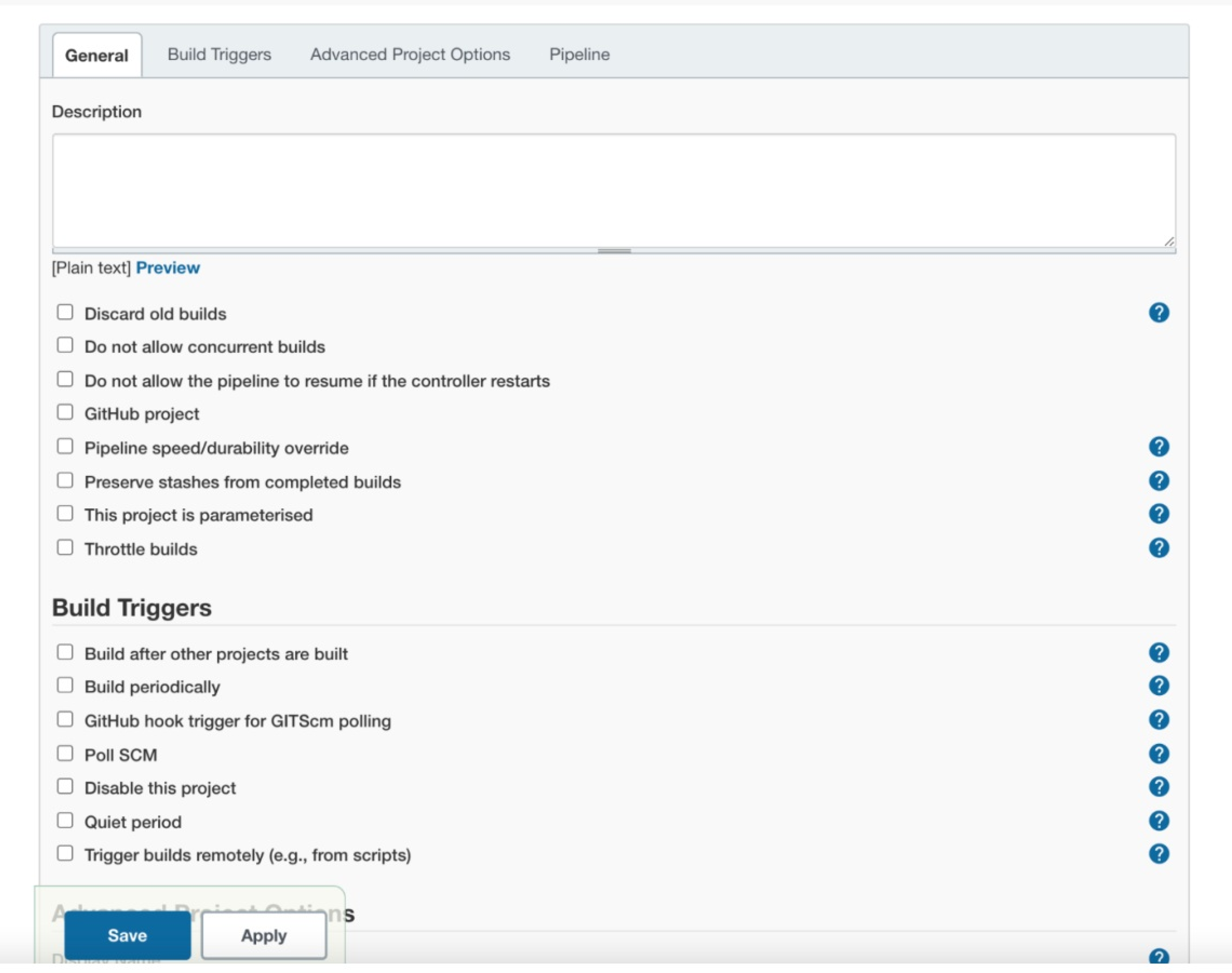
1. Jenkins установлен со стандартными плагинами.
2. Запуск всех пайплайнов будет происходить на той же машине где установлен Jenkins (стандартный build agent), но для remote агентов будет то же самое.

После установки Jenkins, вы можете создать новую build-работу.



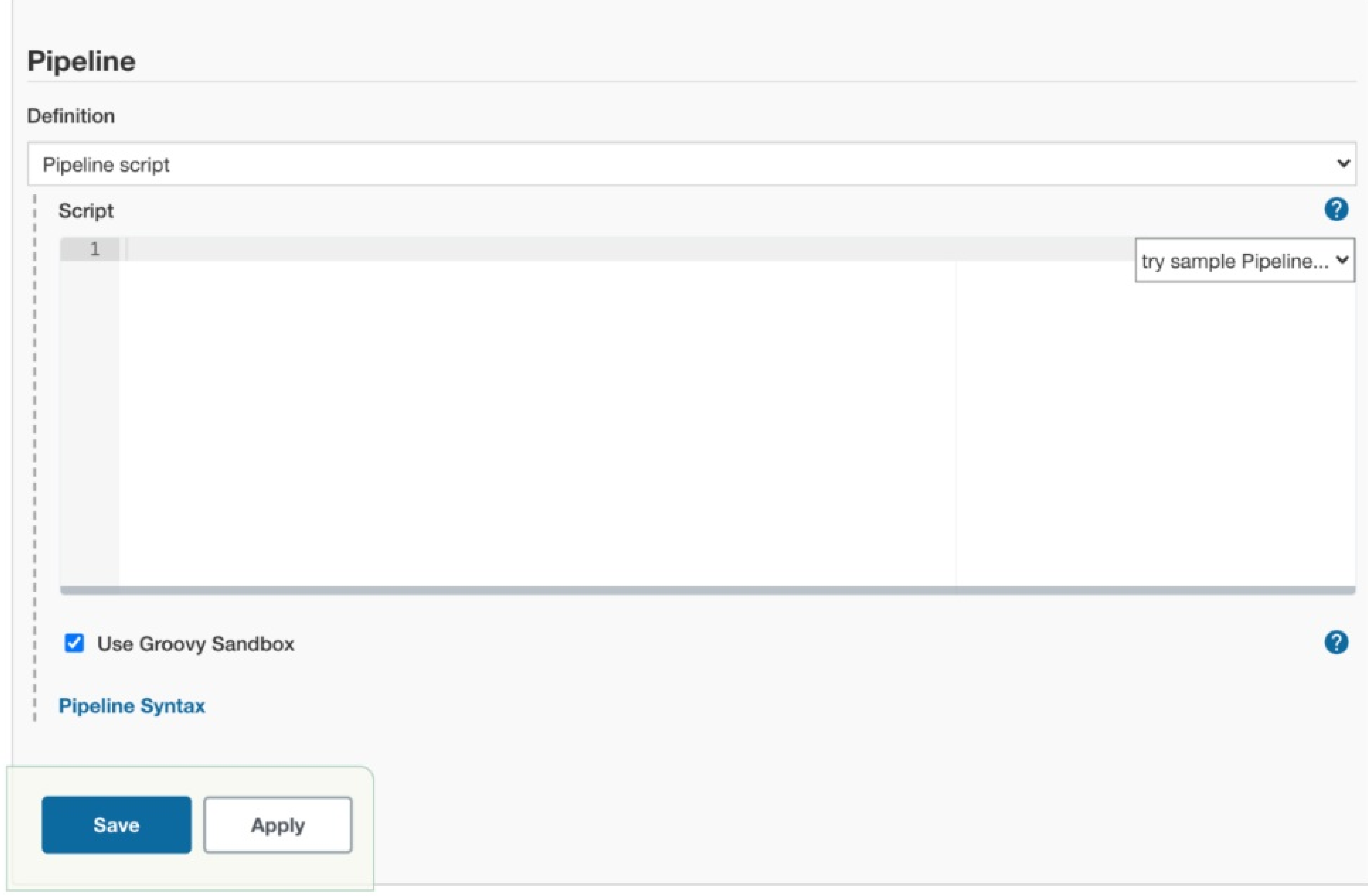
Вы можете создать Freestyle project - который является шаблоном для Jenkins, включающим в себя этапы “Спулить код” -> “Запустить работу”. Они позволяют вам выбрать готовые пайплайны, пользуясь шаблонами. Эта опция была добавлена в Jenkins для повышения дружелюбности интерфейса к начинающим администраторам и автоматизаторам. Но мы не такие, поэтому мы выберем Pipeline.

Окно со свойствами проекта.



Здесь можно задать описание проекта, настроить как будут вести себя пайплайны. Во вкладке Build Triggers можно поставить параметры для автоматического запуска пайплайна, например запускать раз в день, или запускать если появился новый коммит в определенной ветке Git.

Обратите внимание на поле ниже - Pipeline.



Jenkins как и Gitlab использует скрипт для запуска пайплайна по шагам, который умеет выполнять различные консольные команды. В отличие от Gitlab Jenkins использует Groovy - полноценный язык программирования, который дает возможности более тонкой кастомизации работ и позволяет полноценно использовать программные конструкции для вашей автоматизации, однако более сложен в освоении. Для нашей сегодняшней задачи это не столь важно, так как сегодня мы не будем лезть в дебри Groovy.

Итак, давайте, используя Pipeline достигнем следующих целей.

1. Запустим Ansible и выведем его версию
2. Скачаем Playbook с Github и запустим его
3. Постучимся в другую машину изнутри Jenkins
4. Используем специальный плагин Ansible для всего вышеуказанного

Запустим Ansible и выведем его версию:

Начнем с очень простой задачи - запустим Ansible в пайплайне и выведем его версию в консоль билда.

Пайплайны в Jenkins задаются очень простым путем - пишется Stage (аналогично стейджу в Gitlab), а внутри него перечисляются Steps. Именно это я имел в виду,

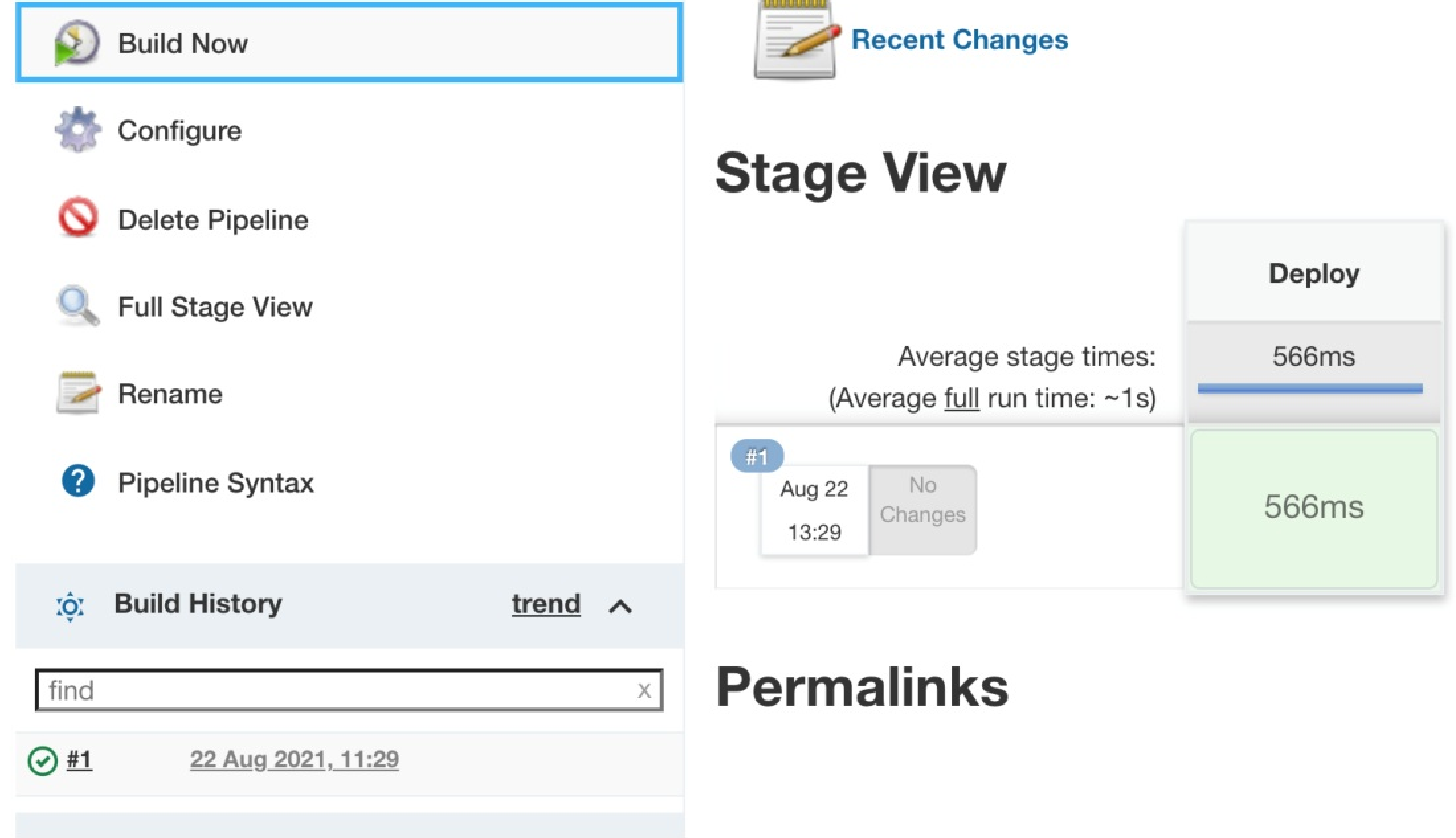
когда говорил что все системы CI/CD основаны на одном принципе, дьявол тут именно в деталях.

Давайте напишем свой Stage, назовем его Deploy и вызовем там версию Ansible.

```
pipeline { // задаем тон groovy, даем понять что здесь у нас шаги пайплайна
    agent any // если у нас есть какие то jenkins агенты специально для Ansible мы
можем их указать здесь, у меня только один агент, поэтому я поставлю any

    stages { // здесь определяем какие шаги в пайплайне
        stage('Deploy') { // наш деплой
            steps { // определяем шаги уже самого стейджа
                sh 'ansible --version' // выводим версию Ansible, команда sh просто
выполнит скрипт из консоли
            }
        }
    }
}
```

Сохраняем наш пайплайн, выходим в главное окно проекта и нажимаем Build Now.

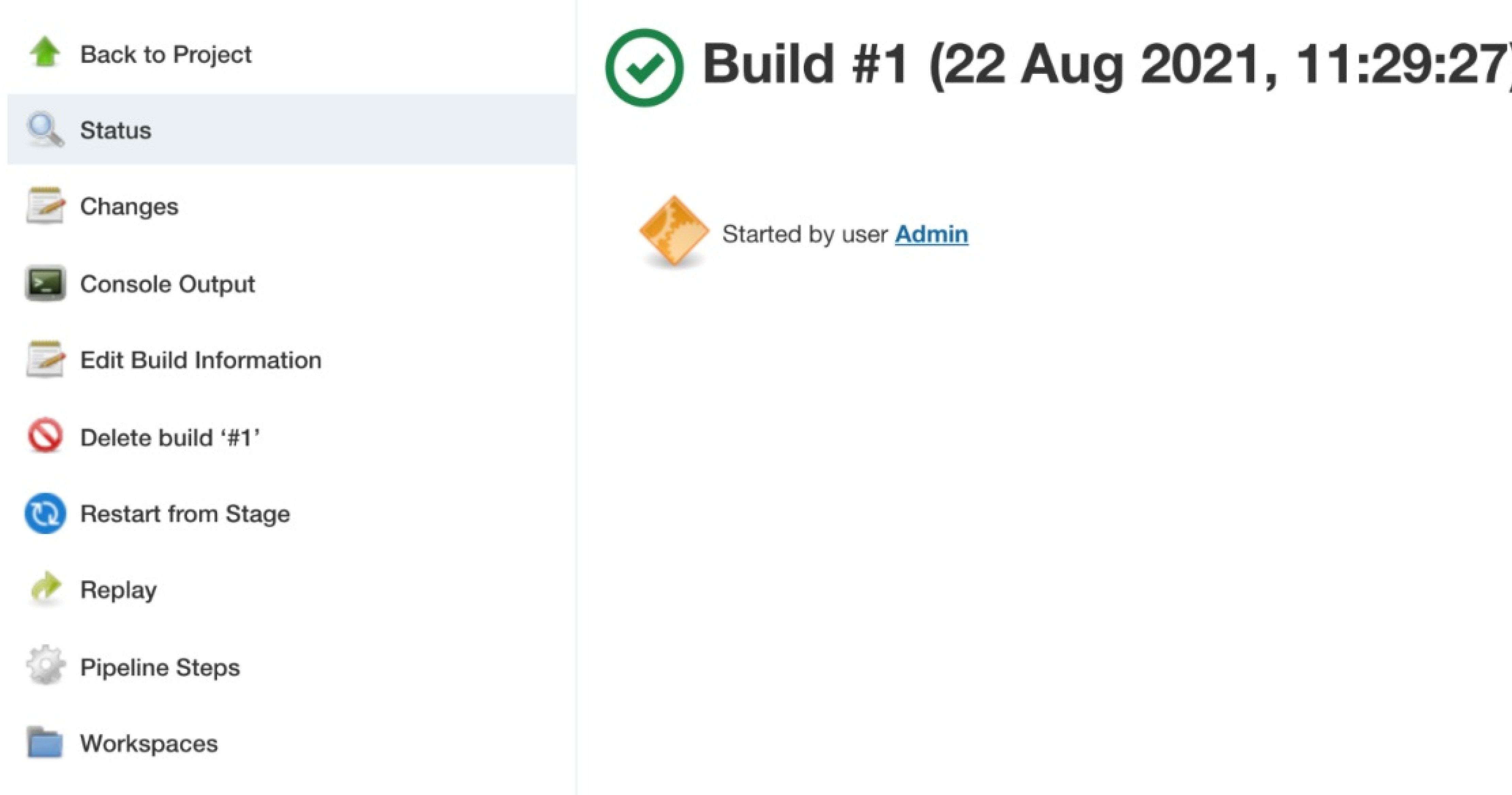
Если у вас на машине уже установлен Ansible то вы получите успех! Наш первый пайплайн абсолютно успешен! Ура!



394783

Давайте заглянем ему под капот.

Нажмем на #1 внизу в Build history. Каждый пайплайн создает history, которую вы можете ограничить настройками билда, чтобы предотвратить засорение диска логами и артефактами сборки. По умолчанию мы ничего не удаляем, я рекомендую на период настройки пайплайнов не удалять билды, чтобы вернуться на любой шаг и посмотреть необходимую информацию.



Жмем на Console Output чтобы посмотреть вывод.

Console Output

```
Started by user Admin
Running in Durability level: MAX_SURVIVABILITY
[Pipeline] Start of Pipeline
[Pipeline] node
Running on Jenkins in /var/lib/jenkins/workspace/pipelineRun
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Deploy)
[Pipeline] sh
+ ansible --version
ansible 2.9.6
  config file = /etc/ansible/ansible.cfg
  configured module search path = ['/var/lib/jenkins/.ansible/plugins/modules', '/usr/share/ansible/plugins/modules']
  ansible python module location = /usr/lib/python3/dist-packages/ansible
  executable location = /usr/bin/ansible
  python version = 3.8.10 (default, Jun  2 2021, 10:49:15) [GCC 9.4.0]
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

Видим пользователя, активировавшего билд, и настройки его ограничений. Внутри Jenkins (по умолчанию) вы очень ограничены во взаимодействии с системой, поскольку вы можете убить ноду с Jenkins неправильным пайплайном.
Далее вы видите шаги пайплайна подсвеченные серым, а вывод консоли черным цветом. После отображения папки, где был запущен пайплайн, выводятся результаты работы нашей sh команды, которая покажет нам версию Ansible в привычном виде, как если бы мы запустили ее из консоли.

Ура! Мы завершили первый шаг!!
Время скачать плейбук из GitHub и запустить его.

Запуск Playbook с Github

Скачаем Playbook с Github и запустим его.

Для изначального тестирования я создам очень простой плейбук.

```
---
    - name: "Test playbook run"
    hosts: localhost
    tasks:
      - name: "Debug"
        ansible.builtin.debug:
          msg: "Test debug"
```

Он располагается в github по адресу https://github.com/Nortsx/jenkinsansiblebook.

Пожалуйста! Перед началом работы, склонируйте его в свой репозиторий, таким образом вы не зависите от оригинала.

Давайте модифицируем наш пайплайн.

```
pipeline {
    agent any

    stages {
        stage('Checkout') { // добавим новый Stage
            steps {
                git branch: 'main', url:
"git@github.com:Nortsx/jenkinsansiblebook.git" // используем встроенный в Jenkins
плагин Git для скачивания проекта из бранча main
            }
        }
        stage('Deploy') {
            steps {
                sh 'ansible-playbook playbook.yml' // поскольку скрипты работают в
одной папке, на втором шаге мы можем просто запустить плейбук
            }
        }
    }
}
```

Я скачиваю плейбук его через ssh, а не http.

Запускаем Pipeline и смотрим на результаты билда.

```
[Pipeline] git
The recommended git tool is: NONE
Workspaces edentials specified
 > git rev-parse --resolve-git-dir /var/lib/jenkins/workspace/AnsibleTest/.git # timeout=10
Fetching changes from the remote Git repository
 > git config remote.origin.url git@github.com:Nortsx/jenkinsansiblebook.git # timeout=10
Fetching upstream changes from git@github.com:Nortsx/jenkinsansiblebook.git
 > git --version # timeout=10
 > git --version # 'git version 2.25.1'
 > git fetch --tags --force --progress -- git@github.com:Nortsx/jenkinsansiblebook.git +refs/heads/*:refs/remotes/origin/* # timeout=10
ERROR: Error fetching remote repo 'origin'
hudson.plugins.git.GitException: Failed to fetch from git@github.com:Nortsx/jenkinsansiblebook.git
        at hudson.plugins.git.GitSCM.fetchFrom(GitSCM.java:1004)
        at hudson.plugins.git.GitSCM.retrieveChanges(GitSCM.java:1245)
        at hudson.plugins.git.GitSCM.checkout(GitSCM.java:1305)
        at org.jenkinsci.plugins.workflow.steps.scm.SCMStep.checkout(SCMStep.java:129)
        at org.jenkinsci.plugins.workflow.steps.scm.SCMStep$StepExecutionImpl.run(SCMStep.java:97)
        at org.jenkinsci.plugins.workflow.steps.scm.SCMStep$StepExecutionImpl.run(SCMStep.java:84)
        at org.jenkinsci.plugins.workflow.steps.SynchronousNonBlockingStepExecution.lambda$start$0(SynchronousNonBlockingStepExecution.java:47)
        at java.base/java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:515)
        at java.base/java.util.concurrent.FutureTask.run(FutureTask.java:264)
        at java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1128)
        at java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:628)
        at java.base/java.lang.Thread.run(Thread.java:829)
Caused by: hudson.plugins.git.GitException: Command "git fetch --tags --force --progress -- git@github.com:Nortsx/jenkinsansiblebook.git +refs/heads/*:refs/remotes/origin/*"
returned status code 128:
stdout:
stderr: Host key verification failed.
fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.

        at org.jenkinsci.plugins.gitclient.CliGitAPIImpl.launchCommandIn(CliGitAPIImpl.java:2681)
        at org.jenkinsci.plugins.gitclient.CliGitAPIImpl.launchCommandWithCredentials(CliGitAPIImpl.java:2102)
        at org.jenkinsci.plugins.gitclient.CliGitAPIImpl.access$500(CliGitAPIImpl.java:86)
        at org.jenkinsci.plugins.gitclient.CliGitAPIImpl$1.execute(CliGitAPIImpl.java:624)
        at hudson.plugins.git.GitSCM.fetchFrom(GitSCM.java:1002)
        ... 11 more
[Pipeline] }
[Pipeline] // stage
```

Jenkins провалился на первом этапе, и не смог скачать исходники с git, по причине невозможности установить соединение с github. Ответ на вопрос “как это пофиксить?”, кроется в тексте ошибки. Мы увидим эту ошибку, если соединимся по ssh, используя локальный ssh-agent, но при этом данного сервера нет в known_hosts. Давайте добавим сервер github для jenkins.

Логинимся на нашу машину с Jenkins, от лица пользователя Jenkins (если вы не переставляли пользователя для Jenkins) запускаем ssh github.com. На вопрос “Do you want to add gihutb.com to known hosts” жмем Y. Все, первая ошибка больше не будет нам докучать.

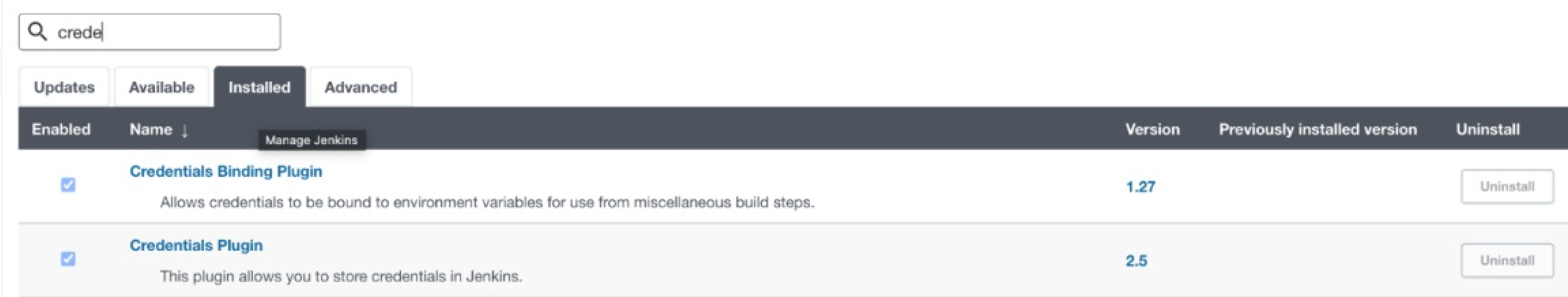
Запускаем пайплайн второй раз.

```
The recommended git tool is: NONE
No credentials specified
 > git rev-parse --resolve-git-dir /var/lib/jenkins/workspace/AnsibleTest/.git # timeout=10
Fetching changes from the remote Git repository
 > git config remote.origin.url git@github.com:Nortsx/jenkinsansiblebook.git # timeout=10
Fetching upstream changes from git@github.com:Nortsx/jenkinsansiblebook.git
 > git --version # timeout=10
 > git --version # 'git version 2.25.1'
 > git fetch --tags --force --progress -- git@github.com:Nortsx/jenkinsansiblebook.git +refs/heads/*:refs/remotes/origin/* # timeout=10
ERROR: Error fetching remote repo 'origin'
hudson.plugins.git.GitException: Failed to fetch from git@github.com:Nortsx/jenkinsansiblebook.git
        at hudson.plugins.git.GitSCM.fetchFrom(GitSCM.java:1004)
        at hudson.plugins.git.GitSCM.retrieveChanges(GitSCM.java:1245)
        at hudson.plugins.git.GitSCM.checkout(GitSCM.java:1305)
        at org.jenkinsci.plugins.workflow.steps.scm.SCMStep.checkout(SCMStep.java:129)
        at org.jenkinsci.plugins.workflow.steps.scm.SCMStep$StepExecutionImpl.run(SCMStep.java:97)
        at org.jenkinsci.plugins.workflow.steps.scm.SCMStep$StepExecutionImpl.run(SCMStep.java:84)
        at org.jenkinsci.plugins.workflow.steps.SynchronousNonBlockingStepExecution.lambda$start$0(SynchronousNonBlockingStepExecution.java:47)
        at java.base/java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:515)
        at java.base/java.util.concurrent.FutureTask.run(FutureTask.java:264)
        at java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1128)
        at java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:628)
        at java.base/java.lang.Thread.run(Thread.java:829)
Caused by: hudson.plugins.git.GitException: Command "git fetch --tags --force --progress -- git@github.com:Nortsx/jenkinsansiblebook.git +refs/heads/*:refs/remotes/origin/*"
returned status code 128:
stdout:
stderr: git@github.com: Permission denied (publickey).
fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.

        at org.jenkinsci.plugins.gitclient.CliGitAPIImpl.launchCommandIn(CliGitAPIImpl.java:2681)
        at org.jenkinsci.plugins.gitclient.CliGitAPIImpl.launchCommandWithCredentials(CliGitAPIImpl.java:2102)
        at org.jenkinsci.plugins.gitclient.CliGitAPIImpl.access$500(CliGitAPIImpl.java:86)
        at org.jenkinsci.plugins.gitclient.CliGitAPIImpl$1.execute(CliGitAPIImpl.java:624)
        at hudson.plugins.git.GitSCM.fetchFrom(GitSCM.java:1002)
        ... 11 more
```

Обратите внимание,ошибка в stderr поменялась на Permission Denied - для подключения по ssh к github нам нужен ключ, который github знает.

Давайте добавим этот ключ.

Убеждаемся, что плагин Credentials установлен (Dashboard -> Manage Jenkins -> Manage Plugins), если нет, то установим его.



Генерируем ключ для Github и добавляем его в аккаунт.

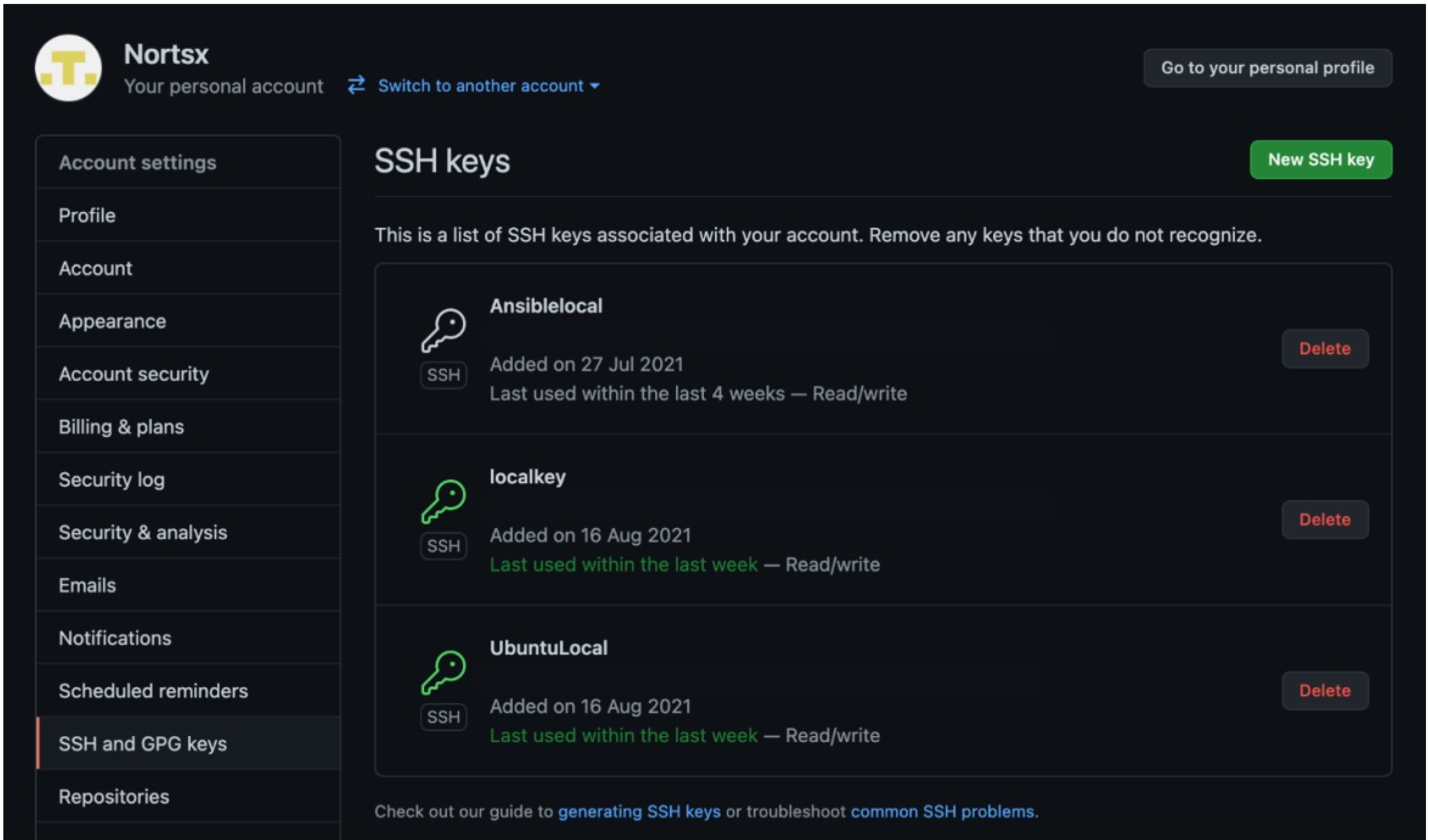
Запустим команду на машине с linux

Ssh-keygen -t rsa.

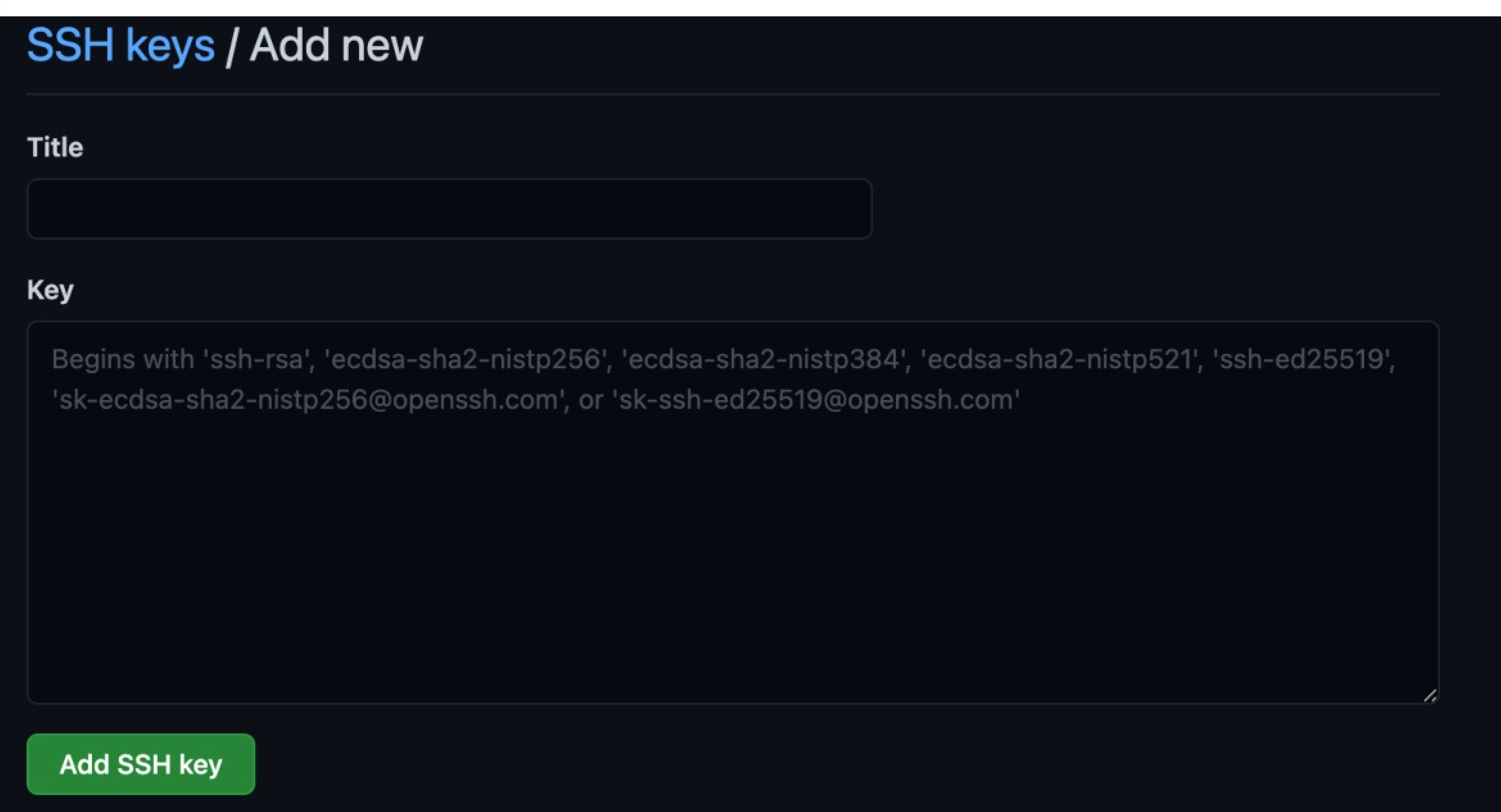
После ответа на вопросы об имени ключа (укажите какое либо другое имя кроме id_rsa если ваш стандартный ключ уже используется, поскольку его можно случайно перезаписать), защита паролем (выбираем “**no**”) вы получите пару приватный/публичный ключ. Публичный ключ имеет расширение .pub после имени, приватный ключ его **не** имеет.

Важно! Старайтесь не светить свой приватный ключ, особенно если он используется где то для доступа к защищенным репозиториям, это вопрос безопасности!

После генерации ключей идем на github.com, кликаем на иконку профиля справа, Settings->SSH and GPG keys и попадаем на страницу добавления ssh ключей профиля.



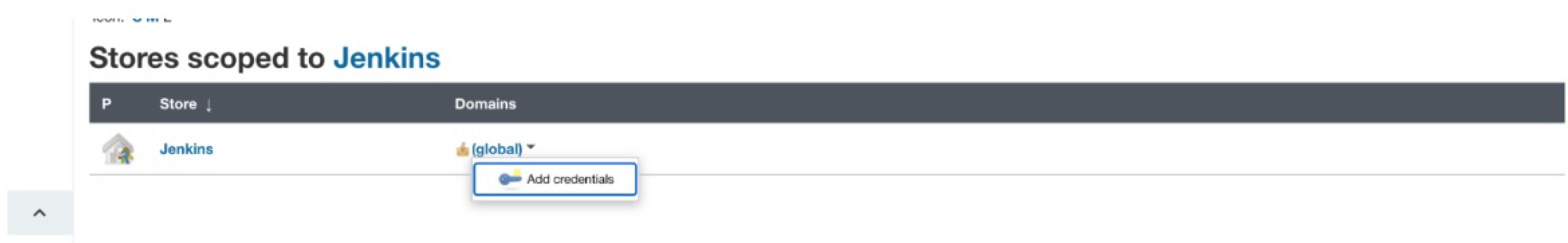
Жмякаем New SSH Key



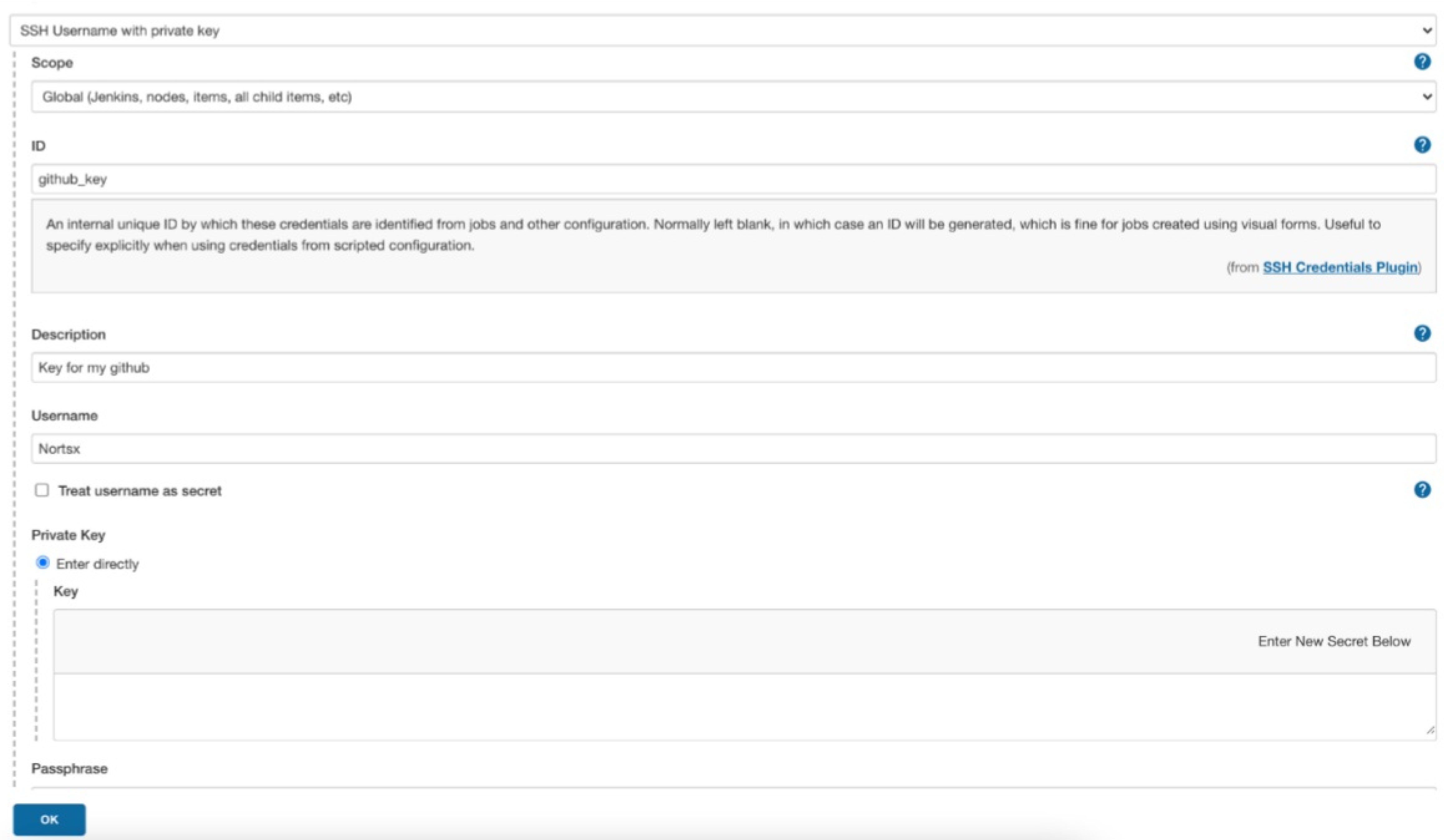
Копируем **ПУБЛИЧНУЮ** часть ключа и называем его произвольным именем

Готово!

В Dashboard -> Manage Jenkins -> Credentials нажимаем на Global и Add Credentials.



В открывшемся окне выбираем SSH Username with private key и заполняем данные.



ID -> идентификатор, уникальный для каждого credential, по которому вы можете их вызывать из пайплайна.

Username - имя для вашего профиля Github.

Private key -> нажмите Enter Directly -> Add и скопируйте значение из сгенерированного **Приватного** ключа.

После добавления мы увидим наш ключ в списке Credentials.

	ID	Name	Kind	Description	
	github_key	Nortsx (Key for my github)	SSH Username with private key	Key for my github	

Время использовать ключик в пайплайне!

```
pipeline {
    agent any

    stages {
```

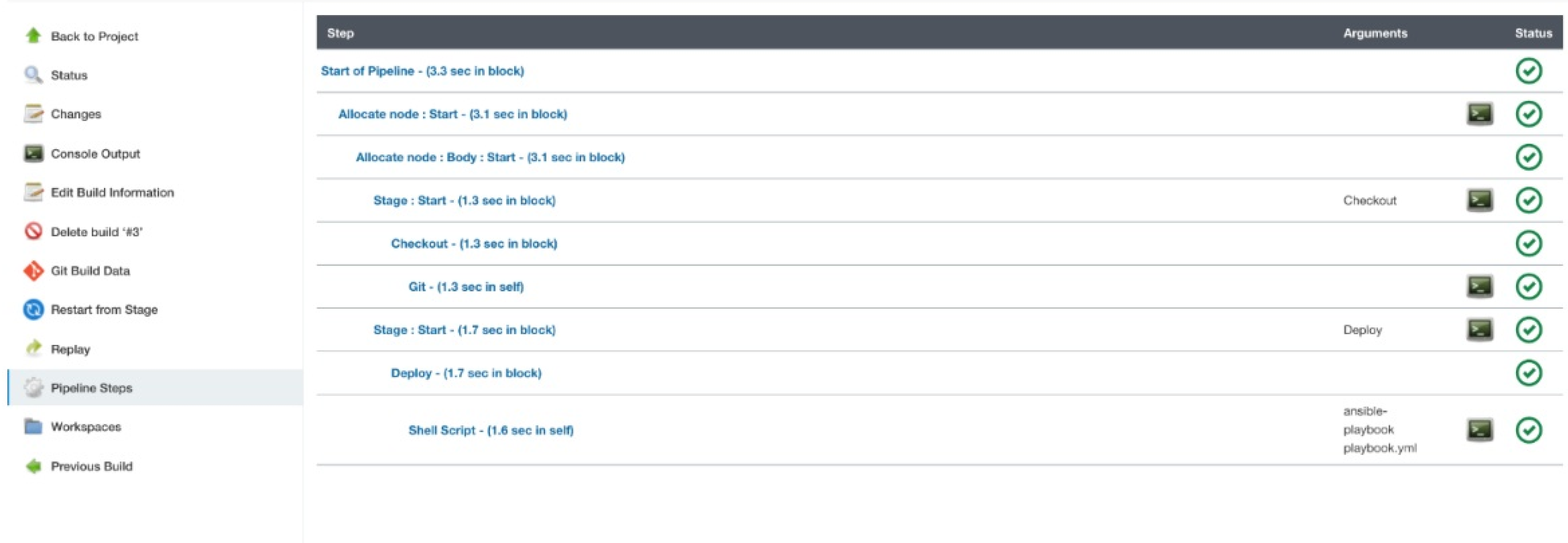
```
        stage('Checkout') {
            steps {
                git branch: 'main', url:
"git@github.com:Nortsx/jenkinsansiblebook.git", credentialsId: 'github_key' // здесь
добавляем credentialsId и указываем наш ID.
            }
        }
        stage('Deploy') {
            steps {
                sh 'ansible-playbook playbook.yml'
            }
        }
    }
}
```

Запускаем пайплайн и смотрим на вывод.

```
[Pipeline] git
The recommended git tool is: NONE
using credential github_key
 > git rev-parse --resolve-git-dir /var/lib/jenkins/workspace/pipelineRun/.git # timeout=10
Fetching changes from the remote Git repository
 > git config remote.origin.url git@github.com:Nortsx/jenkinsansiblebook.git # timeout=10
Fetching upstream changes from git@github.com:Nortsx/jenkinsansiblebook.git
 > git --version # timeout=10
 > git --version # 'git version 2.25.1'
using GIT_SSH to set credentials Key for my github
 > git fetch --tags --force --progress -- git@github.com:Nortsx/jenkinsansiblebook.git +refs/heads/*:refs/remotes/origin/* # timeout=10
 > git rev-parse refs/remotes/origin/main^{commit} # timeout=10
Checking out Revision 5860a0381a7cb27b48bbe8ffca122d83a0b19a1f (refs/remotes/origin/main)
 > git config core.sparsecheckout # timeout=10
 > git checkout -f 5860a0381a7cb27b48bbe8ffca122d83a0b19a1f # timeout=10
 > git branch -a -v --no-abbrev # timeout=10
 > git checkout -b main 5860a0381a7cb27b48bbe8ffca122d83a0b19a1f # timeout=10
Commit message: "Initial commit"
First time build. Skipping changelog.
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Deploy)
[Pipeline] sh
+ ansible-playbook playbook.yml
[WARNING]: provided hosts list is empty, only localhost is available. Note that
the implicit localhost does not match 'all'

PLAY [Test playbook run] *******************************************************

TASK [Gathering Facts] *********************************************************
ok: [localhost]

TASK [Debug] *******************************************************************
ok: [localhost] => {
    "changed": false,
    "msg": "Test debug"
}

PLAY RECAP *********************************************************************
localhost                  : ok=2    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

Ура! Все сработало!

Кстати, посмотреть результаты запусков по шагам можно нажав кнопку Pipeline Steps.



Она покажет аргументы запуска и позволит изолированно посмотреть вывод по какому то конкретному шагу (по клику на мониторчике), что может быть удобно для отладки.

Например вывод только запуска нашего Ansible playbook.

Console Output

```
+ ansible-playbook playbook.yml
[WARNING]: provided hosts list is empty, only localhost is available. Note that
the implicit localhost does not match 'all'

PLAY [Test playbook run] *******************************************************

TASK [Gathering Facts] *********************************************************
ok: [localhost]

TASK [Debug] *******************************************************************
ok: [localhost] => {
    "changed": false,
    "msg": "Test debug"
}

PLAY RECAP *********************************************************************
localhost                  : ok=2    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

Переходим к следующей части.

Постучимся в другую машину изнутри Jenkins

Воспользуемся Ansible, чтобы постучаться в другие хосты.

В плейбук добавим файлы hosts и ansible.cfg

hosts.ini

```
testmachine ansible_host=192.168.60.55 ansible_user=vagrant
```

Я использую Vagrant, поэтому мои адреса принадлежат локальной сети.

ansible.cfg

```
[defaults]
host_key_checking=false
```

playbook.yml

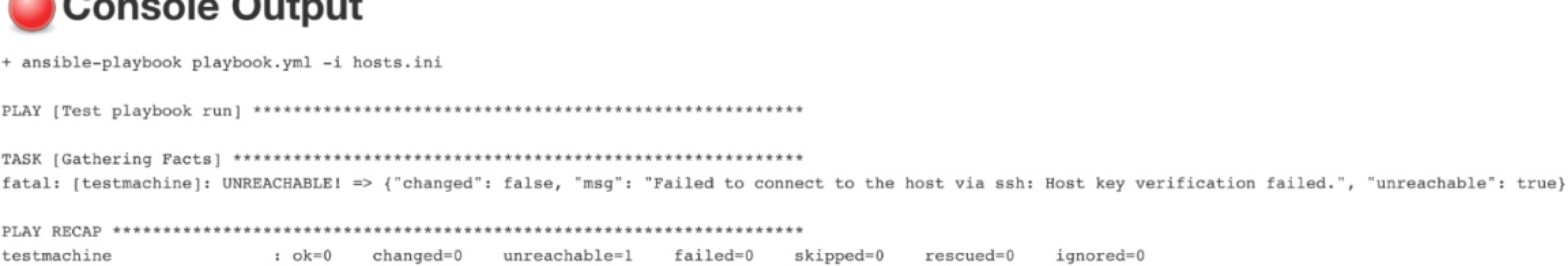
```
---
- name: "Test playbook run"
  hosts: all # теперь запускаем скрипты на удаленных хостах
  tasks:
    - name: "Debug"
      ansible.builtin.debug:
        msg: "Test debug"
```

После изменений в своем гите, нам необходимо изменить пайплайн, для использования файла hosts.ini.

```
pipeline {
    agent any

    stages {
        stage('Checkout') {
            steps {
                git branch: 'main', url:
"git@github.com:Nortsx/jenkinsansiblebook.git", credentialsId: 'github_key'
            }
        }
        stage('Deploy') {
            steps {
                withCredentials([sshUserPrivateKey(credentialsId: 'github_key',
keyFileVariable: 'PRIVATE')]) { // используем credentials и записываем их содержание
во временный файл доступный по пути PRIVATE
                   sh 'ansible-playbook playbook.yml -i hosts.ini --private-key
$PRIVATE' //используем переменную с помощью Ansible
                }
            }
        }
    }
}
```

После запуска видим провал на втором шаге.

	Checkout	Deploy
Average stage times: (Average full run time: ~3s)	1s	870ms
#5 Aug 22 15:07 No Changes	1s	911ms failed



Как можно видеть, ключи не авторизованы на удаленной машине, поэтому мы не можем связаться с ней.

Добавим ключи для авторизации на удаленной машине.

Я буду использовать тот же ключ, что для github. Возьмем **публичную** часть ключа и добавим в ~/.ssh/authorized_keys машины, к которой подключаемся через ansible.

Изменим пайплайн, чтобы Ansible мог использовать ключи авторизации.

```
pipeline {
    agent any

    stages {
        stage('Checkout') {
            steps {
                git branch: 'main', url:
"git@github.com:Nortsx/jenkinsansiblebook.git", credentialsId: 'github_key'
            }
        }
        stage('Deploy') {
            steps {
                withCredentials([sshUserPrivateKey(credentialsId: 'github_key',
keyFileVariable: 'PRIVATE')]) { // используем credentials и записываем их содержание
во временный файл доступный по пути PRIVATE
                    sh 'ansible-playbook playbook.yml -i hosts.ini --private-key
$PRIVATE' //используем переменную с помощью Ansible
                }
            }
        }
    }
}
```

И вуаля!

Console Output

```
+ ansible-playbook playbook.yml -i hosts.ini --private-key ****

PLAY [Test playbook run] *******************************************************

TASK [Gathering Facts] *********************************************************
ok: [testmachine]

TASK [Debug] *******************************************************************
ok: [testmachine] => {
    "changed": false,
    "msg": "Test debug"
}

PLAY RECAP *********************************************************************
testmachine                : ok=2    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

Для каких либо других переменных, которые не надо скрывать, мы можем использовать секцию variables.

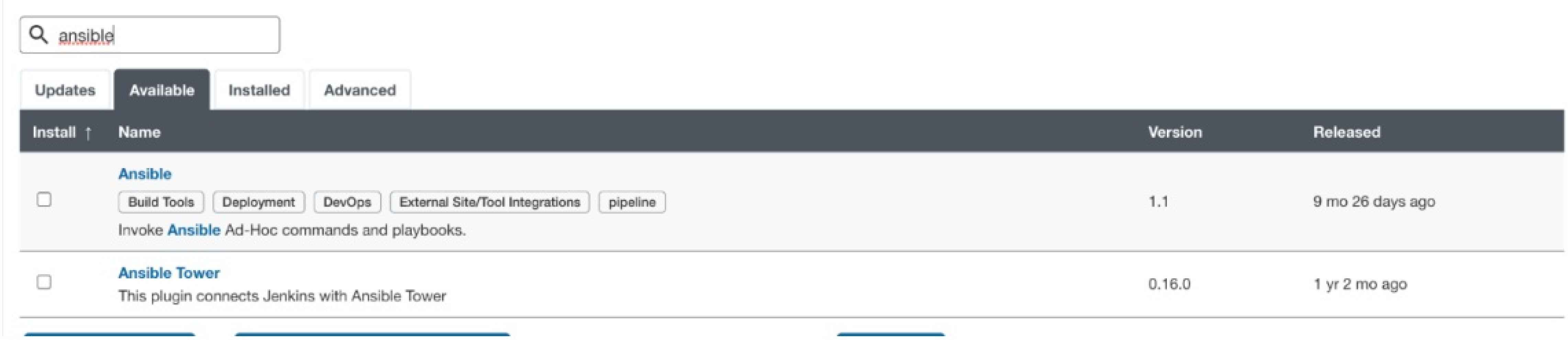
Например:

```
pipeline {
    agent any

    environment {
        GIT_BRANCH = ‘main’
    }

    stages {
        stage('Checkout') {
            steps {
                git branch: $GIT_BRANCH, url:
"git@github.com:Nortsx/jenkinsansiblebook.git", credentialsId: 'github_key'
            }
        }
```

... и так далее.

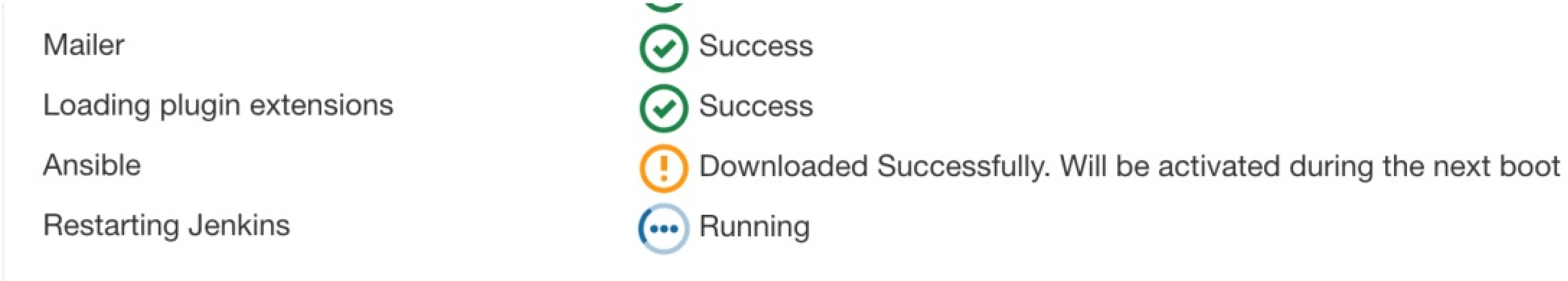
Используем специальный плагин Ansible для всего вышеуказанного

Снова идем в список плагинов в Manage Jenkins и ищем Ansible в списке Available плагинов.



Устанавливаем его.

При установке ставим галочку “Restart Jenkins” и после установки Jenkins перезагрузится.



Теперь, можно переписать наш пайплайн.

```
pipeline {
    agent any

    stages {
        stage('Checkout') {
            steps {
                git branch: 'main', url:
"git@github.com:Nortsx/jenkinsansiblebook.git", credentialsId: 'github_key'
            }
        }
        stage('Deploy') {
            steps {
                ansiblePlaybook playbook: 'playbook.yml', inventory: 'hosts.ini',
credentialsId: 'github_key'
            }
        }
    }
}
```

Как вы можете видеть, вместо оборачивания в credentials, мы передали строчку в Id и вызвали специальный модуль, делающий за нас всю работу. Это сравнимо с использованием, например command или shell в Ansible, когда можно использовать специальный модуль упрощающий и контролирующий работу, вместо “голых” вызовов консоли.

Плагин умеет много больше чем простой запуск плейбука, сильно упрощает пайплайны и саму жизнь. На этом у меня все, всем спасибо за внимание!

7.8

Создаем стенд

Текущий стенд состоит из 3 узлов: `node-1.sXXXXXX`, `node-2.sXXXXXX`, `node-3.sXXXXXX` (XXXXXX - ваш номер студента). Работа стенда 6 часов. 2 попытки запуска стенда.

Что уже установлено на стенде?

- node-1: ansible2.9, python2.7\3.6, molecule
- node-2: python2.7\3.6
- node-3: python2.7\3.6

Какой IP-адрес у них?

1 и 4 октет сети у всех одинаковый -**172.XX.XXX.6** (node-1), **172.XX.XXX.7** (node-2) и **172.XX.XXX.8** (node-3). 2 и 3 узнать через команду `ip -a`. К примеру это **20.247**

```
ip a | grep eth0

inet 172.20.247.6/24 brd 172.20.247.255 scope global eth0
```

1. Над текстом нажмите кнопку «Создать стенд». Каждый стенд создан под определенную тему курса. Сейчас вы находитесь в теме - **7. Использование Ansible в продакшене**, этот стенд не подойдет под практические занятия из других пунктов курса.

Запуск обычно идёт до 10 минут, в редких случаях до 30 минут.

2. После создания стенда авторизуйтесь по SSH на adminbox с адресом **sbox.slurm.io** с помощью логина и пароля, находящихся в настройках профиля или над текстом по кнопке «**Доступы»**.

3. Далее подключаетесь к первой ноде — контролнода. (XXXXXX - ваш номер студента)

```
ssh node-1.sXXXXXX
```

И повышаем себя до root пользователя.

```
sudo -i
```

Можете приступать к выполнению практических заданий. Желаем успешно выполнить их!

Метрики в продакшен

Перед нами стоит задача — установить базу данных PostgreSQL на виртуальную машину, а также настроить экспорт метрик для того, чтобы мы могли следить за её основными жизненными показателями. Плейбук должен делать healthcheck после каждой проверки. Наши задачи:

Установить базу данных PostgreSQL, накатить базовые миграции в базу migration (предварительно ее надо создать), создать пользователя testuser, с паролем.

Эти действия несложно сделать с помощью роли postgres, напоминаю лишь только, что необходимо правильно настроить файл pg_hba.conf, для того чтобы postgres адекватно реагировал на запросы из внешнего мира (0.0.0.0/0 для ipv4 и ::0/0 для ipv6). Также напоминаю что директива в postgresql.conf отвечает за прослушивание постгресом внешних подключений и по умолчанию установлена в localhost, то есть не реагирует на подключения извне. Ее надо будет поменять в * или в имя вашего хоста.

Миграции

```
USE migrations;

--
-- Name: transactions; Type: TABLE; Schema: public; Owner: mytester
--

CREATE TABLE public.transactions (
    id integer,
    "timestamp" integer,
    name text,
    size integer
);

ALTER TABLE public.transactions OWNER TO mytester;

--
-- Data for Name: transactions; Type: TABLE DATA; Schema: public; Owner: mytester
--

--
-- Name: SCHEMA public; Type: ACL; Schema: -; Owner: postgres
--

REVOKE ALL ON SCHEMA public FROM PUBLIC;
REVOKE ALL ON SCHEMA public FROM postgres;
GRANT ALL ON SCHEMA public TO postgres;
GRANT ALL ON SCHEMA public TO PUBLIC;

--
-- PostgreSQL database dump complete
```

```
--
```
https://github.com/cloudalchemy/ansible-prometheus

Для организации healthcheck нам необходимо с помощью delegate_to подключиться к postgres с нашей Ansible машины и убедиться, что порт открыт и работает. Можно выполнить запрос на, скажем, получение версии, и ещё один на получение данных из базы (скажем SELECT * FROM transactions).

Устанавливающий экспортер метрик из postgres на ту же самую машину (https://github.com/prometheus-community/postgres_exporter).

Эта роль устанавливает postgres exporter с помощью Ansible (https://github.com/ome/ansible-role-prometheus-postgres)

Экспортер метрик возьмет все метрики из postgres и выведет по-определенному URL в формате, понятном утилите сбора метрик — prometheus.

Для установки используйте роли, не надо писать все с нуля.

Здесь хорошим healthcheck будет попытка достучаться на порт (по умолчанию 9187) и забрать данные из метрик (например ваш_адрес:9187/metrics). Простой запрос на забор данных должен иметь строчку, например, pg_settings_port. Либо строчку pg_up 1.

При установке обратите внимание на exporter, который руководствуется переменными окружения, в частности DATA_SOURCE_NAME для установки DSN к БД. Рекомендую там использовать формат, по которому вы сами будете подключаться к БД, такой, как **postgresql://пользователь:пароль@хост:порт/базаданных**

Переменные окружения для машины можно установить с помощью раздела environment в **плее.**

Выглядит это следующим образом:

```
hosts: postgres

environment:
 DATA_SOURCE_NAME:
'postgresql://mytester:testerpass@localhost:5432/migrations?sslmode=disable'

vars:
 postgresql_hba_entries:

...здесь переменные…
```

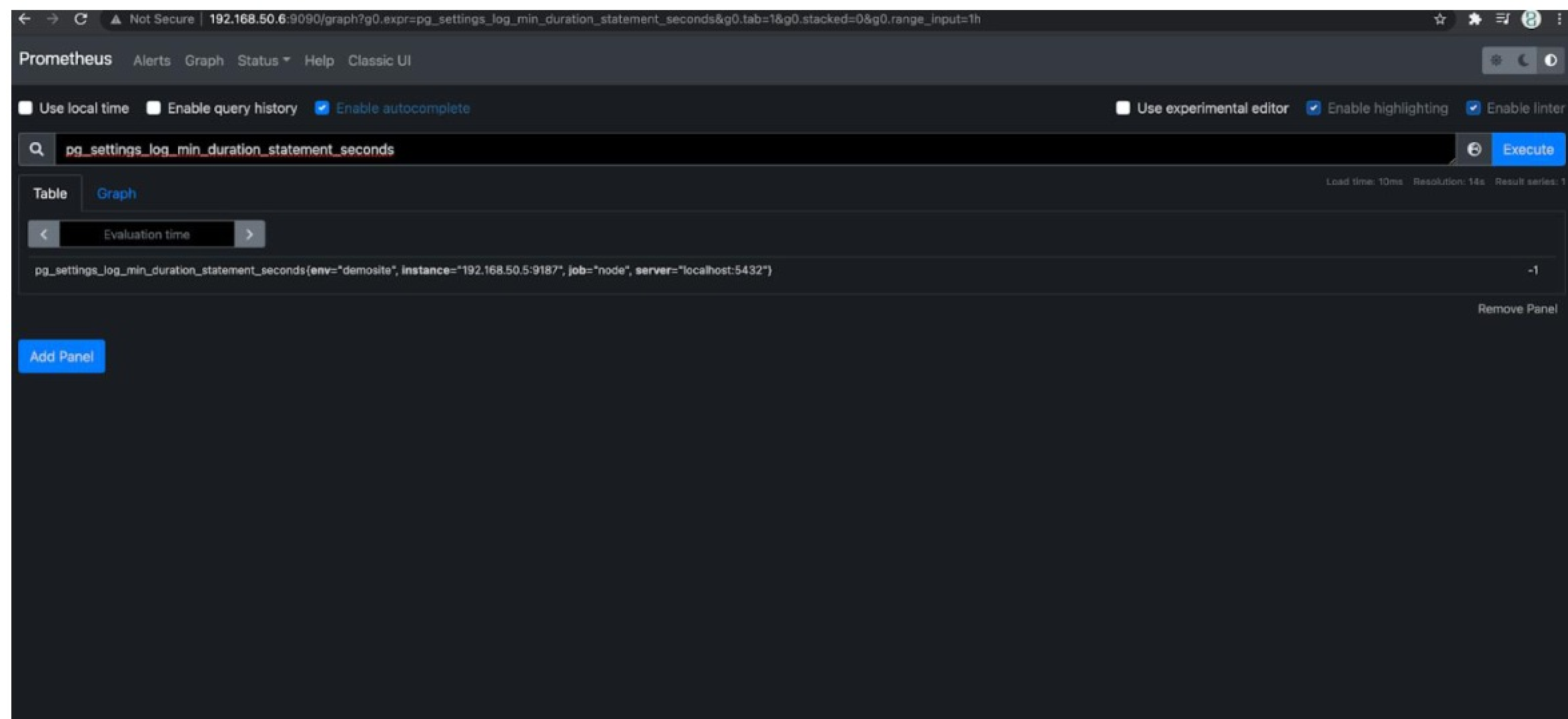
Плей, устанавливающий систему сборки метрик prometheus и настраивающий его на сборку логов из postgres explorer.

Prometheus работает следующим образом: он состоит из API с данными и из так называемых scrapers, которые с каким-то интервалом сами ходят и забирают данные. Интервалы скрейпинга и URL для забора данных должны настраиваться через ваш плейбук.

Также обратите внимание на описание targets в роли. Без них prometheus не будет знать, что забирать.

Prometheus следует устанавливать на node-3, который имеет IP-адрес **172.XX.XXX.8.** Именно на эту ноду настроено проксирование извне по адресу **prometheus.sXXXXXX.edu.slurm.io** (XXXXXX - ваш номер студента) на **:9090** порт. Открывайте приложение в режиме "инкогнито", т.к. в обычном режиме браузер блокирует доступ по http. В Google Chrome к примеру, можно запустить режим с помощью комбинации "Ctrl+Shift+N".

Хорошим хелсчеком будет вызвать prometheus и проверить данные. Можно это сделать руками, вызвав окно в браузере.



Скройте все переменные из плейбука, используя кодирование строк ansible-vault.

Целью данной практической работы является отработка навыка работы с ролями, а также демонстрация того, насколько можно ускорить свою работу используя уже готовые модули.

По желанию. Используя приемы из описанных в лекции (напр. mitogen), измерьте разницу в производительности. Насколько быстрее теперь запускается плейбук?