

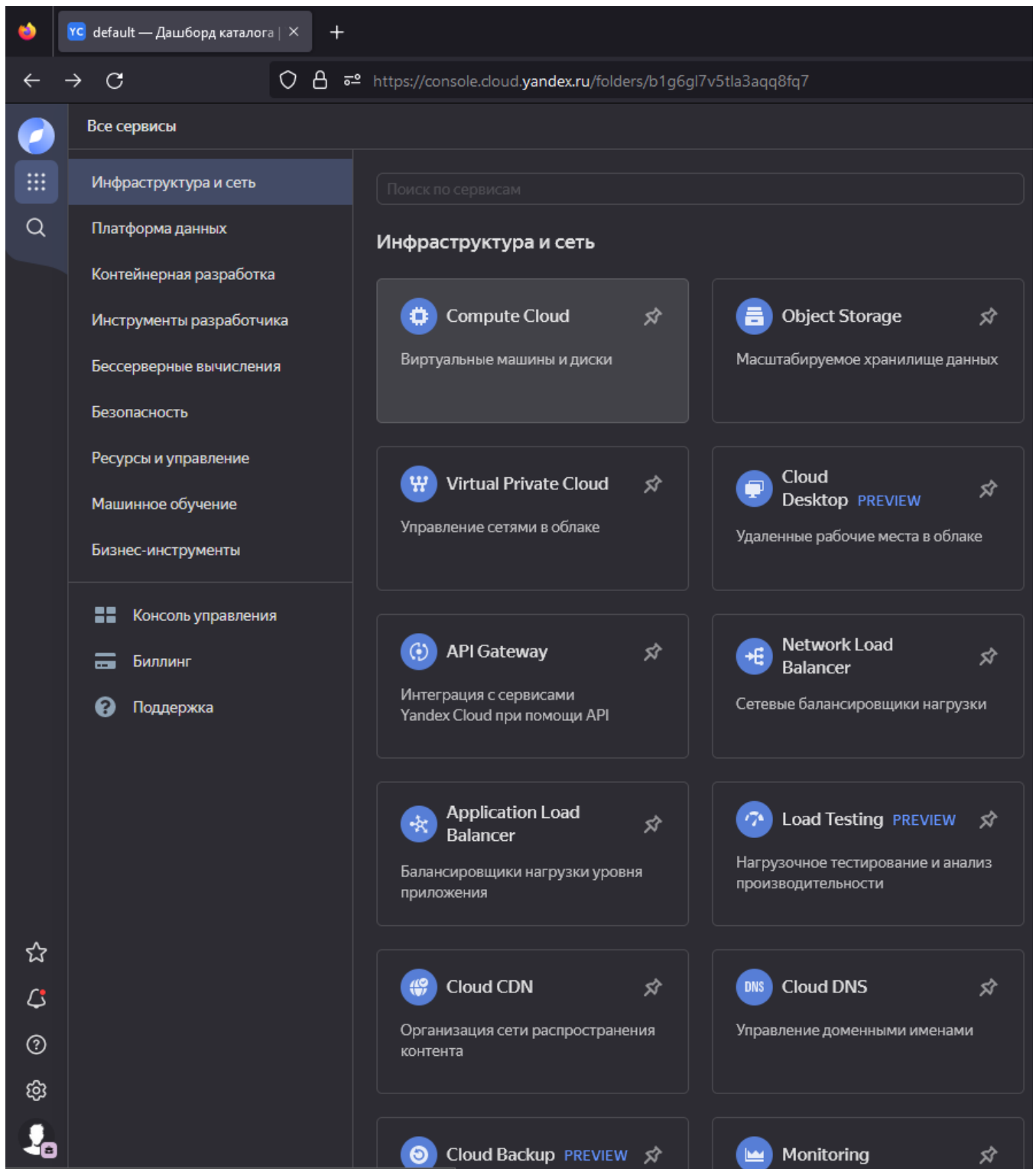
Как настроить CI/CD?

- Часть первая. Как поднять сервер в Yandex Cloud?
- Часть вторая. Как установить docker и docker compose на сервер?
- Часть третья. Как установить Nginx на сервер, написать грамотную конфигурацию для Nginx и сгенерировать автообновляемый сертификат?
- Часть четвертая. Подготовка к CI/CD. Какие существуют раннеры?
- Часть пятая. Стандарты по выбору GitLab раннера от DevOps'a. Почему нужно использовать Docker Runner и как его установить?
- Часть шестая. Специфичные раннеры. А что если у меня необычный случай?
- Часть седьмая. Как подключить GitLab Runner к проекту?
- Часть восьмая. Мой первый CI/CD пайплайн (использовать только для ознакомления). Что такое CI/CD и какие существуют основные термины? Что такое GitFlow?
- Часть девятая. Стандарты для деплоя фронтального приложения через CI/CD (внутри нашей компании)
- Часть десятая. Стандарты для деплоя бэкэнда через CI/CD (внутри нашей компании). А нужен ли мне Docker Registry?

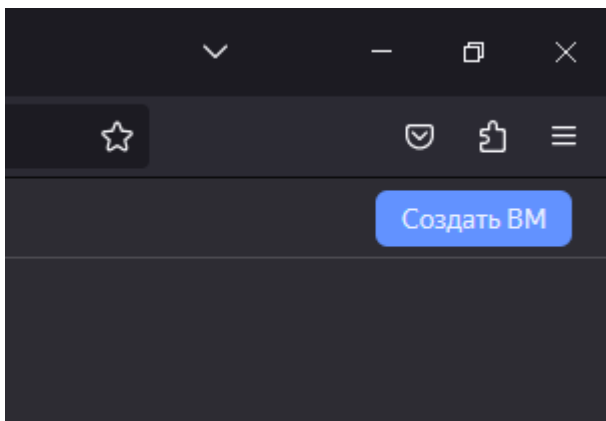
Часть первая. Как поднять сервер в Yandex Cloud?

Для поднятия сервера в Yandex Cloud Вам нужно выполнить следующие шаги:

1. Для необходимо перейти в "Compute Cloud"



2. Создать виртуальную машину (VM) в разделе "Compute Cloud".
В правом верхнем углу нажать на кнопку "Создать VM"



3. Далее необходимо ввести имя, выбрать операционную систему и желаемый тип машины.

Создание виртуальной машины

Базовые параметры

Имя [?]







Описание [?]

Зона доступности [?]

Выбор образа/загрузочного диска

Операционные системы Container Solution Cloud Marketplace Пользовательские

Фильтр по операционной системе

 Ubuntu	22.04 [▼] ⁱ	 CentOS	7 [▼] ⁱ
 Debian	11 [▼] ⁱ	 Fedora	35 [▼] ⁱ
 openSUSE Leap	15.3 [▼] ⁱ	 Fedora CoreOS	35 [▼] ⁱ

[Показать все продукты](#)

4. Настроить параметры машины, такие как количество ядер CPU, объем ОЗУ и дисковое пространство.

Имя диска	Тип	Размер	Макс. IOPS ?	Макс. bandwidth ?
Ubuntu 22.04 LTS	Загрузочный	SSD	15 ГБ	1000 / 1000
		5 ГБ	8192 ГБ	

[Добавить диск](#)

Вычислительные ресурсы

Платформа ? Intel Ice Lake

vCPU 2 2 96

Гарантированная доля vCPU ? 20% 50% 100%

Для решения любых задач, в том числе для высоконагруженных сервисов.

RAM 2 ГБ 2 ГБ 32 ГБ

Дополнительно ☐ Прерываемая ?

Сетевые настройки

Подсеть ? vpn / vpn-server

Публичный адрес Автоматически Список Без адреса

Дополнительно ☐ Защита от DDoS-атак ?

Внутренний IPv4-адрес Автоматически Вручную

Настройки DNS для внутренних адресов

Группы безопасности Не выбрано

5. Создать ssh ключ

```
ssh-keygen -t rsa -b 4096
```

далее необходимо указать путь до ключа его имя

путь для Windows: C:\Users\username\.ssh/key_name

путь для Linux и MacOS: ~/.ssh/key_name

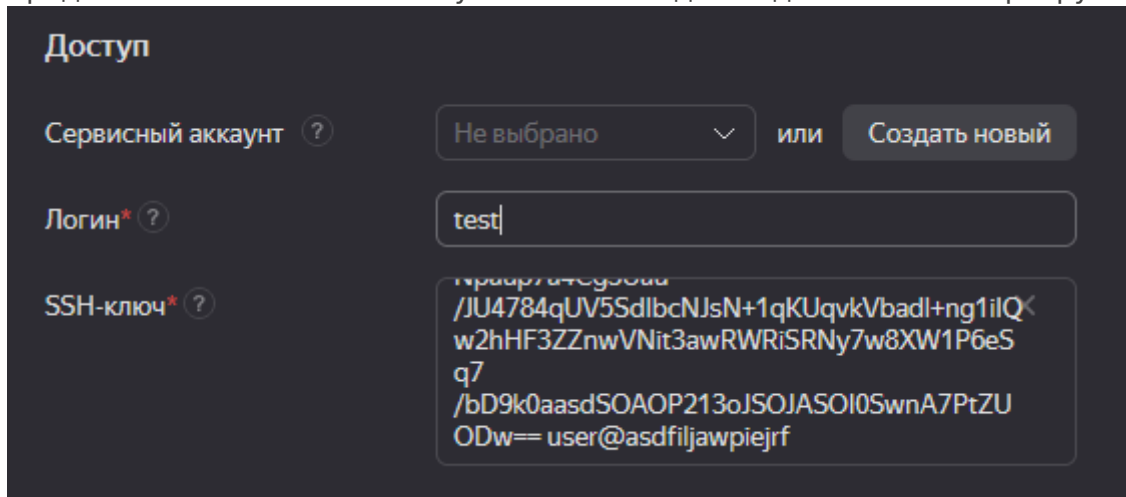
в итоге будет создано два ключа:

приватный - key_name и публичный - key_name.pub

для активации приватного ключа на Linux и MacOS необходимо будет выполнить команду:

```
chmod 400 ~/.ssh/key_name
```

После этого, нужно будет скопировать публичный ключ, вставить его в представленное ниже окно и указать логин для подключения к серверу.



6. Подключиться к серверу через SSH.

Делается это следующим образом:

```
ssh login@ip_address -i ~/.ssh/key_name
# login - это имя пользователя, через которого происходит коннект к серверу
# ip_address - это ip-адрес сервера
# через ключ -i указывается путь до приватного ssh ключа
```

7. Настроить необходимые сервисы и приложения на сервере.

Подробная документация по созданию и настройке виртуальных машин в Yandex Cloud доступна на официальном сайте: <https://cloud.yandex.ru/docs/compute/quickstart>.

Часть вторая. Как установить docker и docker compose на сервер?

Для установки Docker'a и Docker Compose на сервер необходимо выполнить следующие шаги:

1. Обновите пакеты вашей операционной системы:

```
sudo apt-get update
```

2. Установите зависимости, необходимые для добавления новых репозиториев:

```
sudo apt-get install \
    ca-certificates \
    curl \
    gnupg \
    lsb-release
```

3. Добавьте официальный репозиторий Docker в систему:

```
sudo mkdir -m 0755 -p /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o
/etc/apt/keyrings/docker.gpg

echo \
    "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg]
https://download.docker.com/linux/ubuntu \
    $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list >
/dev/null
```

4. Обновите список пакетов и установите Docker:

```
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin
docker-compose-plugin
```

Для того, чтобы взаимодействовать с докером без `sudo`, необходимо выполнить следующие команды:

```
sudo groupadd docker
sudo usermod -aG docker $USER
newgrp docker
```

Эти команды связаны с управлением правами доступа пользователя к Docker.

1. `sudo groupadd docker` - создает новую группу под названием "docker". Это необходимо для управления доступом к Docker с помощью группы.
2. `sudo usermod -aG docker $USER` - добавляет текущего пользователя в группу "docker". Команда `usermod` используется для изменения параметров учетной записи пользователя, а флаг `-aG` означает "добавить пользователя в группу". `$USER` - это переменная, содержащая имя текущего пользователя. После выполнения этой команды, пользователь получает возможность управлять Docker без использования `sudo`.
3. `newgrp docker` - команда запускает новую оболочку с группой "docker" в качестве основной группы текущего пользователя. Это необходимо, чтобы изменения прав доступа вступили в силу без перезапуска сеанса пользователя.

После выполнения этих команд, текущий пользователь должен иметь доступ к управлению Docker без использования команды `sudo`. Однако, не забывайте, что предоставление пользователю привилегий без должного понимания может быть опасно для безопасности системы.

Часть третья. Как установить Nginx на сервер, написать грамотную конфигурацию для Nginx и сгенерировать автообновляемый сертификат?

Для того, чтобы установить Nginx на Ubuntu 22.04, необходимо выполнить следующие команды:

```
sudo apt update  
sudo apt install nginx
```

Для управления процессами Nginx используются команды для запуска, остановки, перезапуска, проверки статуса и перечитывания конфигурационного файла. Вот основные команды:

1. Запустить процесс Nginx:

```
sudo systemctl start nginx
```

 или

```
sudo service nginx start
```

2. Остановить процесс Nginx:

```
sudo systemctl stop nginx
```

 или

```
sudo service nginx stop
```

3. Перезапустить процесс Nginx:

```
sudo systemctl restart nginx
```

 или

```
sudo service nginx restart
```

4. Проверить статус процесса Nginx:

```
sudo systemctl status nginx
```

 или

```
sudo service nginx status
```

5. Перечитать конфигурационный файл без перезапуска процесса Nginx:

```
sudo systemctl reload nginx
```

 или

```
sudo service nginx reload
```

Как сделать приложением доступным через интернет по доменному имени используя Nginx?

Для этого можно использовать `sites-available` и `sites-enabled` (их можно найти по следующему пути - `/etc/nginx/`) - это две директории, которые используются для организации конфигурации веб-сервера Nginx.

Директория `sites-available` содержит все доступные конфигурационные файлы для ваших сайтов, в то время как `sites-enabled` содержит символические ссылки на активные конфигурационные файлы, которые в данный момент используются сервером.

При настройке нового сайта в Nginx, вы создаете конфигурационный файл в директории `sites-available`, а затем создаете символическую ссылку на этот файл в директории `sites-enabled`. Это позволяет Nginx использовать новый конфигурационный файл без необходимости перезапуска сервера.

Также можно легко отключить сайт, удалив его символическую ссылку в `sites-enabled`. Все конфигурационные файлы в `sites-available` остаются доступными и могут быть легко включены снова, создав символическую ссылку в `sites-enabled`.

Использование `sites-available` и `sites-enabled` упрощает управление конфигурацией веб-сервера и позволяет быстро переключаться между различными сайтами и настройками.

Ниже представлен базовый конфиг для домена `example.com`:

```
server {
    listen 80;
    listen [::]:80;

    server_name example.com;

    location / {
        return 301 https://$host$request_uri;
    }
}

server {
    listen 443 ssl http2;
    listen [::]:443 ssl http2;

    server_name example.com;

    root /var/www/example.com;
```

```
index index.html;

include /etc/nginx/mime.types;
charset utf-8;

# Настраиваем логирование
access_log /var/log/nginx/example.com-access.log;
error_log /var/log/nginx/example.com-error.log;

# Настраиваем SSL
ssl_certificate /etc/letsencrypt/live/example.com/fullchain.pem;
ssl_certificate_key /etc/letsencrypt/live/example.com/privkey.pem;

# Настраиваем локацию для статических файлов
location /static/ {
    alias /var/www/example.com/static/;
    expires 30d;
}

# Настраиваем локацию для динамических запросов
location / {
    proxy_pass http://localhost:8000;
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
}
}
```

Далее, создайте символическую ссылку на этот файл в директории `sites-enabled`, чтобы Nginx мог использовать его:

```
sudo ln -s /etc/nginx/sites-available/example.com /etc/nginx/sites-enabled
```

Теперь Nginx будет использовать файл настроек для сайта example.com. Проверьте конфигурационный файл на наличие ошибок, используя команду:

```
sudo nginx -t
```

Если ошибок не обнаружено, перезапустите Nginx, чтобы изменения вступили в силу:

```
sudo systemctl restart nginx
```

Теперь ваш сервер Nginx настроен для обслуживания сайта example.com с помощью безопасного соединения HTTPS с помощью сертификата, который был сгенерирован с помощью Certbot. Важно отметить, что сертификаты должны быть обновлены каждые несколько месяцев, чтобы оставаться действительными. Для этого можно использовать автоматические средства обновления сертификатов Certbot.

Для генерации автообновляемого сертификата Certbot для Nginx на Ubuntu, выполните следующие шаги:

1. Установите Certbot:

```
sudo apt install certbot python3-certbot-nginx
```

2. Запустите Certbot и следуйте инструкциям по запросу и установке SSL-сертификата для вашего домена:

```
sudo certbot --nginx -d example.com
```

3. Проверьте, что конфигурация Nginx была обновлена для использования SSL-сертификата:

```
sudo nginx -t
```

4. Перезапустите Nginx, чтобы изменения вступили в силу:

```
sudo systemctl restart nginx
```

Теперь SSL-сертификат для вашего домена был установлен и будет автоматически обновляться каждые 3 месяца (если настроен автоматический обновление сертификатов).

Чтобы посмотреть какие домены настроены на автоматическое обновление сертификатов, можно перейти в директорию `/etc/letsencrypt/renewal`, а дальше выполнить команду `sudo certbot renew --dry-run` для проверки автоматического обновления сертификатов Certbot.

Часть четвертая.

Подготовка к CI/CD. Какие существуют раннеры?

GitLab Runner - это инструмент для автоматизации CI/CD процессов, который используется в GitLab. Он представляет собой агента, который выполняет команды сборки, тестирования и развертывания на удаленных машинах или контейнерах.

GitLab Runner работает в связке с GitLab CI/CD, которая позволяет автоматизировать сборку, тестирование и развертывание приложений, написанных на различных языках и работающих на разных платформах. GitLab CI/CD использует файлы конфигурации `.gitlab-ci.yml`, которые описывают этапы сборки и развертывания проекта.

GitLab Runner поддерживает множество способов запуска команд, таких как Docker, Kubernetes, SSH и другие. Он также позволяет выполнять многопоточную сборку и тестирование приложений, что ускоряет процесс разработки и улучшает производительность.

Ниже представлены основные типы GitLab раннеров:

1) GitLab SSH Runner - это способ запуска GitLab CI/CD задач, когда GitLab Runner не может быть установлен на целевой сервер или когда задачи должны быть выполнены на удаленном сервере.

GitLab SSH Runner подключается к удаленному серверу по SSH и выполняет команды сборки, тестирования и развертывания проекта на этом сервере. Это позволяет использовать удаленные серверы для выполнения задач, что может быть полезно, например, для развертывания приложений на боевых серверах или для сборки проектов на серверах с большим объемом ресурсов.

GitLab SSH Runner использует публичный ключ SSH для аутентификации на удаленном сервере и выполнения задач. При установке GitLab SSH Runner создает отдельный SSH-ключ, который должен быть добавлен на удаленный сервер, чтобы позволить GitLab SSH Runner подключаться к нему.

В целом, GitLab SSH Runner предоставляет удобный способ выполнения GitLab CI/CD задач на удаленных серверах через SSH без необходимости установки и настройки GitLab Runner на каждом сервере.

2) GitLab Shell Runner - это альтернативный способ запуска GitLab CI/CD задач, когда GitLab Runner не может быть установлен на целевой сервер или когда задачи должны быть выполнены на удаленном сервере, но не может быть использован SSH-доступ.

GitLab Shell Runner использует GitLab Shell для запуска команд на удаленном сервере. GitLab Shell - это оболочка, которая обрабатывает git-команды для удаленного репозитория GitLab. Она используется для авторизации и аутентификации пользователя, а также для выполнения команд сборки, тестирования и развертывания проекта на удаленном сервере.

GitLab Shell Runner устанавливается на удаленный сервер и настраивается для выполнения задач. Когда задача запускается на GitLab, GitLab Shell Runner получает запрос от GitLab, и запускает команды, определенные в задаче, на удаленном сервере.

GitLab Shell Runner также использует публичный ключ SSH для аутентификации на удаленном сервере и выполнения задач. При установке GitLab Shell Runner создает отдельный SSH-ключ, который должен быть добавлен на удаленный сервер, чтобы позволить GitLab Shell Runner подключаться к нему.

В целом, GitLab Shell Runner предоставляет альтернативный способ выполнения GitLab CI/CD задач на удаленных серверах через GitLab Shell без необходимости установки и настройки GitLab Runner на каждом сервере.

3) GitLab VirtualBox Runner - это GitLab Runner, который работает на виртуальной машине VirtualBox.

GitLab Runner - это приложение, которое позволяет запускать процессы CI/CD в GitLab. Это может быть полезно для автоматизации процессов сборки, тестирования и доставки приложений.

GitLab VirtualBox Runner может быть полезен, если вам нужно запускать процессы CI/CD на отдельной виртуальной машине, а не на локальной машине, где находится GitLab. Он может быть установлен на любой операционной системе, которая поддерживает VirtualBox. После установки и настройки VirtualBox Runner будет работать автономно и выполнять процессы CI/CD в соответствии с настройками GitLab.

4) GitLab Parallels Runner - это GitLab Runner, который работает на виртуальной машине Parallels Desktop.

GitLab Runner - это приложение, которое позволяет запускать процессы CI/CD в GitLab. Он может быть полезен для автоматизации процессов сборки, тестирования и доставки приложений.

GitLab Parallels Runner может быть полезен, если вам нужно запускать процессы CI/CD на отдельной виртуальной машине, а не на локальной машине, где находится GitLab. Он может быть установлен на любой операционной системе, которая поддерживает Parallels Desktop. После установки и настройки Parallels Runner будет работать автономно и выполнять процессы CI/CD в соответствии с настройками GitLab.

5) GitLab Docker Runner - это GitLab Runner, который использует контейнеры Docker для запуска процессов CI/CD в GitLab.

GitLab Runner - это приложение, которое позволяет запускать процессы CI/CD в GitLab. Он может быть полезен для автоматизации процессов сборки, тестирования и доставки приложений. GitLab Docker Runner использует Docker-контейнеры для изоляции процессов CI/CD, что делает его более надежным и безопасным.

GitLab Docker Runner может быть полезен, если вам нужно запускать процессы CI/CD на отдельной виртуальной машине, а не на локальной машине, где находится GitLab. Он может быть установлен на любой операционной системе, которая поддерживает Docker. После установки и настройки Docker Runner будет работать автономно и выполнять процессы CI/CD в соответствии с настройками GitLab.

6) GitLab Docker SSH Runner - это GitLab Runner, который использует Docker-контейнеры для запуска процессов CI/CD и SSH-сервера для подключения к удаленному хосту по SSH-протоколу.

GitLab Docker SSH Runner может быть полезен, если вы хотите запускать процессы CI/CD на удаленном хосте, который не имеет доступа к GitLab API. В этом случае Docker Runner используется для запуска процессов CI/CD в изолированных контейнерах Docker, а SSH-сервер используется для подключения к удаленному хосту по SSH-протоколу для выполнения команд.

Для работы GitLab Docker SSH Runner требуется наличие SSH-ключей для подключения к удаленному хосту и конфигурации Docker. После установки и настройки Docker SSH Runner будет работать автономно и выполнять процессы CI/CD в соответствии с настройками GitLab.

7) GitLab Kubernetes Runner - это GitLab Runner, который использует Kubernetes для запуска процессов CI/CD. Он позволяет запускать процессы CI/CD в контейнерах Kubernetes, что обеспечивает высокую степень изолированности и масштабируемости.

GitLab Kubernetes Runner может быть полезен в случае, если у вас есть кластер Kubernetes, и вы хотите использовать его для запуска процессов CI/CD. Он может использоваться для запуска процессов CI/CD как на локальном кластере Kubernetes, так и на облачных платформах, таких как Google Cloud Platform, AWS или Microsoft Azure.

Для работы GitLab Kubernetes Runner требуется наличие кластера Kubernetes и настройки соответствующих Kubernetes объектов, таких как Pod, Service и Secret, а также конфигурации GitLab Runner. После установки и настройки GitLab Kubernetes Runner будет работать автономно и выполнять процессы CI/CD в соответствии с настройками GitLab.

8) GitLab Custom Runner - это возможность запуска GitLab Runner на своей собственной инфраструктуре или в облачной среде, такой как Amazon Web Services, Google Cloud Platform или Microsoft Azure. С помощью GitLab Custom Runner вы можете использовать свои собственные серверы для запуска заданий CI/CD, а также настроить окружения сборки в соответствии с вашими требованиями. GitLab Custom Runner может быть установлен на

любом сервере, который поддерживает GitLab Runner, в том числе на серверах Linux, macOS и Windows.

Часть пятая. Стандарты по выбору GitLab раннера от DevOps'a. Почему нужно использовать Docker Runner и как его установить?

GitLab Docker Runner - это runner, который работает в контейнере Docker и используется для запуска GitLab CI/CD-задач в изолированном окружении. Он имеет следующие преимущества:

1. **Изолированность:** GitLab Docker Runner обеспечивает полную изоляцию приложений, которые выполняются на нем, от операционной системы хоста, что обеспечивает безопасность и предотвращает возможные конфликты между приложениями.
2. **Портативность:** GitLab Docker Runner может быть запущен на любой машине, где установлен Docker, и его можно легко перенести из одного окружения в другое, без необходимости настройки всех параметров заново.
3. **Универсальность:** Docker Runner может использоваться для любого языка программирования, тестирования, сборки и развертывания приложений, так как в Docker-контейнере можно запустить любое окружение.
4. **Простота настройки:** Настройка Docker Runner происходит с помощью нескольких команд в терминале, что делает процесс установки и настройки быстрым и простым.
5. **Масштабируемость:** GitLab Docker Runner может быть масштабирован горизонтально и вертикально, что обеспечивает возможность расширения на более мощные машины или добавление новых машин в кластер для обработки большого количества задач.

Для настройки GitLab Docker Runner требуется создать Docker-образ, в котором установлены необходимые зависимости и инструменты для выполнения задач. Затем нужно зарегистрировать runner в GitLab и настроить его параметры в файле конфигурации. После этого runner готов к запуску задач CI/CD на основе определенных триггеров.

Чтобы установить GitLab Docker Runner на сервер Ubuntu для домена gitlab.com и токена testtesttest, а также настроить его для запуска контейнеров с тегом "test", выполните следующие шаги:

1. Скачайте бинарный файл для вашей системы:

```
sudo curl -L --output /usr/local/bin/gitlab-runner https://gitlab-runner-downloads.s3.amazonaws.com/latest/binaries/gitlab-runner-linux-amd64
```

2. Дайте разрешение на выполнение:

```
sudo chmod +x /usr/local/bin/gitlab-runner
```

3. Создайте пользователя gitlab-runner:

```
sudo useradd --comment 'GitLab Runner' --create-home gitlab-runner --shell /bin/bash
```

4. Установите и запустите как сервис:

```
sudo gitlab-runner install --user=gitlab-runner --working-directory=/home/gitlab-runner
sudo gitlab-runner start
```

5. Команда для регистрации раннера для домена gitlab.com, токена testtesttest (оригинальный будет находится в GitLab внутри репозитория по пути - Settings -> CI/CD -> Runners), с тэгом test:

```
sudo gitlab-runner register \
  --url https://gitlab.com/ \
  --registration-token testtesttest \
  --description "Docker Runner" \
  --executor docker \
  --docker-image "docker:latest" \
  --docker-privileged \
  --tag-list "test"
```

Пример настройки `config.toml` для GitLab Docker Runner с привелегированным режимом, использованием кэша и образом Docker, а также с параллельным запуском для 3 одновременных задач:

```
concurrent = 3

[[runners]]
  name = "Docker Runner"
  url = "https://gitlab.com/"
  token = "YOUR_TOKEN_HERE"
  executor = "docker"
  [runners.docker]
    tls_verify = true
    image = "docker:latest"
    privileged = true
    disable_cache = false
    volumes = ["/cache"]
    shm_size = 0
  [runners.cache]
    Type = "s3"
    Shared = true
    [runners.cache.s3]
      ServerAddress = "s3.amazonaws.com"
      AccessKey = "ACCESS_KEY"
      SecretKey = "SECRET_KEY"
      BucketName = "BUCKET_NAME"
      BucketLocation = "us-east-1"
```

В данном примере:

- `concurrent = 3` указывает, что одновременно может запускаться до 3 задач
- `name`, `url` и `token` указывают на имя, адрес и токен вашего GitLab экземпляра
- `executor = "docker"` указывает на то, что GitLab Runner будет запускать задачи в контейнерах Docker
- `image = "docker:latest"` определяет Docker образ, который будет использоваться для запуска задач
- `privileged = true` позволяет контейнерам запускаться с привилегированным режимом
- `disable_cache = false` указывает на использование кэша GitLab Runner
- `volumes = ["/cache"]` определяет место хранения кэша
- `shm_size = 0` отключает использование разделяемой памяти в контейнерах
- `Type = "s3"` указывает на использование Amazon S3 в качестве хранилища кэша
- `AccessKey`, `SecretKey`, `BucketName` и `BucketLocation` - это настройки для доступа к S3.

Обратите внимание, что для использования S3 в качестве хранилища кэша, необходимо установить пакет `awscli` и настроить учетную запись IAM на AWS.

Часть шестая.

Специфичные раннеры. А что если у меня необычный случай?

GitLab Shell, SSH Runner и Docker SSH Runner - это инструменты, которые позволяют автоматизировать процесс сборки, тестирования и развертывания приложений в GitLab CI/CD.

GitLab Shell - это командный интерфейс, который используется для выполнения некоторых действий, связанных с репозиторием GitLab, таких как получение содержимого файлов, создание веток и тегов, а также редактирование файлов.

SSH Runner - это GitLab Runner, который использует SSH для подключения к удаленным хостам и выполнения команд на них. Это позволяет запускать сборку, тестирование и развертывание приложений на удаленных хостах.

Docker SSH Runner - это GitLab Runner, который запускается внутри контейнера Docker и использует SSH для подключения к удаленным хостам. Это позволяет запускать сборку, тестирование и развертывание приложений на удаленных хостах, используя Docker-контейнеры.

Преимущества использования SSH и Docker SSH Runner включают:

- Возможность запускать сборку, тестирование и развертывание приложений на удаленных хостах.
- Ускорение процесса сборки и тестирования приложений за счет распределения нагрузки между несколькими хостами.
- Изоляция приложения в Docker-контейнере, что позволяет избежать конфликтов с другими приложениями на хосте.
- Удобство настройки и использования благодаря интеграции с GitLab CI/CD.

GitLab Shell, SSH Runner и Docker SSH Runner можно использовать в случаях, когда необходимо автоматизировать процесс сборки, тестирования и развертывания приложений в GitLab CI/CD, а также в случаях, когда необходимо запускать эти процессы на удаленных

хостах.

Установка для всех этих раннеров одинаковая, отличается только выбор исполнителя (executor), ниже представлен процесс установки (для более подробного изучения процесса установки, смотрите [предыдущую главу](#)):

```
# Download the binary for your system
sudo curl -L --output /usr/local/bin/gitlab-runner https://gitlab-runner-
downloads.s3.amazonaws.com/latest/binaries/gitlab-runner-linux-amd64

# Give it permission to execute
sudo chmod +x /usr/local/bin/gitlab-runner

# Create a GitLab Runner user
sudo useradd --comment 'GitLab Runner' --create-home gitlab-runner --shell /bin/bash

# Install and run as a service
sudo gitlab-runner install --user=gitlab-runner --working-directory=/home/gitlab-runner
sudo gitlab-runner start

# Command to register runner
sudo gitlab-runner register --url https://git.fly-code.com/ --registration-token
$REGISTRATION_TOKEN
```

Теперь поговорим о настройке этих раннеров:

1) Чтобы установить и настроить GitLab Shell Runner, выполните следующие шаги:

1. Убедитесь, что у вас установлен GitLab Runner. Если он не установлен, следуйте инструкциям по установке GitLab Runner, приведенным в официальной документации GitLab.
2. Создайте новый Shell Runner в GitLab. Для этого зайдите в раздел "Settings" вашего проекта в GitLab, выберите "CI/CD" и нажмите на кнопку "Show runner installation instructions".
3. Следуйте инструкциям на экране для создания Shell Runner. При создании Runner обязательно запомните токен раннера, который будет использоваться для регистрации Runner на сервере.
4. Откройте терминал на сервере, на котором установлен GitLab Runner.
5. Зарегистрируйте новый Shell Runner, используя следующую команду:

```
sudo gitlab-runner register -n \
  --url https://gitlab.com/ \
  --registration-token REGISTRATION_TOKEN \
```

```
--executor shell \  
--description "My Shell Runner"
```

Где:

- --url - URL GitLab
- --registration-token - токен вашего Runner
- --executor - тип Executor, который вы используете (в данном случае это Shell)
- --description - название вашего Shell Runner

6. После регистрации Runner отредактируйте файл конфигурации GitLab Runner `/etc/gitlab-runner/config.toml`, чтобы настроить свой новый Shell Runner.

Ниже приведен пример файла конфигурации для GitLab Shell Runner:

```
[[runners]]  
  name = "My Shell Runner"  
  url = "https://gitlab.com/"  
  token = "REGISTRATION_TOKEN"  
  executor = "shell"  
  shell = "/bin/bash"  
  builds_dir = "/var/lib/gitlab-runner/builds"  
[runners.cache]  
  [runners.cache.s3]  
  [runners.cache.gcs]  
  [runners.cache.azure]  
  [runners.cache.gcs]
```

7. После того, как вы настроили файл конфигурации, перезапустите GitLab Runner, используя следующую команду:

```
sudo gitlab-runner restart
```

Теперь ваш GitLab Shell Runner готов к использованию для сборки, тестирования и развертывания вашего приложения на локальном сервере.

2) Чтобы установить GitLab SSH Runner, выполните следующие шаги:

1. Убедитесь, что у вас установлен GitLab Runner. Если он не установлен, следуйте инструкциям по установке GitLab Runner, приведенным в официальной документации GitLab.
2. Создайте новый SSH Runner в GitLab. Для этого зайдите в раздел "Settings" вашего проекта в GitLab, выберите "CI/CD" и нажмите на кнопку "Show runner installation instructions".

3. Следуйте инструкциям на экране для создания SSH Runner. При создании Runner обязательно запомните токен раннера, который будет использоваться для регистрации Runner на сервере.
4. Откройте терминал на сервере, на котором установлен GitLab Runner.
5. Зарегистрируйте новый SSH Runner, используя следующую команду:

```
sudo gitlab-runner register -n \  
  --url https://gitlab.com/ \  
  --registration-token REGISTRATION_TOKEN \  
  --executor ssh \  
  --description "My SSH Runner" \  
  --ssh-user my_user \  
  --ssh-host my_host \  
  --ssh-port 22 \  
  --ssh-identity-file /path/to/my/key
```

Где:

- --url - URL GitLab
- --registration-token - токен вашего Runner
- --executor - тип Executor, который вы используете (в данном случае это SSH)
- --description - название вашего SSH Runner
- --ssh-user - имя пользователя, под которым будет выполняться команда на удаленном хосте
- --ssh-host - адрес удаленного хоста
- --ssh-port - порт для подключения к удаленному хосту (по умолчанию 22)
- --ssh-identity-file - путь к файлу ключа SSH

6. После регистрации Runner запустите его с помощью следующей команды:

```
sudo gitlab-runner restart
```

Вот пример `config.toml` для SSH GitLab Runner:

```
[[runners]]  
  name = "My SSH Runner"  
  url = "https://gitlab.com/"  
  token = "REGISTRATION_TOKEN"  
  executor = "ssh"  
  [runners.ssh]  
    user = "USER"  
    host = "HOSTNAME_OR_IP_ADDRESS"  
    port = "22"  
    identity_file = "/path/to/private/key"  
    password = "PASSWORD" # если используется пароль для аутентификации
```



```

# Перечисление дополнительных параметров SSH, например:
#   - strict_host_key_checking=no
#   - UserKnownHostsFile=/dev/null
#   - LogLevel=ERROR
# можно указать через запятую.
# Пример:
#   extra_ssh_options = "LogLevel=ERROR,StrictHostKeyChecking=no"
extra_ssh_options = "LogLevel=ERROR,StrictHostKeyChecking=no"
# Указание каталога, в который будут загружены и запущены билды.
# Например:
#   - /home/user/builds
# По умолчанию билды сохраняются в домашней директории пользователя.
builds_dir = "/home/user/builds"
# Указание дополнительных параметров для rsync, используемого при передаче
файлов.
# Например:
#   - --exclude='.git'
#   - --exclude='build/'
#   - --delete
rsync_options = ["--exclude='.git' ", "--exclude='build/' ", "--delete"]

```

В этом примере конфигурации вы можете заменить `My SSH Runner` на свое название, а `REGISTRATION_TOKEN` на токен раннера, который был создан в вашем проекте на GitLab. Замените `USER` на имя пользователя сервера, `HOSTNAME_OR_IP_ADDRESS` на IP-адрес или имя хоста вашего сервера, `/path/to/private/key` на путь к вашему закрытому ключу, а `PASSWORD` на пароль, если используется аутентификация по паролю.

Также обратите внимание на то, что в `extra_ssh_options` вы можете указать дополнительные параметры SSH через запятую, если это необходимо. А в `rsync_options` можно указать дополнительные параметры для rsync, используемого для передачи файлов на удаленный сервер.

3) Установка GitLab Docker SSH Runner, практически ничем не отличается от установки SSH и Docker раннеров, по сути это их объединение. Вся основная сложность лежит в настройке этого раннера. Вот пример `config.toml` для Docker SSH GitLab Runner:

```

[[runners]]
  name = "My Docker SSH Runner"
  url = "https://gitlab.com/"
  token = "REGISTRATION_TOKEN"
  executor = "docker-ssh"
[runners.docker]

```

```
tls_verify = false
image = "docker:latest"
privileged = true
disable_entrpoint_overwrite = false
oom_kill_disable = false
disable_cache = false
volumes = ["/cache"]
shm_size = 0

[runners.ssh]
user = "USER"
host = "HOSTNAME_OR_IP_ADDRESS"
port = "22"
identity_file = "/path/to/private/key"
password = "PASSWORD" # если используется пароль для аутентификации
# Перечисление дополнительных параметров SSH, например:
# - strict_host_key_checking=no
# - UserKnownHostsFile=/dev/null
# - LogLevel=ERROR
# можно указать через запятую.
# Пример:
# extra_ssh_options = "LogLevel=ERROR,StrictHostKeyChecking=no"
extra_ssh_options = "LogLevel=ERROR,StrictHostKeyChecking=no"
```

В этом примере конфигурации вы можете заменить `My Docker SSH Runner` на свое название, а `REGISTRATION_TOKEN` на токен раннера, который был создан в вашем проекте на GitLab. Замените `USER` на имя пользователя сервера, `HOSTNAME_OR_IP_ADDRESS` на IP-адрес или имя хоста вашего сервера, `/path/to/private/key` на путь к вашему закрытому ключу, а `PASSWORD` на пароль, если используется аутентификация по паролю.

Кроме того, вы можете настроить параметры Docker, такие как `tls_verify`, `privileged`, `disable_entrpoint_overwrite`, `oom_kill_disable`, `disable_cache` и `shm_size`, согласно вашим потребностям.

Обратите внимание, что для Docker SSH Runner требуется установить Docker и настроить доступ к нему для пользователя, под которым запущен GitLab Runner. Вы можете добавить пользователя в группу Docker, чтобы это обеспечить.

Часть седьмая. Как подключить GitLab Runner к проекту?

Чтобы подключить GitLab Runner к проекту и использовать его в GitLab CI, выполните следующие шаги:

1. Установите GitLab Runner на сервере, используя инструкции для вашей операционной системы. После установки настройте его, используя инструкции в соответствующем разделе документации GitLab.
2. Зарегистрируйте GitLab Runner с вашим проектом, выполнив следующую команду на сервере, на котором установлен GitLab Runner:

```
gitlab-runner register
```

Введите запрашиваемую информацию, такую как URL GitLab, токен регистрации и тип раннера (shell, ssh или docker).

После регистрации GitLab Runner будет доступен в разделе "Settings > CI/CD > Runners" вашего проекта.

3. Добавьте тег к GitLab Runner, используя следующую команду:

```
gitlab-runner tag add <runner-name> <tag-name>
```

Замените `<runner-name>` на имя вашего GitLab Runner, и `<tag-name>` на желаемый тег. Это создаст новый тег и добавит его к GitLab Runner.

4. Создайте файл `.gitlab-ci.yml` в корневой папке вашего проекта. Этот файл содержит инструкции для GitLab CI о том, как выполнять ваш пайплайн. Например, следующий файл `.gitlab-ci.yml` выполняет сборку проекта и тестирование на каждом коммите в ветке master:

```
image: node:latest
```

```
stages:
```

- build
- test

```
build:
```

```
stage: build
tags:
  - release
script:
  - npm install
  - npm run build

test:
stage: test
tags:
  - release
script:
  - npm test
```

5. Обратите внимание, что мы добавили секцию `tags`, которая определяет условия, при которых задание будет выполняться. Здесь мы указали, что задание `build` `test` будет выполняться только для коммитов с тегом "release". Так же здесь используется образ `node: latest` в качестве основы для нашего пайплайна, и определяем два этапа: `build` и `test`. В каждом этапе мы выполняем набор команд, используя инструкцию `script`.
6. Сохраните файл `.gitlab-ci.yml` в корневой папке вашего проекта и выполните коммит и пуш в ваш репозиторий. GitLab CI автоматически обнаружит файл `.gitlab-ci.yml` и начнет выполнять пайплайн при каждом коммите в ветке `master`.
7. Чтобы проверить, что GitLab Runner правильно настроен и работает с вашим проектом, откройте раздел "Settings > CI/CD > Runners" вашего проекта и убедитесь, что GitLab Runner отображается как "active". Если у GitLab Runner есть проблемы с регистрацией или выполнением пайплайнов, вы можете проверить журналы и настройки GitLab Runner на сервере, где он установлен.

Чтобы сделать раннер доступным, необходимо сделать следующее:

1. Перейти в проект и зайти в Settings -> CI/CD -> Runners и нажать на кнопку с карандашом (Edit)

Assigned project runners

● #32 (5555555555)


  Remove runner

Assigned to runner

Assigned to runner

2. Далее появится следующее окно и нужно будет выбрать только первый пункт

Runner #~~XXXX~~ project

 This runner is associated with specific projects.

You can set up a project runner to be used by multiple projects but you cannot make this a shared or group runner. [Learn more.](#)

Active

☒ Paused runners don't accept new jobs

Protected

☐ This runner will only run on pipelines triggered on protected branches

Run untagged jobs

☐ Indicates whether this runner can pick jobs without tags

Lock to current projects

☐ When a runner is locked, it cannot be assigned to other projects

Active: Эта опция указывает, что данный раннер активен и может принимать задания от GitLab CI/CD.

Protected: Эта опция указывает, что данный раннер защищен от удаления или изменения его параметров на уровне GitLab.

Run untagged jobs: Эта опция указывает, что данный раннер может выполнять задания, которые не имеют тегов.

Lock to current projects: Эта опция указывает, что данный раннер может быть использован только для текущего проекта в GitLab, и его нельзя использовать для других проектов. Если вы создаете многопроектный раннер, то можете использовать эту опцию, чтобы ограничить доступ к ресурсам раннера только для определенного проекта.

3. Теперь раннер нужно сделать доступным для всех, для этого нужно убрать галочку с "Enable shared runners for this project" - эта опция позволяет включить общие раннеры для проекта. После включения этой опции, для выполнения задач по CI/CD проекта могут использоваться раннеры, доступные для всех проектов в GitLab.

Shared runners

[These runners](#) are available to all groups and projects.

Enable shared runners for this project



This GitLab instance does not provide any shared runners yet. Instance administrators can register shared runners in the admin area.

Group runners

These runners are shared across projects in this group.

Group runners can be managed with the [Runner API](#).

Disable group runners for this project

This group does not have any group runners yet. To register them, go to the [group's Runners page](#).

Часть восьмая. Мой первый CI/CD пайплайн

(использовать только для ознакомления). Что такое CI/CD и какие существуют основные термины? Что такое GitFlow?

CI/CD — это сокращение от Continuous Integration / Continuous Deployment, что означает непрерывную интеграцию и непрерывную доставку.

Continuous Integration (непрерывная интеграция) — это методология разработки, которая заключается в автоматизации процесса сборки и тестирования кода при каждом изменении в репозитории. Такой подход позволяет быстрее обнаруживать и исправлять ошибки, а также обеспечивает единый и надежный процесс сборки приложения.

Continuous Deployment (непрерывное развертывание) и Continuous Delivery (непрерывная доставка) относятся к автоматизации процесса развертывания приложения.

Continuous Deployment — это методология, при которой каждый коммит автоматически разворачивается в производственной среде, без участия человека.

Continuous Delivery — это методология, при которой каждый коммит проходит автоматические тесты, и если они успешно проходят, то приложение готово для ручного развертывания в производственной среде.

Таким образом, различие между Continuous Deployment и Continuous Delivery заключается в том, что в первом случае развертывание происходит автоматически, а во втором - только после ручного подтверждения. Оба подхода направлены на повышение скорости и качества разработки, а также уменьшение рисков ошибок в процессе развертывания приложений.

В CI/CD существует несколько основных понятий, которые используются для описания процесса автоматизации сборки и развертывания приложений:

1. Job (джоба) - это отдельная задача, которая выполняется в рамках пайплайна. Каждая джоба может выполнять определенный этап процесса CI/CD, например, сборку кода, запуск тестов, развертывание приложения и т.д.
2. Pipeline (пайплайн) - это набор джоб, которые выполняются последовательно в рамках процесса CI/CD. Пайплайн можно рассматривать как цепочку этапов, через которые проходит код на пути от исходного кода до развертывания в производственной среде.
3. Stage (стейдж) - это логическая группа джоб в пайплайне, которые выполняют одну и ту же функцию. Например, может быть стейдж для сборки кода, стейдж для запуска тестов и стейдж для развертывания приложения.
4. Artifact (артефакт) - это результат работы джобы, который может использоваться в следующих этапах пайплайна. Например, артефактом может быть скомпилированный бинарный файл, тестовый отчет или собранный Docker-образ.
5. Trigger (триггер) - это событие, которое запускает выполнение пайплайна. Например, это может быть коммит в репозиторий или ручной запуск пайплайна из интерфейса CI/CD системы.

Таким образом, джобы, пайплайны и стейджи представляют собой основные элементы процесса CI/CD, которые помогают автоматизировать различные этапы сборки и развертывания приложений, упрощая и ускоряя процесс разработки.

Для деплоя JS-приложения, используя GitLab CI, необходимо выполнить несколько шагов. В данном примере мы будем использовать GitLab CI в качестве инструмента автоматической сборки, тестирования и деплоя Node.js приложения на сервер.

1. Создать .gitlab-ci.yml файл в корне проекта.
2. Настроить стадии пайплайна. В данном примере мы будем использовать три стадии: build, test и deploy.

```
stages:  
  - build  
  - test  
  - deploy
```

3. Настроить задачи (jobs) для каждой стадии.

```
build:  
  stage: build  
  script:  
    - npm install  
    - npm run build  
  
test:  
  stage: test
```



```
script:
  - npm test

deploy:
  stage: deploy
  script:
    - ssh user@server "cd /path/to/app && git pull && npm install && pm2 restart app.js"
```

4. Добавить линтеры в пайплайн. Например, для проверки кода можно использовать ESLint.

```
build:
  stage: build
  script:
    - npm install
    - npm run lint
    - npm run build
```

5. Добавить автоматическое тестирование в пайплайн. Например, для тестирования можно использовать Mocha.

```
test:
  stage: test
  script:
    - npm install
    - npm run lint
    - npm test
```

6. Добавить деплой в пайплайн. Например, для деплоя можно использовать SSH и pm2.

```
deploy:
  stage: deploy
  script:
    - ssh user@server "cd /path/to/app && git pull && npm install && pm2 restart app.js"
```

7. Настроить переменные окружения. Например, для авторизации на сервере можно использовать SSH ключи.

```
variables:
  SSH_PRIVATE_KEY: $PRIVATE_KEY
```

8. Создать переменные окружения в GitLab.
9. Настроить пайплайн в GitLab, чтобы он запускался при пуше в ветку или при создании тега.
10. Сохранить .gitlab-ci.yml файл и запустить пайплайн в GitLab.

Таким образом, мы настроили пайплайн в GitLab, который выполняет линтеры, автоматические тесты, сборку приложения и деплой на сервер. Это позволит быстро и безопасно доставлять новые версии приложения в производственную среду.

GitFlow

Git Flow - это набор правил и рекомендаций для организации работы с Git-репозиториями в команде разработки ПО. Git Flow определяет следующие основные ветки:

- master: основная ветка, в которой содержится актуальная стабильная версия приложения;
- develop: ветка, на которой разрабатывается новый функционал и выполняются исправления ошибок;
- release/*: ветки, на которых выполняется подготовка к выпуску новой версии приложения;
- feature/*: ветки для работы над новым функционалом;
- hotfix/*: ветки для быстрого исправления критических ошибок в текущей версии приложения.

Наименование веток в Git Flow обычно следует определенным правилам. Например, для веток, созданных для работы над новым функционалом, имя ветки должно начинаться с feature/, например feature/login-page. Для веток, созданных для выпуска новой версии приложения, имя ветки должно начинаться с release/, например release/v1.2.3.

В терминологии CI/CD инстансы prod, dev, stage обычно используются для различных сред, на которых выполняются различные этапы тестирования и развертывания приложения.

- prod (production) - это окружение, в котором работает актуальная версия приложения, доступная для конечных пользователей. Все изменения должны быть тщательно протестированы и подготовлены перед выпуском в продакшн.
- dev (development) - это окружение, которое используется для разработки и тестирования нового функционала. В нем разработчики могут работать над новыми возможностями и проверять их на соответствие ожидаемому поведению приложения, прежде чем код будет отправлен на более продвинутые окружения.
- stage (staging) - это окружение, которое используется для выполнения конечных проверок перед тем, как новые изменения будут отправлены в продакшн. Здесь происходят конечные тесты, интеграционные тесты и проверки совместимости.

Каждый этап CI/CD пайплайна может использовать отдельное окружение для выполнения определенных задач. Например, тестирование юнит-тестов может быть выполнено в окружении dev, а тестирование интеграции может быть выполнено в stage перед тем, как изменения будут отправлены в prod. Важно заметить, что эти окружения могут быть установлены на разных серверах или использовать разные контейнеры, чтобы избежать конфликтов между ними.

Часть девятая. Стандарты для деплоя фронтového приложения через CI/CD (внутри нашей компании)

В корне проекта создается файл `.gitlab-ci.yml` и директория `deploy`, в ней создается `nginx.conf`, `Dockerfile`, `docker-compose.yml` (их может быть сколько угодно, в основном они создаются для конкретной ветки и имеют названия: `docker-compose.prod.yml`, `docker-compose.stage.yml`, `docker-compose.dev.yml` и т.д.).

.gitlab-ci.yml такого содержания:

```
stages:
  - test
  - build
  - deploy

test:
  image: node: $NODE_VERSION
  stage: test
  tags:
    - tags
  script:
    - cp $ENV .env
    - npm install
    - npm run test
  only:
    refs:
      - master

build:
  image: node: $NODE_VERSION
```

```

stage: build
tags:
  - tags
script:
  - cp $ENV .env
  - npm install
  - npm run build
artifacts:
  expire_in: 1 week
  paths:
    - build/
only:
  refs:
    - master

deploy:
  image: docker: $DOCKER_VERSION
  stage: deploy
  tags:
    - tags
  script:
    - docker compose -f ./deploy/docker-compose.yml build
    - docker stack deploy --with-registry-auth -c ./deploy/docker-compose.yml stack_name
  when: manual
  allow_failure: false
  needs:
    - job: build
      artifacts: true
  only:
    refs:
      - master

```

Данный файл (gitlab-ci.yml) содержит конфигурацию CI/CD пайплайна в GitLab для проекта.

Пайплайн содержит 3 этапа: `test`, `build` и `deploy`, каждый из которых представлен в отдельном блоке.

Этап `test` использует образ `node` с версией, указанной в переменной `$NODE_VERSION`, и выполняет следующие действия:

- Копирует файл настроек среды, указанный в переменной `$ENV`, в файл `.env`
- Устанавливает зависимости с помощью `npm install`

- Запускает тесты с помощью `npm run test`
- Этап `test` будет выполнен только для веток, которые начинаются с `master`

Этап `build` использует тот же образ `node` и выполняет следующие действия:

- Копирует файл настроек среды, указанный в переменной `$ENV`, в файл `.env`
- Устанавливает зависимости с помощью `npm install`
- Собирает проект с помощью `npm run build`
- Создает артефакты (архив с результатами сборки) и хранит их на сервере GitLab в течение недели
- Этап `build` будет выполнен только для веток, которые начинаются с `master`

Этап `deploy` использует образ `docker` с версией, указанной в переменной `$DOCKER_VERSION`, и выполняет следующие действия:

- Собирает Docker-образы с помощью `docker compose` на основе файла конфигурации `docker-compose.yml`
- Деплоит стек с помощью `docker stack deploy` с именем `stack_name` на основе файла конфигурации `docker-compose.yml`
- Этот этап запускается вручную с помощью кнопки "Deploy" на веб-интерфейсе GitLab
- Если этап `build` завершился успешно и создал артефакты, то этап `deploy` будет выполнен
- Если этап `build` завершился с ошибкой, то этап `deploy` не будет выполнен
- Этап `deploy` будет выполнен только для веток, которые начинаются с `master`

Переменные `$NODE_VERSION` (node:16-alpine3.15), `$DOCKER_VERSION` (docker:stable), `$ENV` - находятся в GitLab'e внутри проекта по пути - Settings -> CI/CD -> Variables. `$NODE_VERSION` и `$DOCKER_VERSION` имеют тип - variable, а `$ENV` - file.

Update variable



Key

NODE_VERSION

Value

16-alpine3.15

Type

Variable

Environment scope

All (default)

Flags

- ☐ Protect variable
Export variable to pipelines running on protected branches and tags only.
- ☐ Mask variable
Variable will be masked in job logs. Requires values to meet regular expression requirements. [Learn more.](#)
- ☒ Expand variable reference
\$ will be treated as the start of a reference to another variable.

Cancel

Delete variable

Update variable

Update variable



Key

DOCKER_VERSION

Value

stable

Type

Variable



Environment scope

All (default)



Flags



Protect variable

Export variable to pipelines running on protected branches and tags only.



Mask variable

Variable will be masked in job logs. Requires values to meet regular expression requirements. [Learn more.](#)



Expand variable reference

\$ will be treated as the start of a reference to another variable.

Cancel

Delete variable

Update variable

File

TEST_ENV





Expanded

All (default)



Update variable ×

Key

ENV

Value

```
MAIN_URL='https://MAIN_URL.com'
API_URL='https://MAIN_URL.com/api'
STRIPE_KEY='STRIPE_KEY'
REACT_APP_CHAT_URL='https://REACT_APP_CHAT_URL.com'
REACT_APP_CHAT_URL_WS='wss://REACT_APP_CHAT_URL.com/api/v1'
```

Type Environment scope [?](#)

File All (default) [?](#)

Flags [?](#)

☐ Protect variable
Export variable to pipelines running on protected branches and tags only.

☐ Mask variable
Variable will be masked in job logs. Requires values to meet regular expression requirements. [Learn more.](#)

☒ Expand variable reference
\$ will be treated as the start of a reference to another variable.

Cancel Delete variable Update variable

Синтаксис файла nginx.conf:

```
server {
    listen 80;

    location / {
        root /var/www/html;
        index index.html index.htm;
        try_files $uri $uri/ /index.html;
    }
}
```

Данный `nginx.conf` файл определяет конфигурацию сервера Nginx, который прослушивает запросы на порту 80 и обслуживает статические файлы, расположенные в директории `/var/www/html`.

Конфигурация определена в блоке `location /`, где указан корневой каталог и настройки `index` для поиска файлов индекса (по умолчанию `index.html` и `index.htm`). Также присутствует директива `try_files`, которая указывает Nginx на то, как обрабатывать

запросы, которые не соответствуют существующим файлам на сервере.

Синтаксис файла Dockerfile:

```
FROM nginx: latest

WORKDIR /var/www/html

COPY ./deploy/nginx.conf /etc/nginx/conf.d/default.conf
COPY ./build /var/www/html
```

Данный Dockerfile основан на образе `nginx: latest`, устанавливает рабочую директорию как `/var/www/html` и копирует файл `nginx.conf` в директорию `/etc/nginx/conf.d/`. Затем файлы, расположенные в директории `./build`, копируются в директорию `/var/www/html` внутри контейнера.

Таким образом, этот Dockerfile позволяет собирать и запускать Docker-образ, в котором Nginx сервер настроен для обслуживания статических файлов, содержащихся в директории `./build` внутри контейнера.

Синтаксис файла docker-compose.yml:

```
version: "3.9"

services:
  service_name:
    image: flycode/image: prod
    build:
      context: ..
      dockerfile: ./deploy/Dockerfile
    hostname: hostname
    restart: unless-stopped
    ports:
      - target: 80
        published: 8080
        protocol: tcp
        mode: host
    deploy:
      placement:
        constraints:
          - "node.labels.TAG==prod"
```

Данный `docker-compose` файл описывает один сервис, который запускается с использованием образа `flycode/image:prod` или строится из `Dockerfile`, определенного в контексте `".."` с именем файла `"./deploy/Dockerfile"`.

Для контейнера определен `hostname` с именем `hostname`, а также параметры `restart: unless-stopped`, которые позволяют автоматически перезапускать контейнер при остановке.

Также определены параметры портов (`ports`), которые связывают порт 80 внутри контейнера с портом 8080 на хост-системе. Режим `"host"` позволяет использовать сетевые настройки хост-системы вместо настроек контейнера.

В блоке `deploy` определены параметры развертывания, включая ограничения на размещение контейнера на узлах, имеющих метку `TAG==prod`.

Таким образом, данный `docker-compose` файл определяет сервис, который может быть развернут в кластере Docker Swarm и связан с портом на хост-системе для доступа к контейнеру.

Как это работает?

На сервере запускается `docker-swarm`, делается это следующим образом:

1. Инициализируйте Swarm на одном из серверов, выполнив команду:

```
sudo docker swarm init --advertise-addr <IP-адрес-узла>
```

2. После успешной инициализации Swarm на первом сервере, вы получите команду-токен, который необходимо использовать для присоединения других узлов к кластеру Swarm.
3. На каждом из оставшихся серверов, запустите команду-токен, полученный на предыдущем шаге, чтобы присоединиться к кластеру Swarm:

```
sudo docker swarm join --token <TOKEN> <IP-адрес-узла>:<PORT>
```

4. После успешного присоединения каждого узла к кластеру Swarm, проверьте состояние кластера, выполнив команду:

```
sudo docker node ls
```

Если вы видите список всех узлов кластера Swarm, значит, кластер был успешно настроен на нескольких серверах Ubuntu.

А как добавить тэг, для разграничения запуска приложений на серверах? Делается это следующим образом:

1. Перейдите на сервер, где находится узел Swarm, к которому вы хотите добавить метку.

2. Запустите команду Docker для добавления метки. Например, если вы хотите добавить метку "prod" к узлу с именем "worker":

```
docker node update --label-add TAG=prod worker
```

3. Проверьте, что метка была успешно добавлена, выполнив команду:

```
docker node inspect worker --pretty
```

Вы увидите, что метка "prod" была добавлена к узлу Swarm.

4. Теперь вы можете использовать эту метку в своих командах и в настройках развертывания сервисов в кластере Swarm. Например, вы можете развернуть сервис на всех узлах с меткой "prod", используя следующую команду:

```
docker service create --name my-service --replicas 3 --placement-pref  
'spread=node.labels.TAG=="prod"' nginx:latest
```

Эта команда развернет сервис Nginx на трех узлах, которые имеют метку "prod". Таким образом, добавление метки "prod" в кластер Swarm позволяет вам настраивать развертывание сервисов на конкретных узлах с определенными характеристиками и свойствами.

Часть десятая. Стандарты для деплоя бэкэнда через CI/CD (внутри нашей компании). А нужен ли мне Docker Registry?

Что такое Docker Registry?

Docker registry - это хранилище Docker-образов, которые могут быть загружены и использованы другими разработчиками и командами. Оно позволяет сохранять Docker-образы, созданные в локальной среде разработки, в централизованном месте, где они могут быть использованы другими разработчиками и командами в рамках одного проекта или между различными проектами.

Docker registry позволяет управлять версиями Docker-образов, обновлять их и контролировать доступ. Также, Docker registry может быть настроен на автоматическую сборку и обновление Docker-образов при изменениях в коде.

Docker registry используется в различных сценариях разработки и развертывания приложений, например, в CI/CD-системах для автоматической сборки и развертывания приложений, в облачных средах для управления Docker-образами, и т.д.

Как установить Docker Registry на сервер?

Для установки Docker Registry на Ubuntu 22.04 и задания пароля с помощью htpasswd нужно выполнить следующие шаги:

1. Установить Docker и Docker Compose, если они еще не установлены:

```
sudo apt update  
sudo apt install docker.io docker-compose
```

2. Создать файл docker-compose.yml для настройки Docker Registry:

```
version: '3'

services:
  registry:
    container_name: registry
    restart: always
    image: registry:2
    ports:
      - 5000:5000
    environment:
      REGISTRY_HTTP_TLS_CERTIFICATE: /certs/domain.crt
      REGISTRY_HTTP_TLS_KEY: /certs/domain.key
      REGISTRY_AUTH: htpasswd
      REGISTRY_AUTH_HTPASSWD_PATH: /auth/htpasswd
      REGISTRY_AUTH_HTPASSWD_REALM: Registry Realm
    volumes:
      - /opt/docker-registry/data:/var/lib/registry
      - /opt/docker-registry/auth:/auth
      - /opt/docker-registry/certs:/certs
```

3. Создать папки для хранения данных, аутентификации и сертификатов:

```
sudo mkdir -p /opt/docker-registry/data
sudo mkdir -p /opt/docker-registry/auth
sudo mkdir -p /opt/docker-registry/certs
```

4. Сгенерировать SSL-сертификаты и сохранить их в папке /opt/docker-registry/certs:

```
sudo openssl req -newkey rsa:4096 -nodes -sha256 -keyout /opt/docker-
registry/certs/domain.key -x509 -days 365 -out /opt/docker-registry/certs/domain.crt
```

5. Создать файл htpasswd для аутентификации пользователей. В примере используется имя пользователя "user1" и пароль "password1":

```
sudo mkdir -p /opt/docker-registry/auth
sudo apt install apache2-utils
sudo htpasswd -Bbn user1 password1 > /opt/docker-registry/auth/htpasswd
```

6. Запустить Docker Registry с помощью Docker Compose:

```
sudo docker-compose up -d
```

Теперь Docker Registry запущен и доступен по адресу `https://<your-server-ip>:5000`. Для загрузки образа в Docker Registry нужно использовать команду `docker push` с указанием тега образа, например:

```
docker login <your-server-ip>:5000
docker tag my-image:latest <your-server-ip>:5000/my-image:latest
docker push <your-server-ip>:5000/my-image:latest
```

Для загрузки образа на сервере, где установлен Docker Registry, нужно использовать команду `docker pull`, например:

```
docker pull <your-server-ip>:5000/my-image:latest
```

Зачем контейнеризовать приложения? Примеры Докерфайлов для основных языков программирования.

Контейнеризация приложений является одним из способов упаковки и доставки приложений с помощью контейнерных технологий, таких как Docker, Kubernetes и других. Контейнеризация приложений позволяет:

1. Повысить портативность: контейнеризация позволяет упаковать приложение и его зависимости в единый контейнер, который может быть запущен на любой платформе, где есть поддержка контейнеров. Это упрощает перенос приложений между средами разработки, тестирования и производства.
2. Улучшить масштабируемость: контейнеризация позволяет легко масштабировать приложение горизонтально путем запуска нескольких экземпляров контейнера на разных серверах или узлах кластера.
3. Облегчить управление зависимостями: контейнеризация позволяет упаковать все зависимости, необходимые для запуска приложения, в контейнер. Это упрощает управление зависимостями и их версиями, а также уменьшает вероятность конфликтов между зависимостями.
4. Улучшить безопасность: контейнеризация помогает обезопасить приложение, поскольку каждый контейнер работает в изолированном окружении, что уменьшает риск взаимодействия с другими приложениями или операционной системой.
5. Ускорить развертывание: контейнеризация ускоряет процесс развертывания приложения, так как контейнер может быть быстро запущен в любом месте, где есть поддержка контейнеров, без необходимости установки зависимостей и настройки окружения.

Контейнеризация приложений становится все более популярной в современной разработке и развертывании приложений, так как она упрощает управление приложениями, улучшает их масштабируемость и безопасность, и ускоряет их развертывание.

Python

Вот пример `Dockerfile` для Python с multi-stage сборкой и легким образом, который может быть адаптирован для большинства проектов:

```
# Сборочный этап
FROM python:3.9.9-slim-buster AS builder
WORKDIR /app
COPY requirements.txt .
RUN apt-get update && apt-get install -y --no-install-recommends gcc && \
    pip install --no-cache-dir -r requirements.txt && \
    apt-get remove -y gcc && apt-get clean && rm -rf /var/lib/apt/lists/*
COPY . .
RUN python setup.py bdist_wheel && \
    pip wheel --wheel-dir dist/ -r requirements.txt

# Этап запуска
FROM python:3.9.9-slim-buster
WORKDIR /app
COPY --from=builder /app/dist /app/dist
COPY --from=builder /usr/local/lib/python3.9/site-packages /usr/local/lib/python3.9/site-packages
COPY . .
CMD ["python", "app.py"]
```

В этом примере:

- Сначала мы используем официальный образ Python 3.9.9 slim-buster в качестве базового образа.
- Затем мы создаем новый образ на основе базового образа и называем его `builder`.
- На этапе `builder` мы устанавливаем зависимости, копируем код в рабочую директорию и собираем `wheel`-пакеты.
- Затем мы создаем новый образ на основе базового образа и называем его `run`.
- На этапе `run` мы копируем `wheel`-пакеты и установленные библиотеки из образа `builder`, а затем копируем оставшийся код.
- Команда `CMD` запускает приложение при запуске контейнера.

Этот `Dockerfile` использует multi-stage сборку для создания легковесного образа. На этапе `builder` мы устанавливаем зависимости и собираем `wheel`-пакеты, которые затем копируются на этап `run`. Это уменьшает размер окончательного образа, так как он не содержит ненужных пакетов, необходимых только для сборки.

PHP

Вот пример `Dockerfile` для PHP с multi-stage сборкой и легким образом, который может быть адаптирован для большинства проектов:

```
# Сборочный этап
FROM php: 7.4-cli AS builder
WORKDIR /app
COPY composer.json composer.lock ./
RUN apt-get update && apt-get install -y git && \
    curl -sS https://getcomposer.org/installer | php -- --install-dir=/usr/local/bin --
filename=composer && \
    composer install --no-dev --no-autoloader --no-scripts
COPY . .
RUN composer dump-autoload --no-dev --optimize && \
    composer run-script --no-dev post-install-cmd

# Этап запуска
FROM php: 7.4-alpine
WORKDIR /app
COPY --from=builder /app ./
CMD ["php", "src/index.php"]
```

В этом примере:

- Сначала мы используем официальный образ PHP 7.4 CLI в качестве базового образа.
- Затем мы создаем новый образ на основе базового образа и называем его `builder`.
- На этапе `builder` мы устанавливаем зависимости с помощью Composer, копируем код в рабочую директорию и создаем оптимизированные загрузчики классов.
- Затем мы создаем новый образ на основе базового образа Alpine Linux и называем его `run`.
- На этапе `run` мы копируем установленные библиотеки и оптимизированные загрузчики классов из образа `builder`, а затем копируем оставшийся код.
- Команда `CMD` запускает приложение при запуске контейнера.

Этот `Dockerfile` использует multi-stage сборку для создания легковесного образа. На этапе `builder` мы устанавливаем зависимости и создаем оптимизированные загрузчики классов, которые затем копируются на этап `run`. Это уменьшает размер окончательного образа, так как он не содержит ненужных пакетов, необходимых только для сборки.

С#

Вот пример `Dockerfile` для С# с multi-stage сборкой и легким образом, который может быть адаптирован для большинства проектов:

```
# Основной образ для сборки приложения
FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build

WORKDIR /app

# Копируем файлы проекта и восстанавливаем зависимости
COPY *.csproj .

RUN dotnet restore

# Копируем исходный код проекта и выполняем сборку
COPY . .

RUN dotnet publish -c Release -o out

# Образ для запуска приложения
FROM mcr.microsoft.com/dotnet/aspnet:6.0 AS runtime

WORKDIR /app

COPY --from=build /app/out .

# Задаём переменную среды ASPNETCORE_URLS
ENV ASPNETCORE_URLS=http://+:80

# Открываем порт 80 нашего приложения
EXPOSE 80

# Запускаем наше приложение
ENTRYPOINT ["dotnet", "MyProject.dll"]
```

Этот Dockerfile использует multi-stage сборку для уменьшения размера конечного образа. На первом этапе мы используем SDK-образ для восстановления зависимостей, сборки и публикации приложения в каталог `out`. Затем мы создаем новый образ из runtime-образа и копируем только необходимые файлы из первого образа (из каталога `out`) в новый. Также мы устанавливаем переменную среды `ASPNETCORE_URLS`, чтобы наше приложение слушало запросы на порту 80, и открываем этот порт для внешнего доступа.

Чтобы использовать этот Dockerfile, вам нужно поместить его в корневую папку вашего проекта. Затем выполните команду `docker build -t myimage .`, чтобы создать Docker-образ с именем `myimage`. Вы можете запустить этот образ, используя команду `docker run -p 80:80 myimage`. Это привяжет порт 80 внутри контейнера к порту 80 на хост-машине, и ваше приложение будет доступно по адресу `http://localhost:80`.

GoLang

Вот пример `Dockerfile` для GoLang с multi-stage сборкой и легким образом, который может быть адаптирован для большинства проектов:

```
# Основной образ для сборки приложения
FROM golang:latest AS build
WORKDIR /app

# Копируем файлы проекта и восстанавливаем зависимости
COPY go.mod go.sum ./
RUN go mod download

# Копируем исходный код проекта и выполняем сборку
COPY . .
RUN CGO_ENABLED=0 go build -o /app/app

# Образ для запуска приложения
FROM alpine:latest AS runtime
WORKDIR /app

# Копируем бинарник из предыдущего образа
COPY --from=build /app/app /app/app

# Задаём переменные среды
ENV PORT=80

# Открываем порт нашего приложения
EXPOSE $PORT

# Запускаем наше приложение
CMD ["/app/app"]
```

Этот Dockerfile использует multi-stage сборку для уменьшения размера конечного образа. На первом этапе мы используем Golang-образ для скачивания зависимостей, сборки и создания бинарного файла в каталоге `/app`. Затем мы создаем новый образ из Alpine-образа и копируем только необходимый бинарный файл из предыдущего образа в новый. Также мы устанавливаем переменную среды `PORT`, чтобы наше приложение слушало запросы на порту 80, и открываем этот порт для внешнего доступа.

Чтобы использовать этот Dockerfile, вам нужно поместить его в корневую папку вашего проекта. Затем выполните команду `docker build -t myimage .`, чтобы создать Docker-образ с именем `myimage`. Вы можете запустить этот образ, используя команду `docker run -p 80:80 myimage`. Это привяжет порт 80 внутри контейнера к порту 80 на хост-машине, и ваше

приложение будет доступно по адресу `http://localhost: 80`.

Как настроить CI/CD для бэкэнда?

















































Ниже представлен жизненный пример деплоя Python приложения с использованием оркестратора Docker Swarm, Docker Registry и тестами.

Для деплоя необходимо создать в корне проекта директорию `deploy`, в нее поместить `.env` файлы для каждого окружения, либо создать один общий и использовать его в нескольких окружениях по Environments в GitLab, а так же `docker-compose` файлы для каждого окружения. Еще будет необходим `.gitlab-ci.yml`.

Содержание `.env` файла:

```
POSTGRES_HOST=$POSTGRES_HOST
POSTGRES_PASSWORD=$POSTGRES_PASSWORD_DEV
POSTGRES_USER=$POSTGRES_USER_DEV
POSTGRES_DB=$POSTGRES_DB_DEV
POSTGRES_PORT=$POSTGRES_PORT
BOT_TOKEN=$BOT_TOKEN_DEV
CHAT_ID=$CHAT_ID_DEV
HOST_ROOM=$HOST_ROOM_DEV
```

Сами переменные, которые начинаются с \$, расположены в репозитории по пути Settings -> CI/CD -> Variables

Type	↑ Key	Value	Options	Environments	
Variable	BOT_TOKEN_DEV 	***** 	Expanded	All (default) 	
Variable	CHAT_ID_DEV 	***** 	Expanded	All (default) 	
Variable	HOST_ROOM_DEV 	***** 	Expanded	All (default) 	
Variable	POSTGRES_DB_DEV 	***** 	Expanded	All (default) 	
Variable	POSTGRES_DB_TEST 	***** 	Expanded	All (default) 	
Variable	POSTGRES_HOST_DEV 	***** 	Expanded	All (default) 	
Variable	POSTGRES_HOST_TEST 	***** 	Expanded	All (default) 	
Variable	POSTGRES_PASSWORD_DEV 	***** 	Masked, Expanded	All (default) 	
Variable	POSTGRES_PASSWORD_TEST 	***** 	Masked, Expanded	All (default) 	
Variable	POSTGRES_PORT_DEV 	***** 	Expanded	All (default) 	
Variable	POSTGRES_PORT_TEST 	***** 	Expanded	All (default) 	
Variable	POSTGRES_USER_DEV 	***** 	Expanded	All (default) 	

Содержание docker-compose.dev.yaml:

```
version: '3.9'
services:
  app:
    image: registry-server:5000/back:prod
    build:
      context: ..
    command: sh -c "python manage.py migrate && uvicorn clinica.asgi:application --host 0.0.0.0 --port 8080 --workers 4"
    depends_on:
      - db
    volumes:
```

- prod_static: /home/clinic/clinica/api/static
- prod_media: /home/clinic/clinica/api/media

env_file:

- .prod.env

ports:

- target: 8080
- published: 8090
- protocol: tcp

extra_hosts:

domain.com: 111.111.111.111
auth.domain.com: 111.111.111.111
chat.domain.com: 111.111.111.111

restart: always

deploy:

placement:

constraints:

- "node.labels.TAG==prod"

replicas: 4

update_config:

parallelism: 1
order: start-first
failure_action: rollback
delay: 10s

rollback_config:

parallelism: 0
order: stop-first

restart_policy:

condition: any
delay: 5s
max_attempts: 3
window: 120s

healthcheck:

test: curl --fail http://111.111.111.111:18090/api/ || echo 1
interval: 30s
timeout: 3s
retries: 12

db:

image: postgres:12.0-alpine
container_name: db_prod

```

volumes:
  - postgres_data: /var/lib/postgresql/data/
environment:
  - POSTGRES_USER=$POSTGRES_USER
  - POSTGRES_PASSWORD=$POSTGRES_PASSWORD
  - POSTGRES_DB=$POSTGRES_DB
ports:
  - target: 5432
    published: 54320
    protocol: tcp
env_file:
  - .prod.env
restart: always
deploy:
  placement:
    constraints:
      - "node.labels.TAG==prod"

```

```

volumes:
  postgres_data:
  prod_static:
  prod_media:

```

Этот docker-compose файл описывает два сервиса: `app` и `db`. Сервис `app` использует образ `registry-server:5000/back:prod`, который запускает приложение на основе фреймворка Django с помощью ASGI сервера Uvicorn. Команда запуска приложения включает миграцию базы данных и запуск Uvicorn с 4 рабочими процессами на 0.0.0.0:8080. Зависит от сервиса `db`. Также определяет переменные окружения из файла `.prod.env`, монтирует тома данных `prod_static` и `prod_media` внутри контейнера и настраивает порт на 8090. Определяет набор дополнительных хостов `extra_hosts` для использования внутри контейнера. Определяет конфигурацию разворачивания `deploy`, включающую размещение контейнеров на узле с меткой `prod`, использование 4 реплик, настройку обновления, задержку перед откатом и политику перезапуска.

Сервис `db` использует образ `postgres:12.0-alpine`, который предоставляет базу данных Postgres. Монтирует том `postgres_data` для хранения данных Postgres, определяет переменные окружения из файла `.prod.env`, настраивает порт на 54320 и определяет конфигурацию разворачивания `deploy`, также с меткой `prod`.

Наконец, файл определяет три тома данных: `postgres_data` для хранения данных Postgres, `prod_static` для хранения статических файлов приложения и `prod_media` для хранения медиафайлов, используемых приложением.

Содержание .gitlab-ci.yml:

```
stages:
  - linting
  - test
  - deploy

linting:
  image: registry.gitlab.com/mafda/python-linting
  stage: linting
  tags:
    - clinic
  script:
    - flake8 -j 2 --show-source --statistics --tee --output-file=flake_results.txt ||
NO_CONTAINER=1
  artifacts:
    paths:
      - flake_results_dev_2.txt
    expire_in: 1 week
    when: on_failure
  only:
    refs:
      - develop

test:
  image: docker:$DOCKER_VERSION
  stage: test
  tags:
    - clinic
  script:
    - docker compose -f ./deploy/docker-compose.test-dev.yaml down || NO_CONTAINER=1
    - docker compose -f ./deploy/docker-compose.test-dev.yaml up -d --build
    - docker exec -i $(docker ps | grep back_dev | awk '{print$1}') pytest -qs --reuse-db --
junitxml=report.xml
    - docker exec -i $(docker ps | grep back_dev | awk '{print$1}') cat
/home/clinic/clinica/report.xml > report.xml
    - docker compose -f ./deploy/docker-compose.test-dev.yaml down
  artifacts:
    when: always
    paths:
```



```

    - report.xml
  reports:
    junit: report.xml
  only:
    refs:
      - develop

deploy:
  image: docker:$DOCKER_VERSION
  stage: deploy
  tags:
    - clinic
  before_script:
    - docker login -u "$REGISTRY_USER" -p "$REGISTRY_PASSWORD" $REGISTRY_HOST
  script:
    - docker compose -f ./deploy/docker-compose.dev.yaml build
    - docker push $REGISTRY_HOST/clinic-back:dev
    - docker stack deploy --with-registry-auth -c ./deploy/docker-compose.dev.yaml back_dev
  environment:
    name: dev
    url: http://111.111.111.111:11111
  only:
    refs:
      - dev

```

Этот файл GitLab CI представляет пайплайн с тремя этапами: `linting`, `test` и `deploy`.

На этапе `linting` используется образ Docker `registry.gitlab.com/mafda/python-linting` для запуска инструмента статического анализа кода `flake8`, который проверяет соответствие кода Python стандарту PEP8. Результаты сохраняются в файл `flake_results.txt`, который доступен в качестве артефакта при неудачном завершении этапа.

На этапе `test` используется образ Docker `$DOCKER_VERSION` для запуска команд Docker Compose, которые тестируют приложение на стенде для разработки. Команда `docker compose down` удаляет существующие контейнеры, а затем команда `docker compose up` запускает их заново с помощью файла конфигурации `docker-compose.test-dev.yaml`. Затем запускаются тесты с помощью Pytest и создается отчет в формате JUnit XML. Контейнеры удаляются после завершения тестирования, а отчет сохраняется как артефакт.

На этапе `deploy` используется образ Docker `$DOCKER_VERSION` для создания и развертывания образа Docker на стенде `dev`. Образ собирается из файла конфигурации `docker-compose.dev.yaml`, загружается в репозиторий Docker и развертывается на стенде `dev` с помощью команды `docker stack deploy`. Перед выполнением скрипта `before_script`

выполняется вход в репозиторий Docker с помощью учетных данных из переменных окружения. Этап `deploy` выполняется только при изменении ветки `dev`.