KARPOV.COURSES >>>
КОНСПЕКТ

> Конспект >7 урок > Сценарий развёртывания

> Оглавление

> Комментарий про развёртывание

В 5-й лекции мы рассмотрели deploy в Swarm и увидели, что он многое даёт "из коробки". При этом то, чего в нём нет, можно просто реализовать с помощью Bash.

Деплой в Swarm или Kubernetes — это хорошо, быстро и надёжно, но не всегда возможно, поскольку:

1. Далеко не всегда доступны те или иные инструменты: Docker, Swarm или Kubernetes.
2. Если у вас какой-нибудь нетрадиционный случай развёртки, то даже возможностей Kubernetes кластера может не хватить, а значит, что-то придётся доделывать руками через Bash. Например, такое может случиться, когда у вас есть тысячи машин по всей стране.
3. Корпоративные политики безопасности тоже не делают задачу проще. Если у вас в компании открытый контур, то это замечательно, и вы можете использовать любые инструменты. В закрытом контуре всё уже не так просто — вы зависите от команды DevOps'ов. В изолированном контуре, скорее всего, придётся деплоить вообще всё вручную, т.к. Swarm там может быть вовсе запрещён.

> Сценарии развёртывания

Давайте вспомним сценарии развёртывания, которые мы уже затрагивали, и затем рассмотрим новые.

> Rolling update

Этот сценарий мы использовали в 5-й лекции, когда мы деплоили в Swarm. Суть его заключается в том, что мы обновляем не все реплики и делаем это поочерёдно. То есть мы можем разворачивать новую реплику сервиса и затем гасить одну из старых реплик, тем самым сохраняя количество активных реплик неизменным. Или можем сначала сворачивать старую реплику, а затем поднимать новую.

Rolling update обладает следующими свойствами:

- Постепенное обновление всего приложения (в рамках временного окна, в которое происходит обновление).
- Относительная безопасность развёртывания. Если, например, в Swarm настроены rollback и healthcheck'и, то при обновлении на нерабочую версию Swarm не даст вам развернуть более `n` (которое вы сами указываете) реплик, а весь трафик пойдёт на старые рабочие реплики.
- В рамках временного окна работают разные версии приложения (пользователи могут попадать попеременно то на новую версию, то на старую). Это необходимо иметь в виду, если, например, вы вместе с обновлением меняете

политику взаимодействия с БД. Тогда в случае обновления и приложения, и БД старые реплики перестанут работать. Если же обновлять БД после приложения, то новые реплики не будут работать.

Скорее всего, из-за вышеизложенных свойств Rolling update не будет оптимальным сценарием для обновления каждого компонента вашего приложения.

> Blue / Green

Green — псевдоним для текущей работающей версии. Blue — псевдоним новой версии, которую вы разворачиваете параллельно с работающей Green и независимо от неё. Когда с помощью различных тестов вы на 100% убедились, что Blue часть полностью рабочая, вы можете переключить трафик с Green части на Blue часть. Отличие от Rolling update состоит в отсутствии временного периода, когда работают обе версии приложения. Сам переход практически моментальный, хотя и требует больше доступных вычислительных ресурсов. Стоит отметить, что далеко не всегда после включения Blue части реплик отключают Green часть: если вдруг на новых репликах что-то пойдёт не так, то почти всегда можно будет довольно быстро переключить трафик обратно на Green часть. Однако в случае обновления БД (миграции) придётся откатывать и его.

> Canary

Canary — нечто среднее между первыми двумя. При последовательном обновлении реплик/воркеров мы делаем большие паузы между обновлениями, во время которых ведётся наблюдение за новыми развёрнутыми версиями, чтобы обнаруживать какие-то скрытые проблемы на маленьком количестве пользователей (чтобы потенциально сломанную версию не получили сразу все пользователи).

> Bash

В 6-й лекции в CI/CD-задаче мы использовали `ssh`, который remote-ssh команде позволял получать ключ, благодаря которому эта команда выполнялась на удалённом сервере.

> ssh

Если ваш сценарий деплоя подразумевает многократное выполнение удалённых ssh-команд, то рекомендуется сделать доступ на удалённые машины беспарольным с помощью пары RSA ключей (приватного и публичного).

Рекомендуем ознакомиться с документацией команды `ssh-keygen`. Сама генерация пары происходит по команде `ssh-keygen -t rsa`.

Также рекомендуем отключить доступ по паролю. Для этого нужно поменять две строчки в файле конфигурации сервиса ssh и перезапустить сервис. Файл конфигурации сервиса находится по адресу `/etc/ssh/sshd_config`. Строчка, отвечающая за пароль: `PasswordAuthentication yes`. Перезапуск сервиса делается через `systemctl restart sshd`.

`sudo` тоже можно сделать беспарольным. Для этого необходимо выполнить следующую команду: `sudo echo 'admin ALL=(ALL) NOPASSWD:ALL' > /etc/sudoers.d/admin`

Для ускорения входа на удалённые машины также рекомендуем использовать `~/.ssh/config` файл, в котором можно прописать информацию для входа:

```
Host host_name
    HostName xxx.xxx.xxx.xxx
    Port 0000
    User username
    IdentityFile ~/.ssh/host_rsa_key
```

Тогда, чтобы войти на удалённую машину по `ssh`, необходимо всего лишь сделать `ssh host_name`

> Доставка кода и артефактов

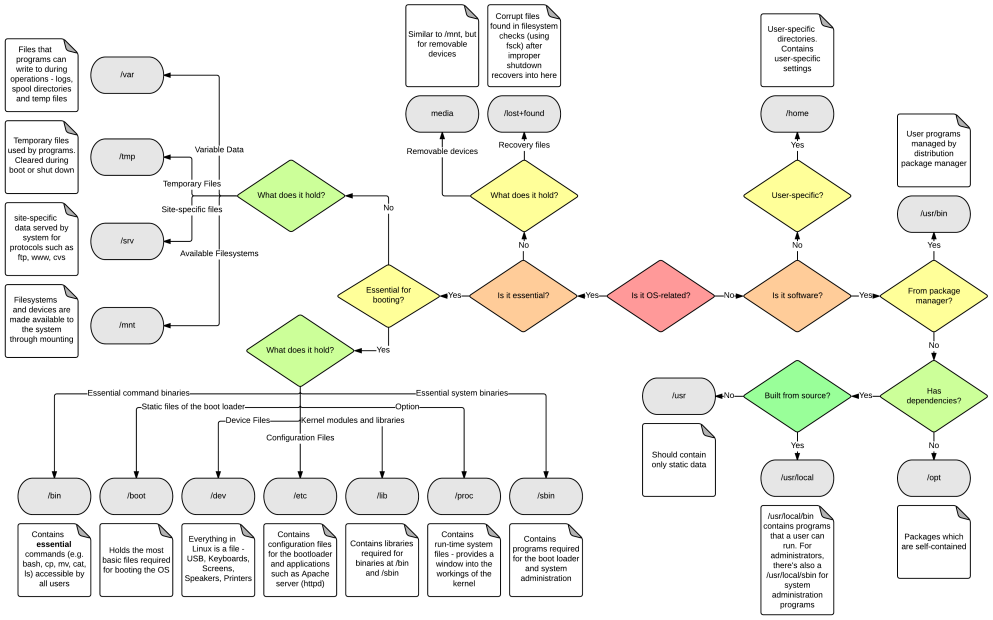
Теперь, когда мы разобрались с доступом на машины, давайте подумаем, как доставлять код и артефакт (от artifactory).

1. Можно в пайплайне грузить артефакты в s3 Artifactory, а уже при запуске приложения реализовать логику скачивания из s3. Код в s3 загружать не нужно, т.к. он получается командой `pull` из Git. Если вы собираете бинарный файл, то тоже используете Artifactory. Далее через `ssh`, как в 6-й лекции, запускаете скрипт, который удалённо пуллит, а потом и запускает.
2. Можно напрямую загружать артефакты и код, используя `scp` и `sftp`
3. Часто, особенно для больших файлов, из-за некоторых преимуществ используют `rsync` с похожим синтаксисом. В дополнительных материалах конспекта будет ссылка, объясняющая почему так делают.
4. Можно устанавливать пакеты.

5. Для хранения артефактов модели можно использовать Git LFS. GitLab будет выступать в качестве прокси. Такой метод может подтормаживать при большой нагрузке на GitLab.

В том как доставлять мы разобрались, теперь посмотрим куда можно доставлять код.

В ОС для большинства файлов есть характерные места хранения. Если вы их знаете, то даже с системой работаете быстрее и увереннее. На приведённой ниже схеме показано, где и что следует хранить:



Например, мы можем держать модели в:

- `/opt/models/`
- `/var/models/`
- `/storage/models/` (можно создать такую папку или попросить администратора установить внешний диск)

Доставлять код в домашнюю директорию — не лучшая идея. Можно доставлять, например, в `/var/projects/project_name`. Заодно это будет и Git-репозиторием, который можно будет checkout'ить в запускающем shell скрипте. При запуске в shell'е можно как использовать абсолютный путь, так и заходить в директорию с

кодом, используя относительный путь. Лучше всего создать ссылку на запускающий скрипт.

> Запуск кода

В Linux можно сделать так, чтобы ваш скрипт вызывался как стандартная команда Linux. Для этого необходимо сделать несколько вещей:

1. Разрешить исполнять файл с помощью команды `chmod +x script.py`
2. В начало исполняемого файла добавить дескриптор для интерпретатора вида `#!/usr/bin/python3`
3. Переименовать script.py-> script с помощью команды `mv script.py script`
4. Создать символьную ссылку на этот скрипт в папке /usr/local/bin (она есть в `PATH`) с помощью команды `ln -s /var/projects/project_name/scritp /usr/local/bin/script`

Теперь вы сможете вызывать ваш скрипт из любого места с помощью простой команды `script`, т.к. ссылка на него есть в одной из директорий, прописанных в переменной окружения `PATH`.

Следует отметить, что Bash находится близко к ядру системы, и поэтому довольно часто используется для деплоя приложений. Это довольно мощный инструмент, хотя и не всегда удобный из-за чувствительности синтаксиса к неочевидным вещам (например к пробелам). Но тем не менее, Bash с нами надолго.

> Systemd и nginx

В 3-й лекции мы упомянули systemd. Мы сошлись на том, что почти всегда он нам не подходит, потому что в нём нет сетевых абстракций и в нём неудобно масштабировать.

Тем не менее некоторые вещи запускаются как systemd сервис, например nginx. Давайте для начала его запустим.

Устанавливается nginx через `apt-get install nginx`. После установки он автоматически запускается. В этом можно убедиться с помощью следующей команды: `systemctl status nginx`

```
root@kcloud-production-user-6-vm-12:~# systemctl status nginx
● nginx.service - A high performance web server and a reverse proxy server
     Loaded: loaded (/lib/systemd/system/nginx.service; enabled; vendor preset: enabled)
     Active: active (running) since Wed 2021-07-07 14:57:23 CEST; 42s ago
       Docs: man:nginx(8)
   Main PID: 128608 (nginx)
      Tasks: 2 (limit: 2286)
     Memory: 4.1M
     CGroup: /system.slice/nginx.service
             ├─128608 nginx: master process /usr/sbin/nginx -g daemon on; master_process on;
             └─128609 nginx: worker process
```

Он также появился в списке всех сервисов systemd. Проверить это можно с помощью команды: `systemctl list-units --type=service | grep nginx`

```
root@kcloud-production-user-6-vm-12:~# ^C^C
root@kcloud-production-user-6-vm-12:~# systemctl list-units --type=service | grep nginx
  nginx.service                                loaded active running A high performance web server and a reverse proxy server
```

Ещё один способ проверки: `systemctl list-unit-files`

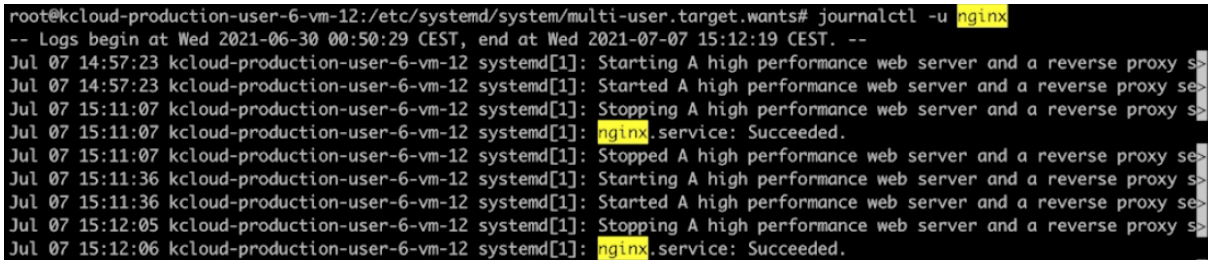
Unit — ключевое понятие в systemd, это обсуждалось на 3-й лекции. Самый частый тип юнита, с которым мы будем сталкиваться — юнит типа service, которым и является nginx. Юниты описываются в специальных файлах формата systemd. Например, после установки nginx в соответствующей директории (`/etc/systemd/system/multi-user.target.wants/`) появился файл `nginx.service`

```
root@kcloud-production-user-6-vm-12:/etc/systemd/system/multi-user.target.wants# ls
atd.service                 irqbalance.service          rsync.service
console-setup.service       networkd-dispatcher.service rsyslog.service
cron.service                nginx.service               ssh.service
dmesg.service               ondemand.service            systemd-resolved.service
grub-initrd-fallback.service pollinate.service          ufw.service
hc-net-scan.service         remote-fs.target            unattended-upgrades.service
```

На самом деле в эту папку сервисы попадают не просто после установки, а только если они enabled, т.е. не просто включены (active), а именно доступны для включения. Если же выполнить команду `systemclt disable nginx`, то файл `nginx.service` пропадёт из данной директории. Сам список сервисов доступен в директории `/lib/systemd/system`.

Запуск, остановка и перезапуск сервисов производится командами `systemctl start nginx`, `systemctl stop nginx` и `systemctl restart nginx`.

Просмотр логов сервиса возможен по команде `journalctl -u nginx`



Команды довольно простые. Давайте посмотрим на Unit-файл сервиса.

```
[Unit]
Description=The NGINX HTTP and reverse proxy server
Documentation=man:nginx(8)
After=network.target

[Service]
Type=forking
PIDFile=/run/nginx.pid
ExecStartPre=/usr/sbin/nginx -t -q -g 'daemon on; master_process on;'
ExecStart=/usr/sbin/nginx -g 'daemon on; master_process on;'
ExecReload=/usr/sbin/nginx -g 'daemon on; master_process on;' -s reload
ExecStop=-/sbin/start-stop-daemon --quite --stop --retry QUIT/5 --pidfile /run/nginx.pid
TimeoutStopSec=5
Killmode=mixed

[Install]
WantedBy=multi-user.target
```

Первая секция — `[Unit]`, она описывает сервис. Ключевое слово `After` указывает на то, после чего должен подняться этот сервис. В данном случае сервис nginx будет подниматься сразу после готовности службы network.

Вторая секция — `[Service]`, основная для сервиса.

Третья секция — `[Install]`

Далее мы отдельно рассмотрим, что означают все эти директивы.

> Unit-файлы

Здесь приведём краткую выжимку из документации, чтобы можно было быстро разобраться, что означают отдельные поля в unit-файлах.

```
[Unit]
# Описание сервиса
Description=X, service with model for production

# Через пробел можно добавлять другие таргеты. Таргеты также можно создавать и самим
After=network.target nss-lookup.target syslog.target
# С чем наш сервис конфликтует и не может стартовать одновременно
```

```
Conflicts=...
# И до каких сервисов
Before=...


[Service]
# Пользователи и группы, которые запускают сервис
User=root
Group=root

# Тип сервиса
Type=< simple | exec | forking | oneshot | dbus | notify | idle >
# "simple | exec" - очень похожи: процессы юнита запускаются сразу после того,
# как создается процесс сервиса, но exec, в отличие от simple, ждет, когда форкнутый процесс
# стартанет, и поэтому может детектировать failure. Это важно для follow-up юнитов
# "forking" - systemd считает службу запущенной после того, как процесс разветвляется
# с завершением родительского процесса. Используется для запуска классических демонов
# за исключением тех случаев, когда в таком поведении процесса нет необходимости.
# "oneshot, notify, idle, dbus" - нерелевантно

# NB: Для сервисов, которые работают долго, рекомендуется использовать "Type=simple",
# т.к. это быстро и просто. Но он не позволяет пробрасывать ошибки о старте, а значит,
# не годится, если нужно соблюдать очередь запусков.
# Exec подходит, когда нужно убедиться, что исполняемый файл смог запуститься.

# Для "Type=fork", чтобы systemd точно знал, какой процесс ему нужен после форка
# юнит-процесса и смерти его родителя
PIDFile=/run/x-imodel.pid

Restart=< no | on-success | on-failure | on-abnormal | on-watchdog | on-abort | always >
# Это обсуждалось в предыдущих лекциях. Как правило, мы будем использовать <always>

# Можно передавать переменные окружения или даже путь до файла с такими переменными
Environment=ONE='one' "TWO='two two' too" THREE=
EnvironmentFile=...

# Назначение следующих команд довольно понятно. Стоит отметить, что команда ExecStart
# не будет выполнена, если ExecStartPre выполнится с ошибкой.
# ExecStartPost же всегда, вне зависимости от того, как завершится ExecStart
ExecStartPre="echo first ExecStartPre"
ExecStartPre="echo second ExecStartPre"
ExecStart=kek  # Выполнится, только если оба ExecStartPre имели exit code = 0
ExecStartPost="echo first ExecStartPost"


# ExecStop будет работать только тогда, когда ExecStart завершится успешно.
ExecStop=kill -s SIGTERM $(cat /run/x_%imodel.pid) # Будет выполняться асинхронно !
# Специфика завершения в systemd заключается в том, что оно выполняется в 2 этапа:
# сначала выполняется ExecStop, а потом процессам, которые по какой-то причине
# не завершились, направляется KillSignal, который и можно указать
KillMode=   # Можно указать "режим" удаления (все / не все / все, кроме главного и т.д.)
KillSignal= # и посылаемый сигнал
# ExecStopPost выполнится вне зависимости от результата ExecStop
ExecStopPost="echo some post-mortem clean-up operation"

# ExecReload предназначена для плавной перезагрузки, особенно если используется UWSGI,
# который позволяет плавно перезапускать воркеры с zero-downtime. Команда ниже позволит
# это сделать по SIGHUP-сигналу.
ExecReload=kill -HUP $MAINPID # Будет выполняться асинхронно !
RestartSec=
```

```
TimeoutStartSec=
TimeoutStopSec=
RuntimeMaxSec=

#Помимо этого перед командами можно ставить префиксы
"-" - # failure коды запишутся, но не будут иметь эффекта
# (будет выглядеть как успешное завершение)
# Остальные нерелевантны

[Install]
# Если ваш сервис ничего не Want и ничего не Require,
# то ваш сервис не будет запускаться автоматически
WantedBy=multi-user.target
```

> Масштабирование через systemd

Давайте напишем простое Flask-приложение (наш исполняемый файл script):

```
#!/usr/bin/python3
import sys
import time
import flask

app = flask.Flask(__name__)
port = int(sys.argv[1])

@app.route('/')
def index():
    print(f'Hello from port: {port}')

app.run('0.0.0.0', port)
```

Также создадим простейший unit-файл `x-model.service`:

```
[Unit]
Description=X, service with model for production
After=network.target

[Service]
Type=exec
Restart=always
ExecStart=script 5000

[Install]
WantedBy=multi-user.target
```

Затем этот файл мы помещаем в директорию `/lib/systemd/system/` и далее enable'им его командой: `systemctl enable x-model`. Запустить его можем командой `systemctl start x-model`.

Если взглянуть на папку `/lib/systemd/system/`, то можно заметить, что некоторые unit-файлы имеют в своём названии знак `@`. Такие файлы называются файлами-шаблонами. Само по себе это означает, что мы сможем применять этот unit-файл для запуска нескольких сервисов, используя `@` как параметр (переменную подстановки) и передавая его в unit-файл.

Скопируем и исправим наш unit-файл, превратив его в unit-файл-шаблон `x-model@.service`:

```
[Unit]
Description=X, service with model for production on port %i
After=network.target

[Service]
Type=exec
Restart=always
ExecStart=script %i

[Install]
WantedBy=multi-user.target
```

Передача параметра осуществляется через `%i` (можно также через `%I`, но в таком случае он передастся в первоначальном виде, т.е. с кавычками и т.д.)

Теперь, сделав enable и запуск сервиса через `systemctl enable x-model@5001`, `systemctl start x-model@5001`, мы увидим, что сервис стал доступным для включения и запустился. Соответственно, можно сделать enable несколько приложений на разных портах.

Управлять отдельными репликами независимо довольно трудно, поэтому их можно объединить в systemd.target. Они нужны для объединения в группу. Для это создадим target-файл в директории `/etc/systemd/system/`:

```
[Unit]
Description=Workers
Wants=x-model@5001.service x-model@5002.service x-model@5003.service

[Install]
WantedBy=multi-user.target
```

При этом шаблонный unit-файл необходимо немного изменить, указав, что сервисы, получаемые из шаблона, являются частью группы:

```
[Unit]
Description=X, service with model for production on port %i
After=network.target
PartOf=x-model.target
```

```
[Service]
Type=exec
Restart=always
ExecStart=script %i

[Install]
WantedBy=multi-user.target
```

Теперь через `systemctl start x-model.target` можно запустить сразу всю группу (при условии, что сервисы мы до этого остановили).

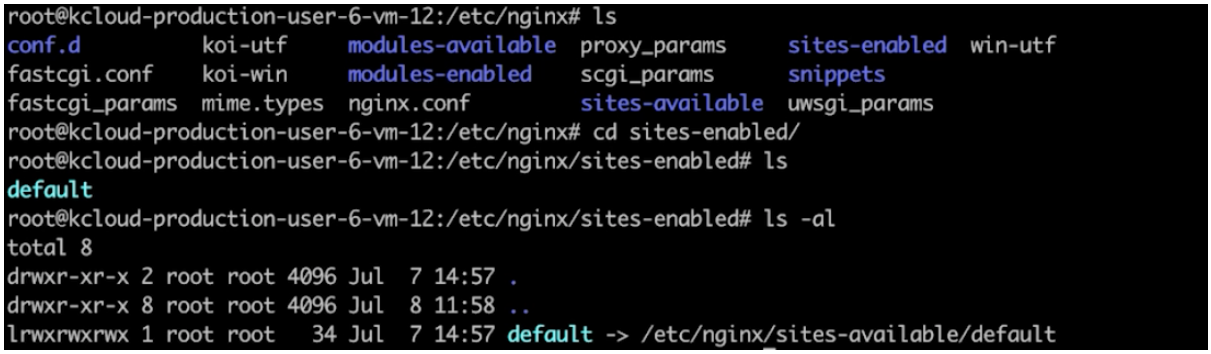
> Конфигурация nginx

Nginx можно считать балансировщиком и reverse-прокси (proxy маршрутизирует трафик из внутреннего контура во внешний, а reverse proxy, соответственно, наоборот). В нашем же случае трафик будет приходить на nginx извне и перенаправляться на реплики, распределяясь более-менее равномерно между репликами (балансировщик).

Стоит помнить архитектуру nginx (3-я лекция) и понимать, что сам nginx является точкой отказа. В больших компаниях с высоким трафиком балансировкой занимается специализированное устройство, которое делает это на уровне "железа".

Мы уже видели unit-файлы nginx, который описывает systemd сервис, но это конфигурация именно самого systemd сервиса, а сам nginx имеет свои config-файлы. Главный конфиг-файл находится по адресу `/etc/nginx/nginx.conf`. Этот конфигурационный файл мы рекомендуем менять только в случае, если вы точно знаете, что делаете и зачем это нужно.

В директориях `sites-enabled` и `sites-available` находятся пользовательские конфигурации того, как следует балансировать и проксировать. Если очень грубо, то схема похожа на то, как хранились unit-файлы в systemd. В `sites-available` мы храним все файлы конфигураций, а в `sites-enabled` — символьные ссылки на конфиги из `sites-available`. В ней же находится ссылка на конфигурацию по умолчанию `default`.



Давайте создадим файл `go.conf`, который будет пробрасывать нас на google.ru:

```
server {
        listen 80;
        location /go {
                proxy_pass https://www.google.ru/;
                }
}
```

Теперь необходимо создать "мягкую" ссылку до папки sites-enabled с помощью команды: `ln -s /etc/nginx/sites-enabled/go.conf /etc/nginx/sites-available/go.conf`

Также нужно отключить ссылку на конфигурацию по умолчанию: `unlink default`

Чтобы nginx "подхватил" новую конфигурацию, необходимо "мягко" перезагрузить nginx с помощью `systemctl reload nginx`

Чтобы теперь заменить проброс на наше приложение, необходимо немного изменить conf-файл:

```
server {
        listen 80;
        location / {
                proxy_pass http://localhost:5001;
                }
}
```

Обратите внимание, что мы также изменили "ручку" прокси, поскольку в нашем приложении ручка находится именно по такому корневому адресу, а nginx только подставляет `location` к `proxy_pass`.

Можно попытаться запустить nginx вместе с этими двумя конфигами, но работать будет только один из них, т.к. оба они "слушают" один и тот же порт и поэтому конфликтуют. Чтобы они оба смогли работать, у одного из них необходимо сменить порт.

> Балансировка трафика

Теперь нас, конечно же, интересует вопрос, как объявить серверы таким образом, чтобы балансировать трафик между ними. Делается это через ключевое слово `upstream`, и мы просто прописываем адреса и порты — но не протокол, по которому осуществляется взаимодействие.

```
upstream x-model{
        server localhost:5001 weight=1;
        server localhost:5002;
        server localhost:5003;
}

server {
        listen 80;
        location / {
                proxy_pass http://x-model;
                }
}
```

Теперь всё, что осталось сделать, — заменить в `proxy_pass` конкретный хост и порт на группу `upstream`.

Сейчас, каждый раз заходя на веб-сервер, мы будем видеть, что он последовательно отправляет нас на один из трёх серверов. Более того, мы можем устанавливать вес (`weight`), с которым будет выбран тот или другой сервер.

Теперь попробуем вставить несуществующий сервер `localhost:2000` в список серверов и посмотрим, как поведёт себя nginx. Самый приятный момент заключается в том, что он ни разу не отправит на неработающий сервер.