



Docker:

Особенности использования
при разработке на разных языках
программирования



Разработка и Docker

Ожидаемо есть проблемы с legacy





Разработка и Docker

Ожидаемо есть проблемы с legacy

Но если начинать проект с нуля – есть best practices,





Разработка и Docker

Ожидаемо есть проблемы с legacy

Но если начинать проект с нуля – есть best practices,

а именно – направленные на:





Разработка и Docker

Ожидаемо есть проблемы с legacy

Но если начинать проект с нуля – есть best practices,

а именно – направленные на:

упрощение отладки при разработке;





Разработка и Docker

Ожидаемо есть проблемы с legacy

Но если начинать проект с нуля – есть best practices,

а именно – направленные на:

упрощение отладки при разработке;

сокращение времени на сборку образа;





Разработка и Docker

Ожидаемо есть проблемы с legacy

Но если начинать проект с нуля – есть best practices,

а именно – направленные на:

- упрощение отладки при разработке;
- сокращение времени на сборку образа;
- уменьшение размера получившегося образа.





Аксиомы разработки



Монтирование каталога с кодом в контейнер при отладке.





Аксиомы разработки

- Монтирование каталога с кодом в контейнер при отладке.

- Разделение команд по установке зависимостей и копированию исходников





Аксиомы разработки

- Монтирование каталога с кодом в контейнер при отладке.

- Разделение команд по установке зависимостей и копированию исходников

- Копирование в образ самых часто изменяемых элементов в последнюю очередь





Аксиомы разработки

- Монтирование каталога с кодом в контейнер при отладке.
- Разделение команд по установке зависимостей и копированию исходников
- Копирование в образ самых часто изменяемых элементов в последнюю очередь
- Использование многоступенчатой сборки





Аксиомы разработки

- Монтирование каталога с кодом в контейнер при отладке.
- Разделение команд по установке зависимостей и копированию исходников
- Копирование в образ самых часто изменяемых элементов в последнюю очередь
- Использование многоступенчатой сборки
- Унификация образа для разных окружений при помощи [CMD](#) / [ENTRYPOINT](#)





Python



Точные версии пакетов везде – в поле **FROM**, в **requirements.txt**





Python

- Точные версии пакетов везде – в поле **FROM**, в **requirements.txt**
- **COPY requirements.txt** и ставим зависимости, а второй **COPY** – для кода (либо bind mount)





Python

- Точные версии пакетов везде – в поле **FROM**, в **requirements.txt**
- **COPY requirements.txt** и ставим зависимости, а второй **COPY** – для кода (либо bind mount)
- Переменная **PYTHONUNBUFFERED: 1** для упрощения отладки





Python

- Точные версии пакетов везде – в поле **FROM**, в **requirements.txt**
- COPY requirements.txt** и ставим зависимости, а второй **COPY** – для кода (либо bind mount)
- Переменная **PYTHONUNBUFFERED: 1** для упрощения отладки
- Миграция БД и запуск приложения – это разные операции (для разделения – **ENV**)

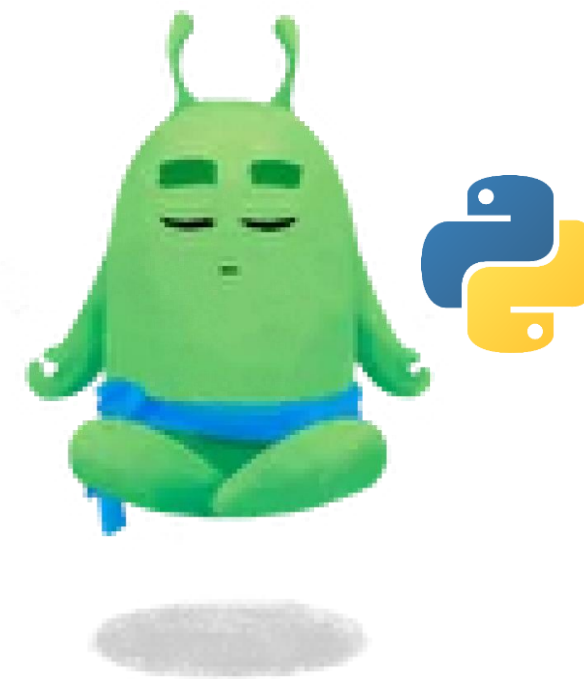




Ещё немного о Python



Нужен virtualenv - в Dockerfile используем **ENV**





Ещё немного о Python



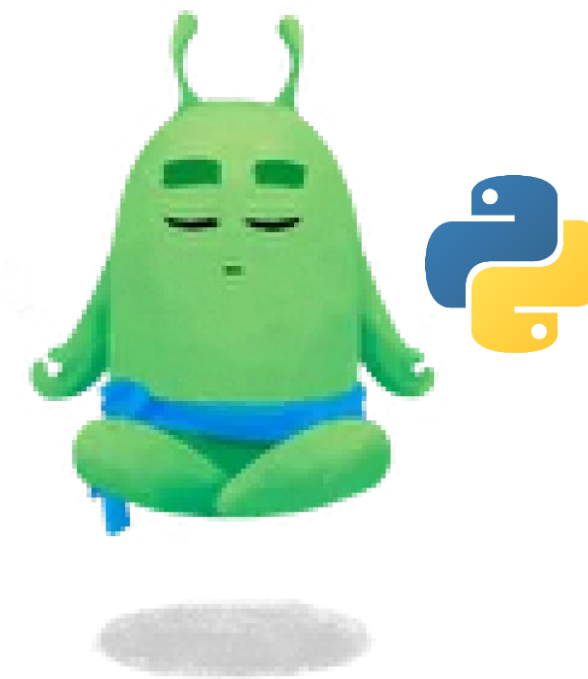
Нужен virtualenv - в Dockerfile используем ENV

Памятка по gunicorn:

- `gunicorn --worker-tmp-dir /dev/shm` – перенос tmp в RAM;



Ещё немного о Python



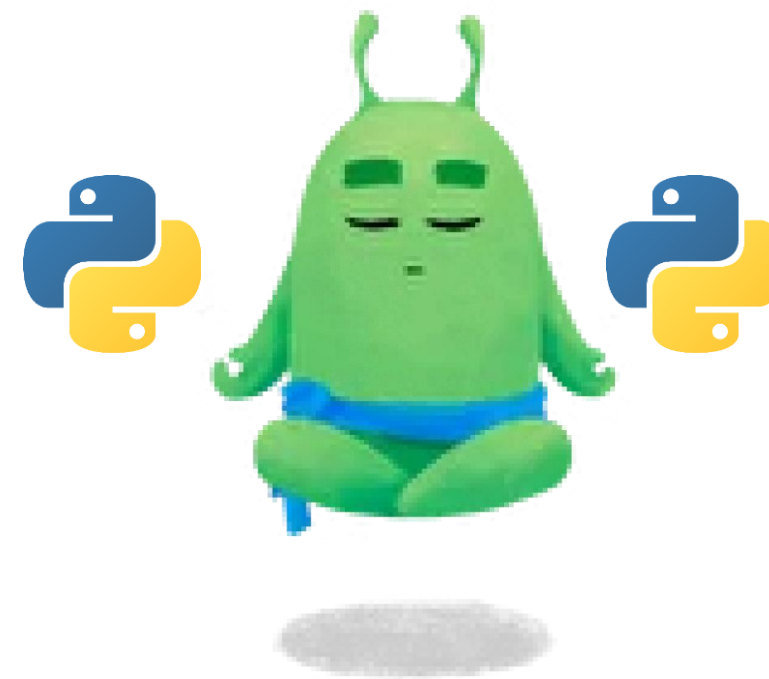
Нужен virtualenv - в Dockerfile используем ENV

Памятка по gunicorn:

- `gunicorn --worker-tmp-dir /dev/shm` – перенос tmp в RAM;
- `gunicorn --workers=2 --threads=4 --worker-class=gthread` – два воркера, если есть шанс медленных ответов;



Ещё немного о Python



Нужен virtualenv - в Dockerfile используем ENV

Памятка по gunicorn:

- `gunicorn --worker-tmp-dir /dev/shm` – перенос tmp в RAM;
- `gunicorn --workers=2 --threads=4 --worker-class=gthread` – два воркера, если есть шанс медленных ответов;
- `gunicorn --log-file=-` - пишем логи в STDOUT/STDERR.



Python Dockerfile, два примера:



```
FROM python:3.8 AS builder
COPY requirements.txt .

# Устанавливаем зависимости в локальный каталог
# (например /root/.local)
RUN pip install --user -r requirements.txt

# Второй шаг, уже без имени
FROM python:3.8-slim
WORKDIR /code

# Копируем только зависимости из образа,
# полученного на первом шаге
COPY --from=builder /root/.local/bin /root/.local
COPY ./src .

# Устанавливаем переменные окружения
ENV PATH=/root/.local:$PATH
ENV PYTHONUNBUFFERED=1

CMD [ "python", "./server.py" ]
```



Python Dockerfile, два примера:



```
FROM python:3.8 AS builder
COPY requirements.txt .
```

```
# Устанавливаем зависимости в локальный каталог
# (например /root/.local)
RUN pip install --user -r requirements.txt
```

```
# Второй шаг, уже без имени
FROM python:3.8-slim
WORKDIR /code
```

```
# Копируем только зависимости из образа,
# полученного на первом шаге
COPY --from=builder /root/.local/bin /root/.local
COPY ./src .
```

```
# Устанавливаем переменные окружения
ENV PATH=/root/.local:$PATH
ENV PYTHONUNBUFFERED=1
```

```
CMD [ "python", "./server.py" ]
```

```
FROM python:3.8-slim-buster
```

```
# Устанавливаем переменные окружения для
«активации» виртуального окружения
ENV VIRTUAL_ENV=/opt/venv
RUN python3 -m venv $VIRTUAL_ENV
ENV PATH="$VIRTUAL_ENV/bin:$PATH«
ENV PYTHONUNBUFFERED=1
```

```
# Устанавливаем зависимости
COPY requirements.txt .
RUN pip install -r requirements.txt
```

```
# Запускаем приложение, установленные переменные
# окружения из шагов выше всё ещё действуют
COPY myapp.py .
CMD ["python", "myapp.py"]
```



Java



Java до 8u121 не знала про ограничения cgroups





Java

Java до 8u121 не знала про ограничения cgroups

...а сейчас знает, поэтому выставляйте квоты при запуске





Java

Java до 8u121 не знала про ограничения cgroups

...а сейчас знает, поэтому выставляйте квоты при запуске

Для сборки - Maven, для запуска – openjdk (или Zulu)





Java



Java до 8u121 не знала про ограничения cgroups

...а сейчас знает, поэтому выставляйте квоты при запуске

Для сборки - Maven, для запуска – openjdk (или Zulu)

Подключайте тома для переиспользования кэша сборок



Java



Java до 8u121 не знала про ограничения cgroups

...а сейчас знает, поэтому выставляйте квоты при запуске

Для сборки - Maven, для запуска – openjdk (или Zulu)

Подключайте тома для переиспользования кэша сборок

Ограничивайте размер кучи (`-Xmx`, `-XX:MaxRAMPercentage`)



Java



Java до 8u121 не знала про ограничения cgroups

...а сейчас знает, поэтому выставляйте квоты при запуске

Для сборки - Maven, для запуска – openjdk (или Zulu)

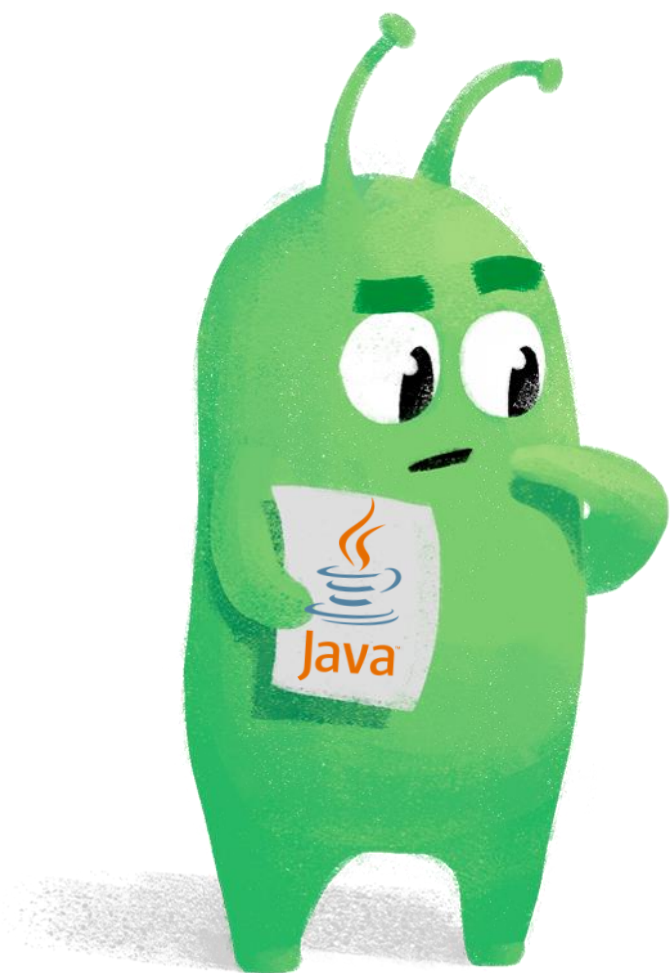
Подключайте тома для переиспользования кэша сборок

Ограничивайте размер кучи (`-Xmx`, `-XX:MaxRAMPercentage`)

... а контейнеру давайте на 25% больше



Пример Java Dockerfile



```
# Первый шаг сборки
FROM maven:3.6-jdk-8-alpine AS builder

ARG USER_HOME_DIR="/root"
VOLUME "$USER_HOME_DIR/.m2"
ENV MAVEN_CONFIG "$USER_HOME_DIR/.m2"
WORKDIR /app

# Копируем зависимости и устанавливаем их
COPY pom.xml .
RUN mvn -e -B dependency:resolve

# Сборка jar-файла
COPY src ./src
RUN mvn -e -B package

# Второй шаг, копирование приложения и его запуск
FROM openjdk:8-jre-alpine
COPY --from=builder /app/target/app.jar /
CMD ["java", "-jar", "/app.jar"]
```



PHP

Скорее всего процессов в контейнере будет больше одного





PHP

- Скорее всего процессов в контейнере будет больше одного
- Nginx и apache пишут логи в файлы





PHP

- Скорее всего процессов в контейнере будет больше одного
- Nginx и apache пишут логи в файлы
- Вот поэтому PHP в Docker – редкий случай





PHP Dockerfile (на примере Laravel):



```
FROM php:7.2-fpm

COPY composer.lock composer.json /var/www/
WORKDIR /var/www
RUN apt-get update && apt-get install -y \
    build-essential \
    libpng-dev \
    libjpeg62-turbo-dev \
    libfreetype6-dev \
    locales \
    zip \
    jpegoptim optipng pngquant gifsicle \
    vim \
    unzip \
    git \
    curl

RUN apt-get clean && rm -rf /var/lib/apt/lists/*

RUN docker-php-ext-install pdo_mysql mbstring zip
exif pcntl
```

```
RUN docker-php-ext-configure gd --with-gd --with-
freetype-dir=/usr/include/ --with-jpeg-
dir=/usr/include/ --with-png-dir=/usr/include/
RUN docker-php-ext-install gd

RUN curl -sS https://getcomposer.org/installer |
php -- --install-dir=/usr/local/bin --
filename=composer

RUN groupadd -g 1000 www
RUN useradd -u 1000 -ms /bin/bash -g www www

COPY . /var/www

COPY --chown=www:www . /var/www

USER www

EXPOSE 9000
CMD ["php-fpm"]
```



Ruby



Типичный стэк: postgres, redis, sidekiq, nginx, app





Ruby



Типичный стэк: postgres, redis, sidekiq, nginx, app



Можно запускать в с [rails server](#) ещё и [guard](#) – для отладки фронтенда





Ruby



Типичный стэк: postgres, redis, sidekiq, nginx, app

Можно запускать в с rails server ещё и guard – для отладки фронтенда

Для приближения к prod - rails и nginx в одном контейнере





Ruby

- Типичный стэк: postgres, redis, sidekiq, nginx, app

- Можно запускать в с rails server ещё и guard – для отладки фронтенда

- Для приближения к prod - rails и nginx в одном контейнере

- ...а на реальном продакшене лучше nginx держать отдельно

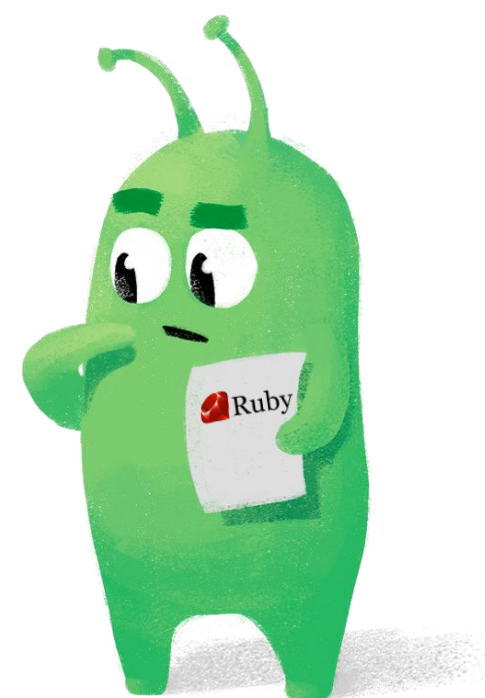




Ещё немного о Ruby

Для установки зависимостей:

```
RUN bundle check || bundle install
```



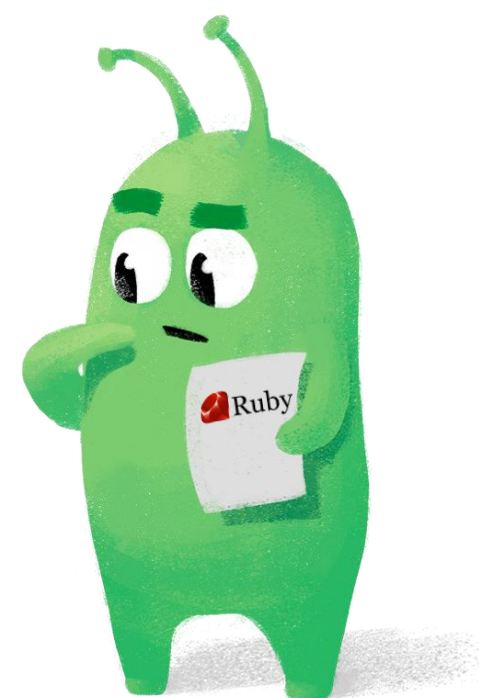


Ещё немного о Ruby

Для установки зависимостей:

```
RUN bundle check || bundle install
```

Хранить під-файл в tmpfs, либо запускать bundle скриптом с проверкой





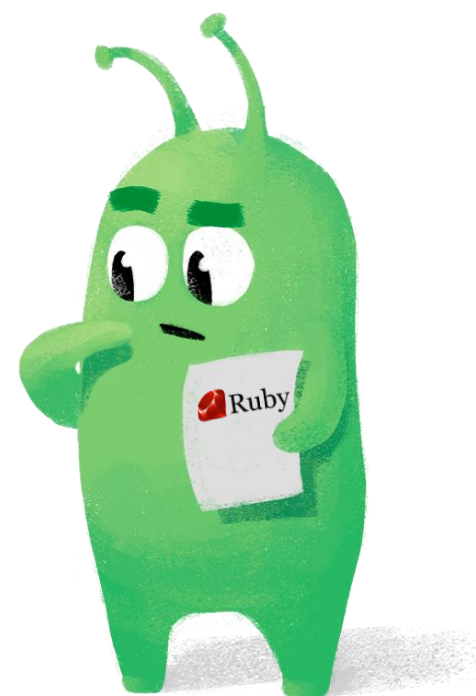
Ещё немного о Ruby

Для установки зависимостей:

```
RUN bundle check || bundle install
```

Хранить pid-файл в tmpfs, либо запускать bundle скриптом с проверкой

Оптимизируйте размер образа - удаляйте статику и кэши





Ruby on Rails Dockerfile

```
FROM ruby:2.3-alpine
```

```
RUN set -ex && apk add --update --virtual runtime-deps postgresql-client nodejs  
libffi-dev readline sqlite && rm -rf /var/cache/apk/*
```

```
WORKDIR /tmp
```

```
COPY Gemfile Gemfile.lock ./
```

```
RUN set -ex && apk add --virtual build-deps build-base openssl-dev postgresql-dev  
libc-dev linux-headers libxml2-dev libxslt-dev readline-dev && \  
    bundle install --clean --no-cache --without development --jobs=2 && \  
    rm -rf /var/cache/apk/* && \  
    apk del build-deps
```

```
ENV APP_HOME /app
```

```
COPY . $APP_HOME
```

```
WORKDIR $APP_HOME
```

```
ENV RAILS_ENV=production RACK_ENV=production
```

```
EXPOSE 3000
```

```
CMD ["bundle", "exec", "puma", "-C", "config/puma.rb"]
```





Golang

— Docker сам написан на Go – значит с этим языком меньше проблем (нет)

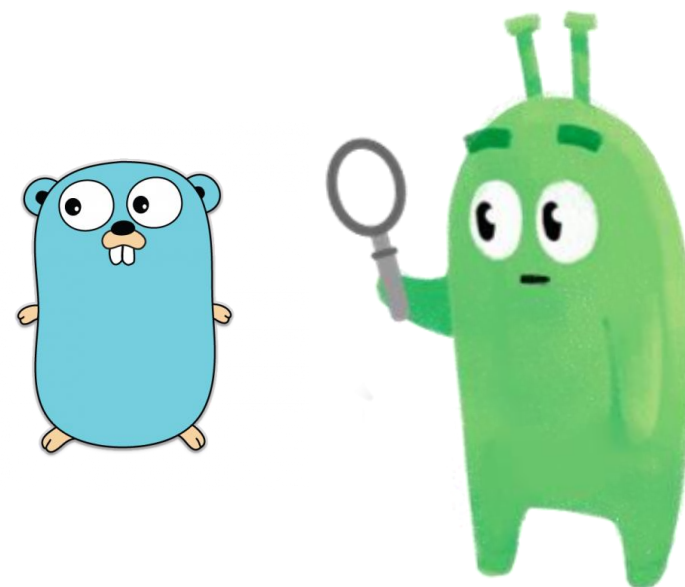




Golang

- Docker сам написан на Go – значит с этим языком меньше проблем (нет)

- Можно запускать собранный бинарь в образе **FROM scratch**





Golang

- Docker сам написан на Go – значит с этим языком меньше проблем (нет)
- Можно запускать собранный бинарь в образе **FROM scratch**
- При локальной сборке часто применяют Makefile





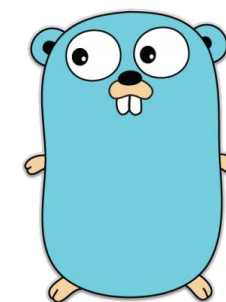
Golang

- Docker сам написан на Go – значит с этим языком меньше проблем (нет)

- Можно запускать собранный бинарь в образе **FROM scratch**

- При локальной сборке часто применяют Makefile

- Всё так же актуальна многоступенчатая сборка образов.





Golang

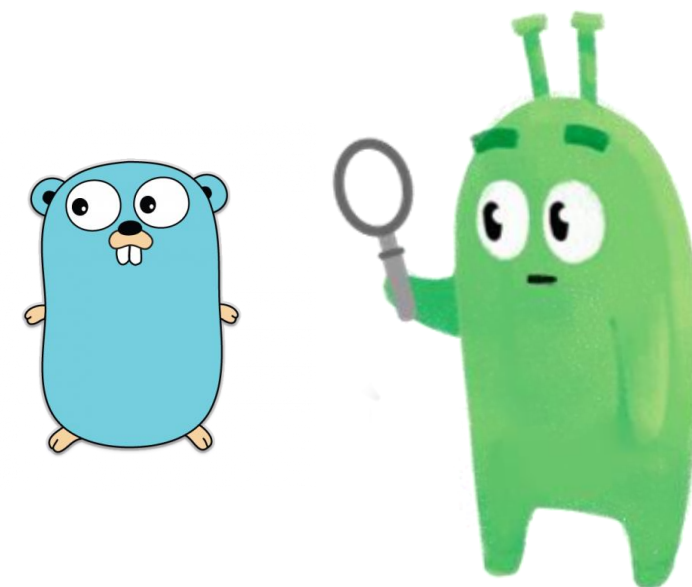
Docker сам написан на Go – значит с этим языком меньше проблем (нет)

Можно запускать собранный бинарь в образе **FROM scratch**

При локальной сборке часто применяют Makefile

Всё так же актуальна многоступенчатая сборка образов.

Goroutines в контейнерах под большой нагрузкой – жди беды





Golang

Docker сам написан на Go – значит с этим языком меньше проблем (нет)

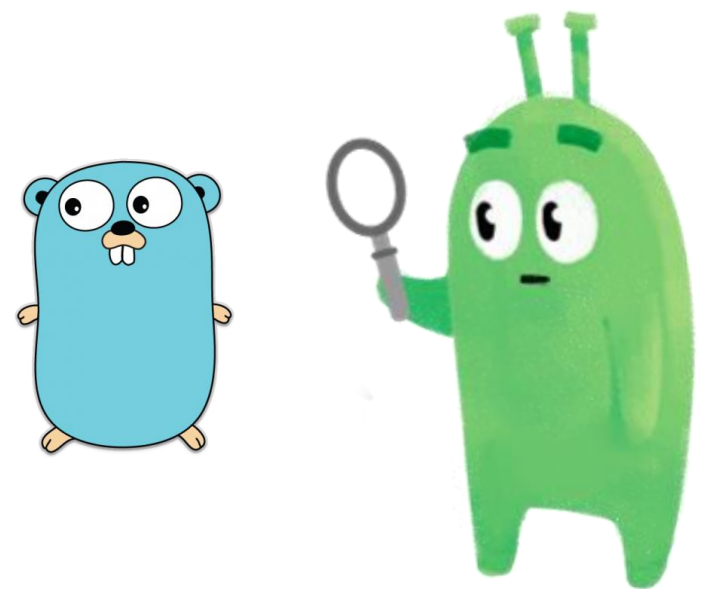
Можно запускать собранный бинарь в образе **FROM scratch**

При локальной сборке часто применяют Makefile

Всё так же актуальна многоступенчатая сборка образов.

Goroutines в контейнерах под большой нагрузкой – жди беды

...или жертвуй производительностью





Golang Dockerfile

```
# Первый шаг сборки
FROM golang:alpine AS builder

# Ставим git, это нужно для установки зависимостей
RUN apk update && apk add --no-cache git

ENV USER=appuser
ENV UID=10001

# Создаём пользователя максимально безопасно
RUN adduser \
  --disabled-password \
  --gecos "" \
  --home "/nonexistent" \
  --shell "/sbin/nologin" \
  --no-create-home \
  --uid "${UID}" \
  "${USER}"WORKDIR $GOPATH/src/mypackage/myapp/

# Копируем код в образ и ставим зависимости
COPY . .
# Если используете Go 1.10 и ниже
# RUN go get -d -v
```





Golang Dockerfile

```
# Первый шаг сборки
FROM golang:alpine AS builder

# Ставим git, это нужно для установки зависимостей
RUN apk update && apk add --no-cache git

ENV USER=appuser
ENV UID=10001

# Создаём пользователя максимально безопасно
RUN adduser \
  --disabled-password \
  --gecos "" \
  --home "/nonexistent" \
  --shell "/sbin/nologin" \
  --no-create-home \
  --uid "${UID}" \
  "${USER}"WORKDIR $GOPATH/src/mypackage/myapp/

# Копируем код в образ и ставим зависимости
COPY . .
# Если используете Go 1.10 и ниже
# RUN go get -d -v
```

```
# Если используете Go 1.11 и выше
RUN go mod download
RUN go mod verify

# Запускаем оптимизированную сборку
RUN GOOS=linux GOARCH=amd64 go build -ldflags="-w -s" -o /go/bin/hello

# Второй шаг сборки
FROM scratch

# Копируем всё необходимое из первого шага
COPY --from=builder /etc/passwd /etc/passwd
COPY --from=builder /etc/group /etc/group
COPY --from=builder /go/bin/hello /go/bin/hello

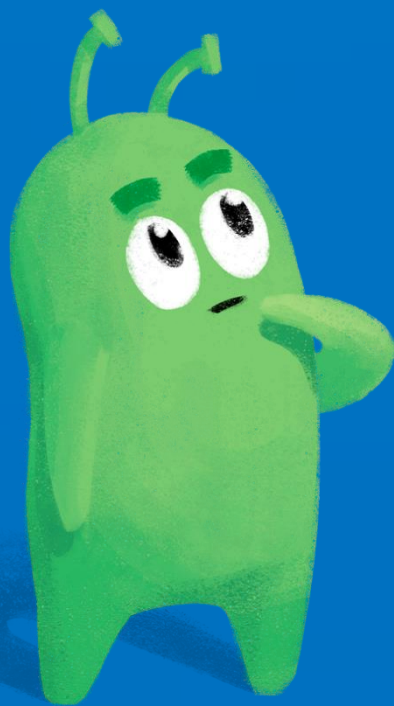
# Простой пользователь
USER appuser:appuser

# Порт с номером более 1024
EXPOSE 9292

ENTRYPOINT ["/go/bin/hello"]
```



Практика





southbridge.io

Спасибо!

слёрм

slurm.io