Введение в мониторинг. Системы мониторинга.

Проект microservices и проверка ДЗ

Создайте новую ветку в вашем microservices репозитории для выполнения данного ДЗ. Это первое задание про мониторинг, ветку назовите **monitoring-1**.

Проверка данного ДЗ будет производиться через Pull Request ветки с ДЗ.

После того, как один из преподавателей сделает approve пул реквеста, ветку с ДЗ можно смерджить и закрыть Pull Request.

План

- Prometheus: запуск, конфигурация, знакомство с Web UI
- Мониторинг состояния микросервисов
- Сбор метрик хоста с использованием экспортера
- Задания со *

Подготовка окружения

Создадим правило фаервола для Prometheus и Puma:

```
$ gcloud compute firewall-rules create prometheus-default --allow tcp:9090
$ gcloud compute firewall-rules create puma-default --allow tcp:9292
```

Создадим Docker хост в GCE и настроим локальное окружение на работу с ним (ссылка на gist):

```
$ export GOOGLE_PROJECT=_ваш-проект_

$ docker-machine create --driver google \
    --google-machine-image https://www.googleapis.com/compute/v1/projects/ubuntu-os-cloud/global/
images/family/ubuntu-1604-lts \
    --google-machine-type n1-standard-1 \
    --google-zone europe-west1-b \
    docker-host
...
Docker is up and running!
To see how to connect your Docker Client to the Docker Engine running on this virtual machine, run:
docker-machine env docker-host

$ eval $(docker-machine env docker-host)
```

Запуск Prometheus

Систему мониторинга Prometheus будем запускать внутри Docker контейнера. Для начального знакомства воспользуемся готовым образом с DockerHub.

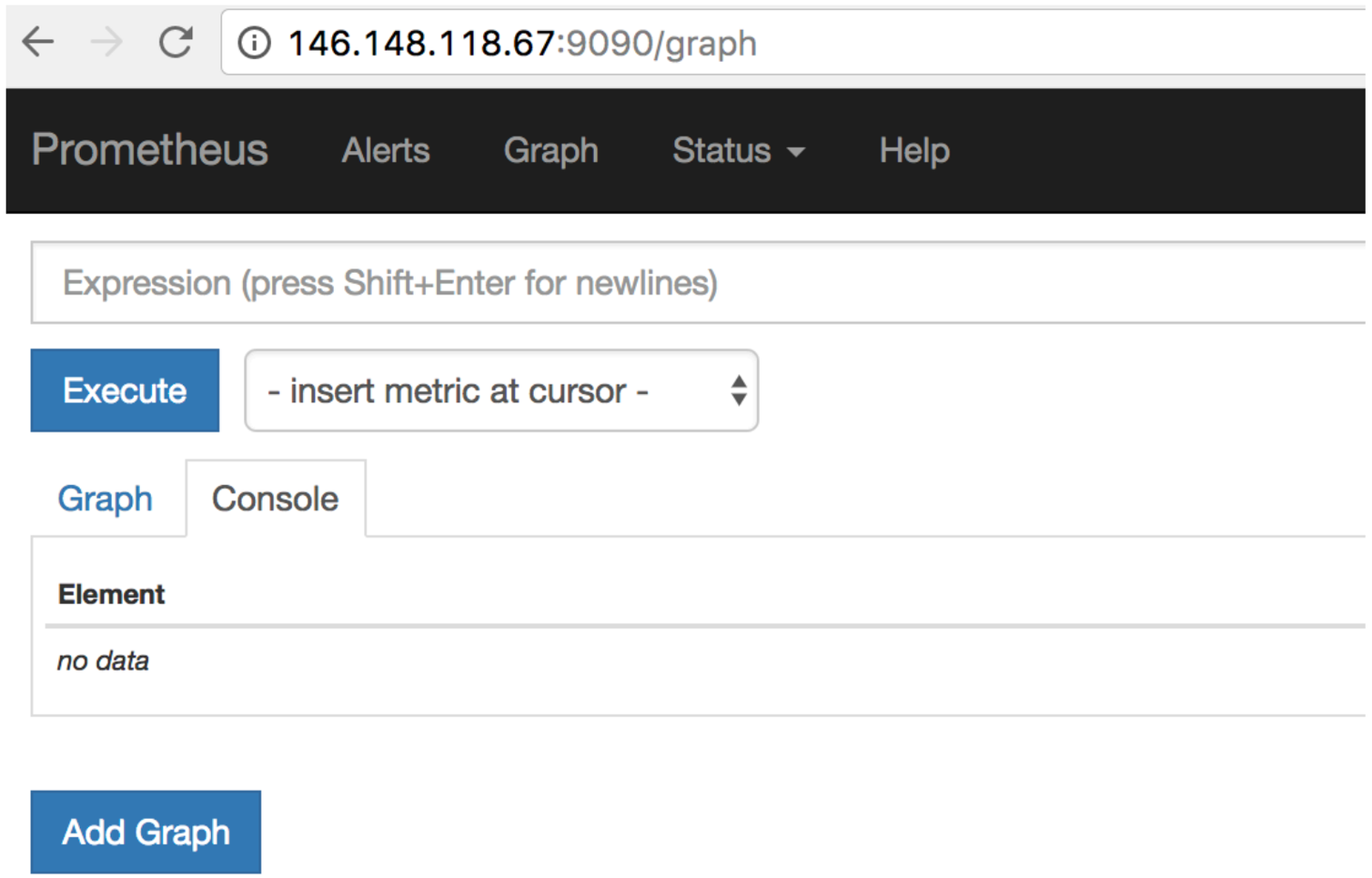
```
$ docker run --rm -p 9090:9090 -d --name prometheus  prom/
prometheus:v2.1.0

e6c684c9b9c9ae1071fafc4adfa75d2c96f683f2489f05a4d8662c531054ab45

$ docker ps

CONTAINER ID  IMAGE                 STATUS                 PORTS
NAMES
e6c684c9b9c9  prom/prometheus  Up 4 seconds        0.0.0.0:9090-
>9090/tcp   prometheus
```

Откроем веб интерфейс

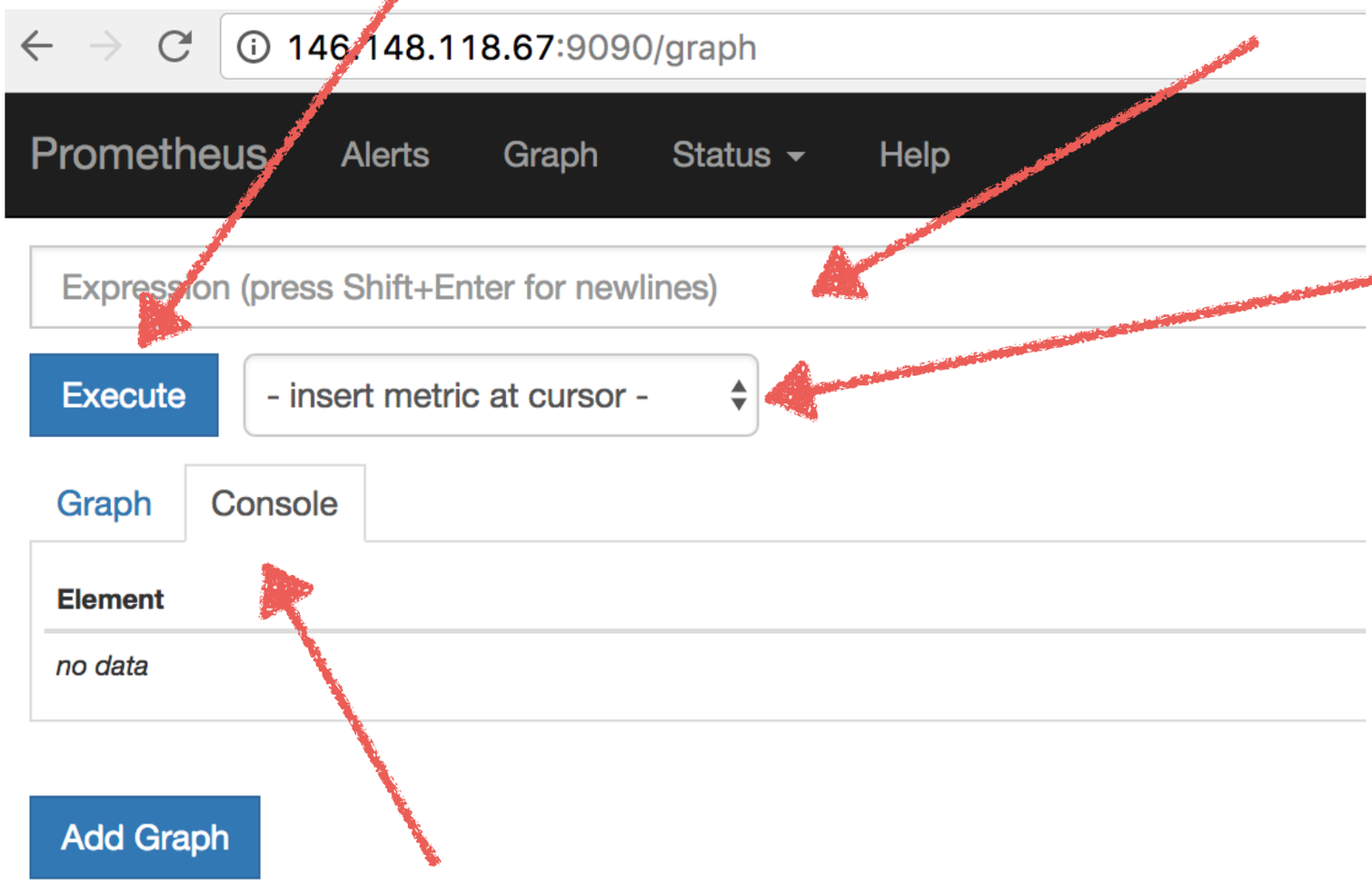


По умолчанию сервер слушает на порту 9090, а IP адрес созданной VM можно узнать, используя команду:

```
$ docker-machine ip docker-host
```

Выполнение запроса

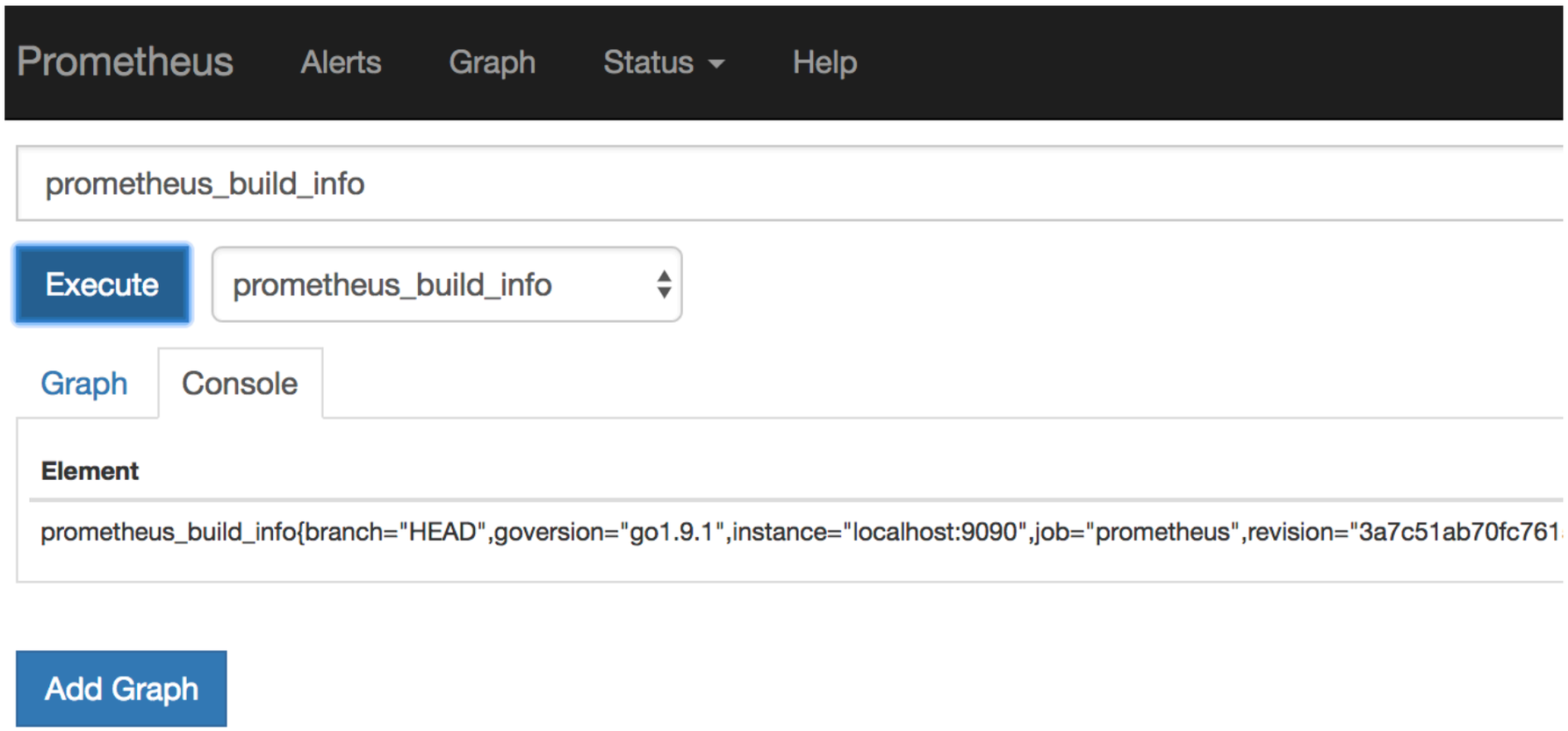
Строка ввода выражений для получения и анализа информации мониторинга (метрик) из хранилища



Выбор имеющихся метрик

Вкладка Console, которая сейчас активирована, выводит численное значение выражений. Вкладка Graph, левее от нее, строит график изменений значений метрик со временем

Если кликнем по "insert metric at cursor", то увидим, что Prometheus уже собирает какие-то метрики. По умолчанию он собирает статистику о своей работе. Выберем, например, метрику prometheus_build_info и нажмем Execute, чтобы посмотреть информацию о версии.



Поясним результат вывода

```
prometheus_build_info{branch="HEAD",goversion="go1.9.1",instanc
e="localhost:9090", job="prometheus", revision=
"3a7c51ab70fc7615cd318204d3aa7c078b7c5b20",version="1.8.1"}  1
```

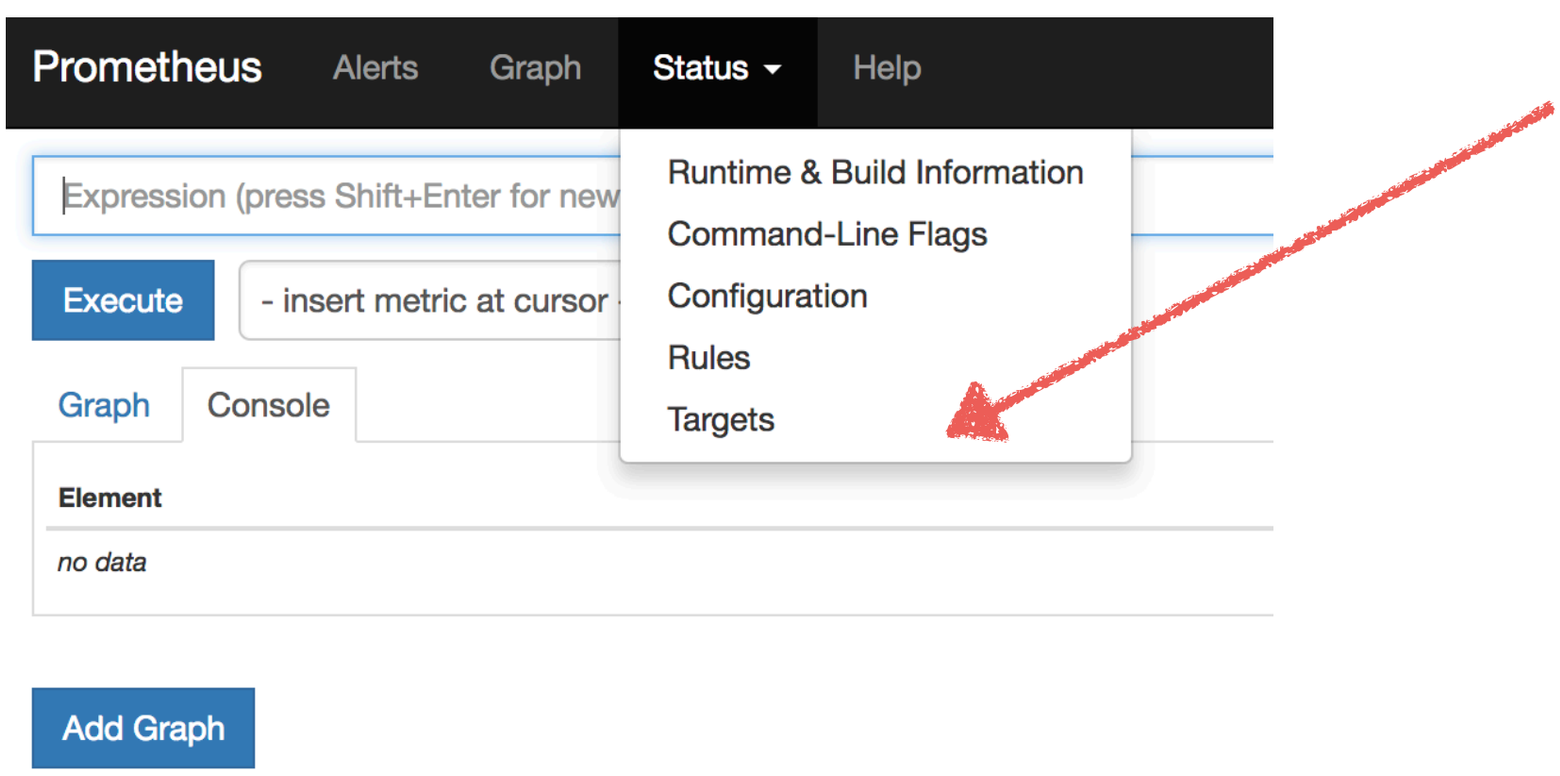
название метрики - идентификатор собранной информации.

лейбл - добавляет метаданных метрике, уточняет ее. Использование лейблов дает нам возможность не ограничиваться лишь одним названием метрик для идентификации получаемой информации. Лейблы содержаться в {} скобках и представлены наборами "ключ=значение".

значение метрики - численное значение метрики, либо NaN, если значение недоступно

Targets

Targets (цели) - представляют собой системы или процессы, за которыми следит Prometheus. Помним, что Prometheus является pull системой, поэтому он постоянно делает HTTP запросы на имеющиеся у него адреса (endpoints). Посмотрим текущий список целей



Targets

Only unhealthy jobs

prometheus (1/1 up) show less

Endpoint	State	Labels	Last Scrape	Error
http://localhost:9090/metrics	UP	instance="localhost:9090"	8.056s ago	

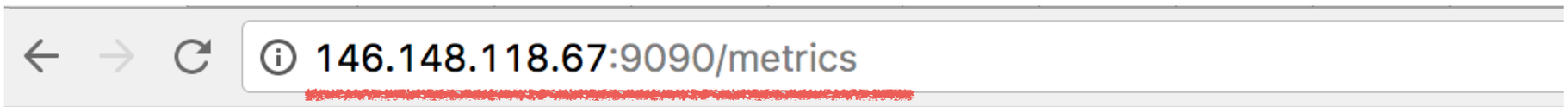
В Targets сейчас мы видим только сам Prometheus. У каждой цели есть свой список адресов (endpoints), по которым следует обращаться для получения информации.

В веб интерфейсе мы можем видеть состояние каждого endpoint-а (up); лейбл (instance="someURL"), который Prometheus автоматически добавляет к каждой метрике, получаемой с данного endpoint-а; а также время, прошедшее с момента последней операции сбора информации с endpoint-а.

Также здесь отображаются ошибки при их наличии и можно отфильтровать только неживые таргеты.

Обратите внимание на endpoint, который мы с вами видели на предыдущем слайде.

Мы можем открыть страницу в веб браузере по данному HTTP пути (host:port/metrics), чтобы посмотреть, как выглядит та информация, которую собирает Prometheus.




```
# HELP go_gc_duration_seconds A summary of the GC invocation durations.
# TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 6.203e-06
go_gc_duration_seconds{quantile="0.25"} 1.746e-05
go_gc_duration_seconds{quantile="0.5"} 2.0916e-05
go_gc_duration_seconds{quantile="0.75"} 2.602e-05
go_gc_duration_seconds{quantile="1"} 5.8589e-05
go_gc_duration_seconds_sum 0.001162098
go_gc_duration_seconds_count 54
# HELP go_goroutines Number of goroutines that currently exist.
# TYPE go_goroutines gauge
go_goroutines 142
# HELP go_memstats_alloc_bytes Number of bytes allocated and still in use.
# TYPE go_memstats_alloc_bytes gauge
go_memstats_alloc_bytes 4.63048e+07
```

Остановим контейнер

```
$ docker stop prometheus
```

Переупорядочим структуру директорий

До перехода к следующему шагу приведем структуру каталогов в более четкий/удобный вид:

1. Создадим директорию **docker** в корне репозитория и перенесем в нее директорию **docker-monolith** и файлы **docker-compose.*** и все **.env** (.env должен быть в .gitgnore), в репозиторий закоммичен **.env.example**, из которого создается **.env**

2. Создадим в корне репозитория директорию **monitoring**. В ней будет хранится **все**, что относится к мониторингу

3. Не забываем про .gitgnore и актуализируем записи при необходимости

P.S. С этого момента сборка сервисов отделена от docker-compose, поэтому инструкции build можно удалить из docker-compose.yml.

Создание Docker образа

Познакомившись с веб интерфейсом Prometheus и его стандартной конфигурацией, соберем на основе готового образа с DockerHub свой Docker образ с конфигурацией для мониторинга наших микросервисов.

Создайте директорию **monitoring/prometheus**. Затем в этой директории создайте простой Dockerfile, который будет копировать файл конфигурации с нашей машины внутрь контейнера:

monitoring/prometheus/Dockerfile

```
FROM prom/prometheus:v2.1.0
ADD prometheus.yml /etc/prometheus/
```

Конфигурация

Вся конфигурация Prometheus, в отличие от многих других систем мониторинга, происходит через файлы конфигурации и опции командной строки.

Мы определим простой конфигурационный файл для сбора метрик с наших микросервисов. В директории **monitoring/prometheus** создайте файл **prometheus.yml** со следующим содержимым (см. след. слайд)

prometheus.yml

(ссылка на gist)

```
---
global:
  scrape_interval: '5s'

scrape_configs:
  - job_name: 'prometheus'
    static_configs:
      - targets:
        - 'localhost:9090'

  - job_name: 'ui'
    static_configs:
      - targets:
        - 'ui:9292'

  - job_name: 'comment'
    static_configs:
      - targets:
        - 'comment:9292'
```

С какой частотой собирать метрики

Джобы объединяют в группы endpoint-ы, выполняющие одинаковую функцию

Адреса для сбора метрик (endpoints)

Создаем образ

В директории prometheus собираем Docker образ:

```
$ export USER_NAME=username
$ docker build -t $USER_NAME/prometheus .
```

Где USER_NAME - **ВАШ** логин от DockerHub.

В конце занятия нужно будет запушить на DockerHub собранные вами на этом занятии образы.

Образы микросервисов

В коде микросервисов есть healthcheck-и для проверки работоспособности приложения.

Сборку образов теперь **необходимо** производить при помощи скриптов docker_build.sh, которые есть в директории каждого сервиса. С его помощью мы добавим информацию из Git в наш **healthcheck**.

Соберем images

Выполните сборку образов при помощи скриптов docker_build.sh в директории каждого сервиса.

```
/src/ui          $ bash docker_build.sh

/src/post-py $ bash docker_build.sh

/src/comment $ bash docker_build.sh
```

Или сразу все из корня репозитория:

```
for i in ui post-py comment; do cd src/$i; bash
docker_build.sh; cd -; done
```

docker-compose.yml

Будем поднимать наш Prometheus совместно с микросервисами. Определите в вашем **docker/docker-compose.yml** файле новый сервис.

(ссылка на gist)

```
services:
  ...
  prometheus:
    image: ${USERNAME}/prometheus
    ports:
      - '9090:9090'
    volumes:
      - prometheus_data:/prometheus
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'
      - '--storage.tsdb.path=/prometheus'
      - '--storage.tsdb.retention=1d'
volumes:
  prometheus_data:
```

- Передаем доп. параметры в командной строке

- Задаем время хранения метрик в 1 день

Отметим, что сборка Docker образов с данного момента производится через скрипт **docker_build.sh**.

Поэтому удалите **build** директивы из **docker_compose.yml** и используйте директиву image.

docker-compose.yml

Мы будем использовать Prometheus для мониторинга всех наших микросервисов, поэтому нам необходимо, чтобы контейнер с ним мог общаться по сети со всеми другими сервисами, определенными в компоуз файле.

Самостоятельно добавьте секцию networks в определение сервиса Prometheus в **docker/docker-compose.yml**.

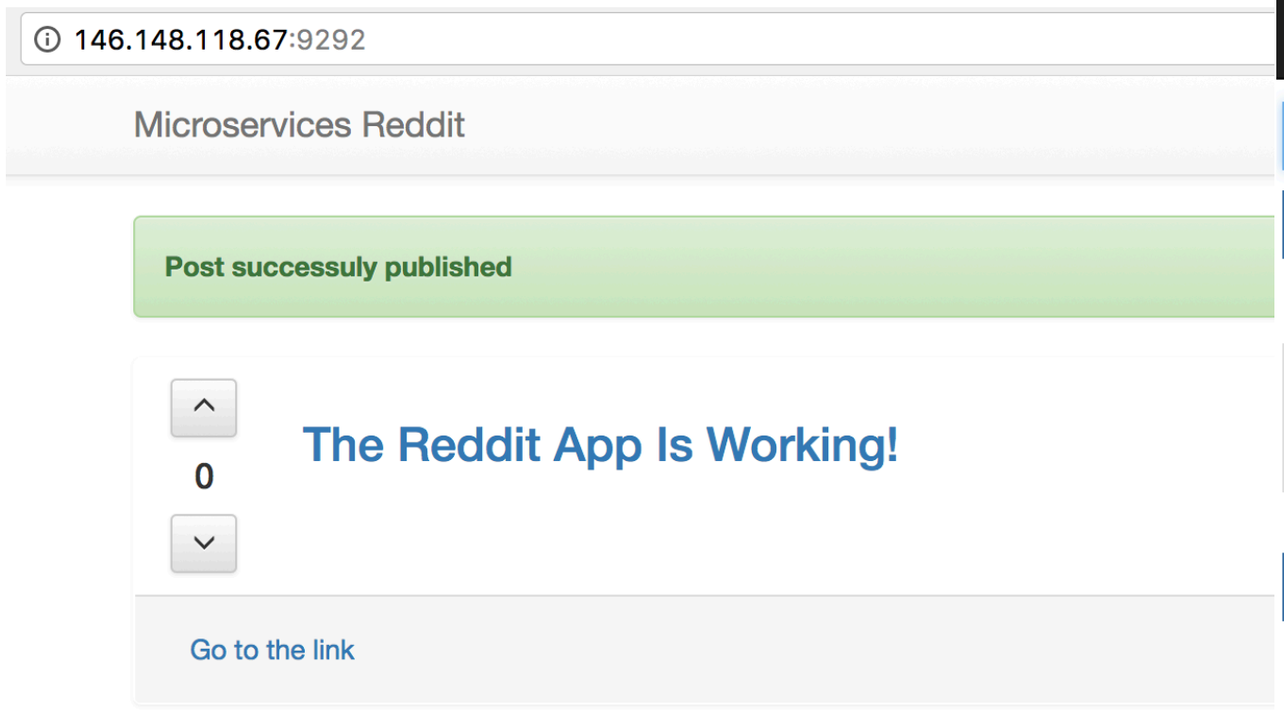
Также проверьте актуальность версий сервисов в .env и .env.example

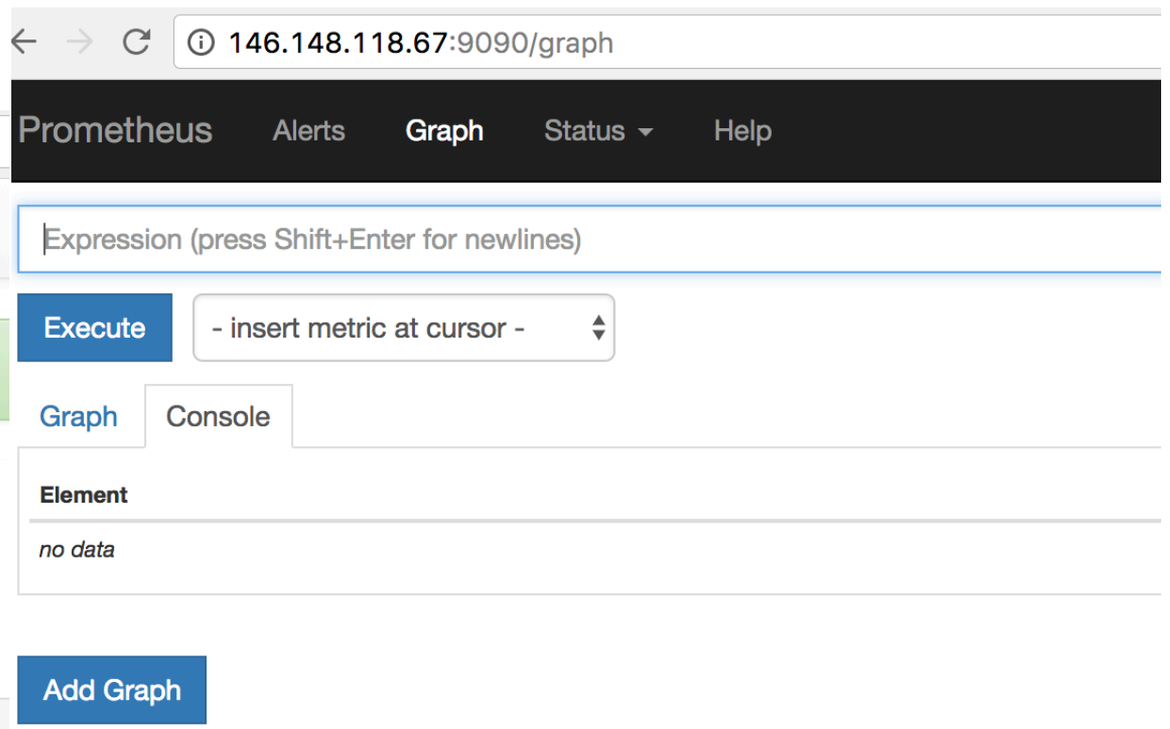
Запуск микросервисов

Поднимем сервисы, определенные в docker/docker-compose.yml

```
$ docker-compose up -d
```

Проверьте, что приложение работает и Prometheus запустился.

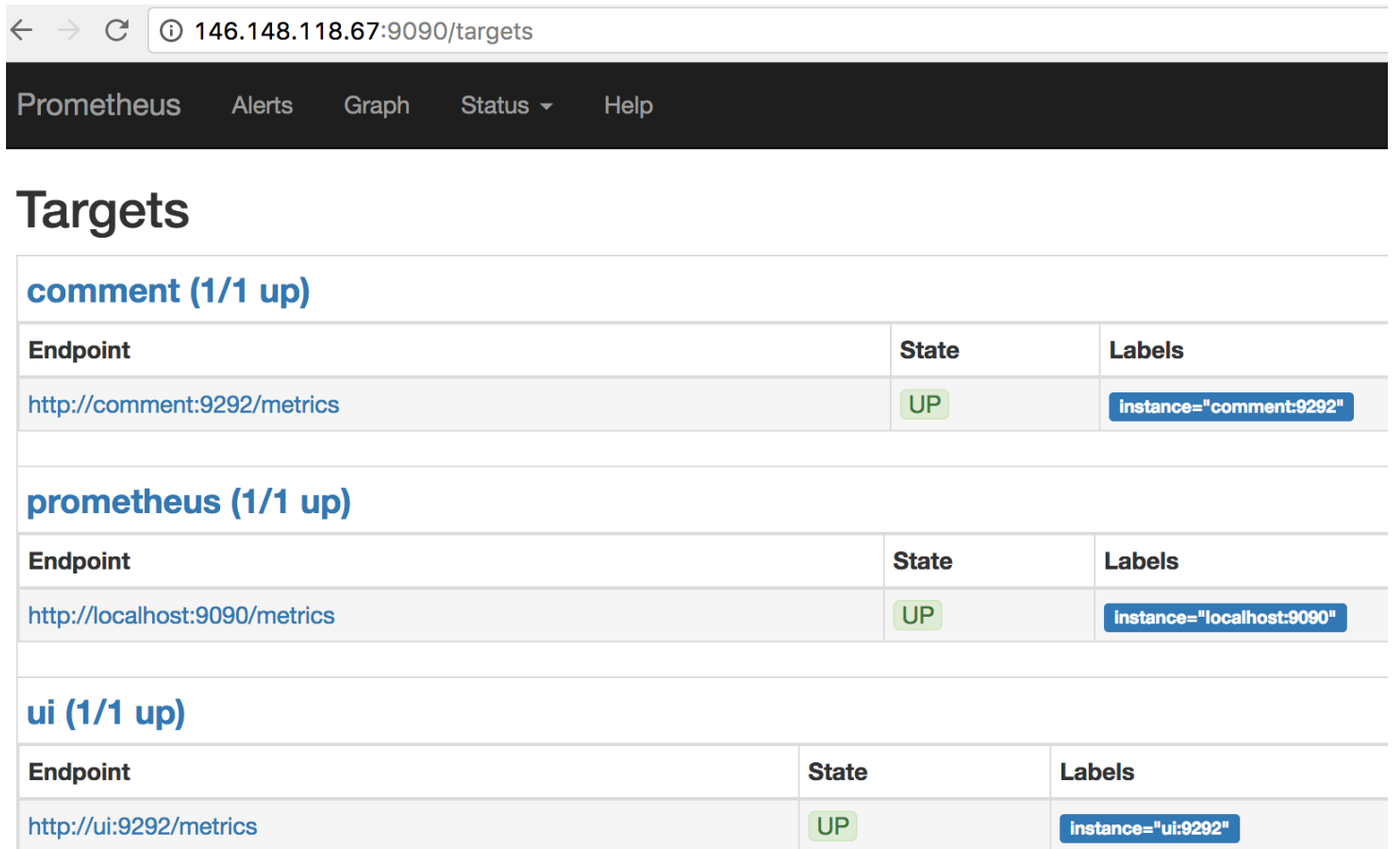




Мониторинг состояния микросервисов

Список endpoint-ов

Посмотрим список endpoint-ов, с которых собирает информацию Prometheus. Помните, что помимо самого Prometheus, мы определили в конфигурации мониторинг **ui** и **comment** сервисов. Endpoint-ы должны быть в состоянии UP.



Healthchecks

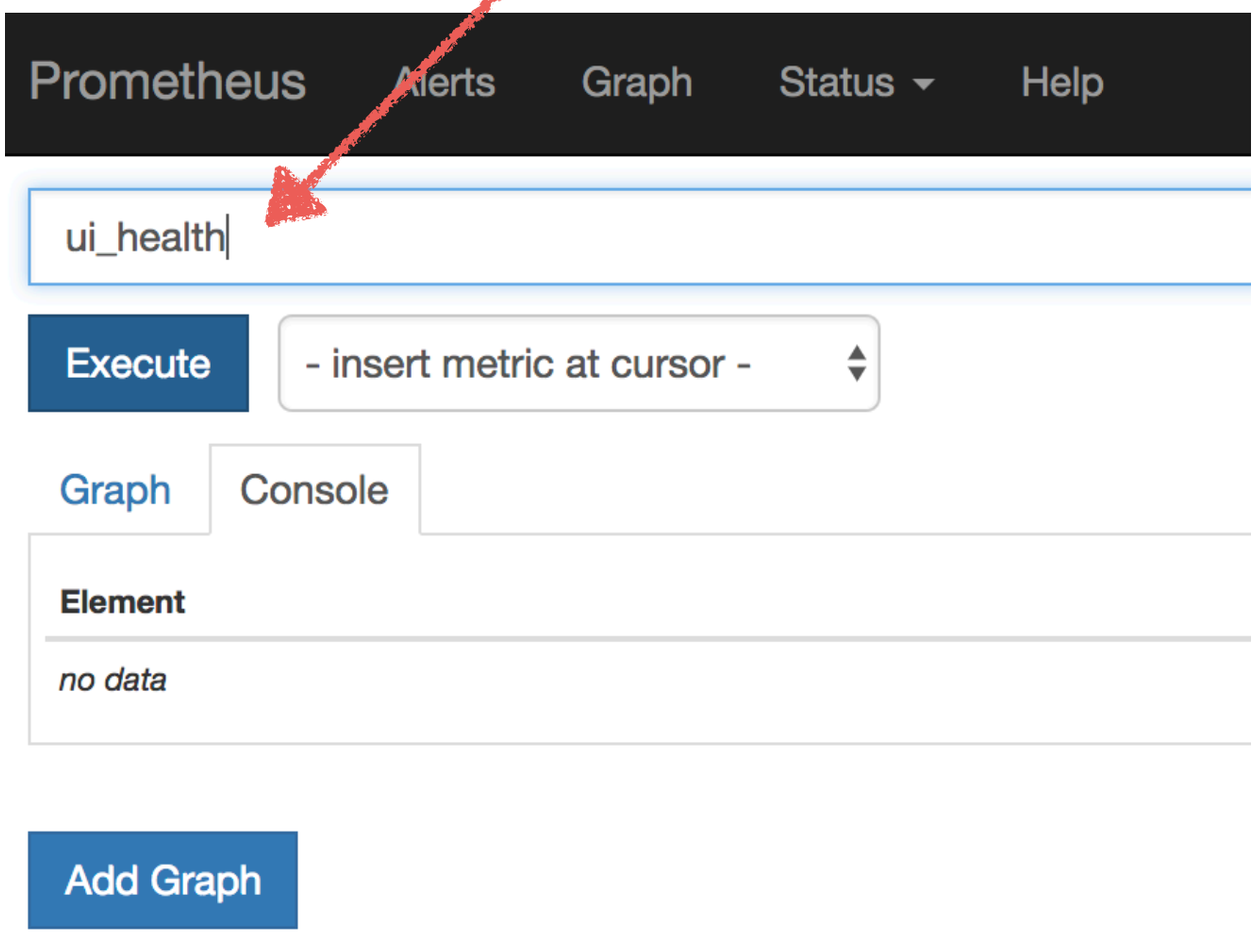
Healthcheck-и представляют собой проверки того, что наш сервис здоров и работает в ожидаемом режиме. В нашем случае healthcheck выполняется внутри кода микросервиса и выполняет проверку того, что все сервисы, от которых зависит его работа, ему доступны.

Если требуемые для его работы сервисы здоровы, то healthcheck проверка возвращает **status = 1**, что соответсвует тому, что сам сервис здоров.

Если один из нужных ему сервисов нездоров или недоступен, то проверка вернет **status = 0**.

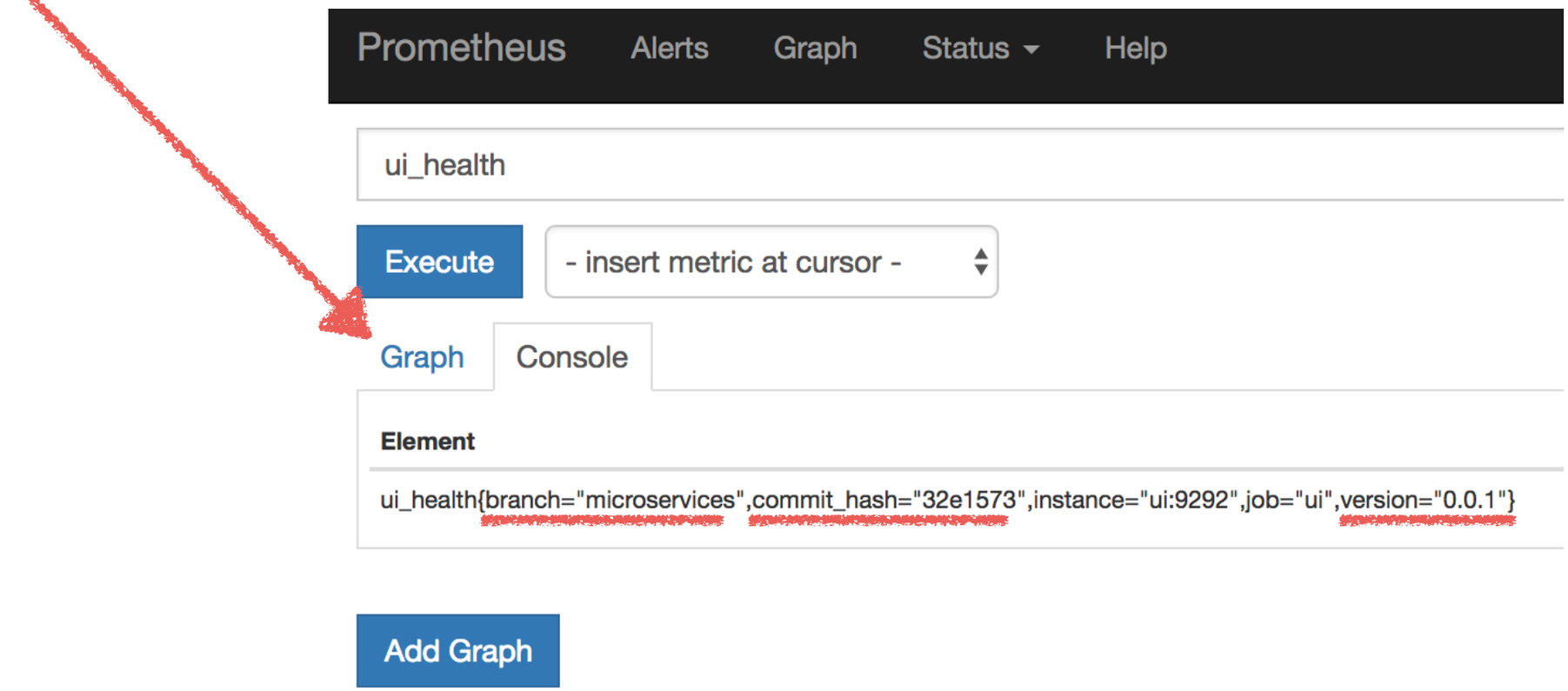
Состояние сервиса UI

В веб интерфейсе Prometheus выполните поиск по названию метрики ui_health



Состояние сервиса UI

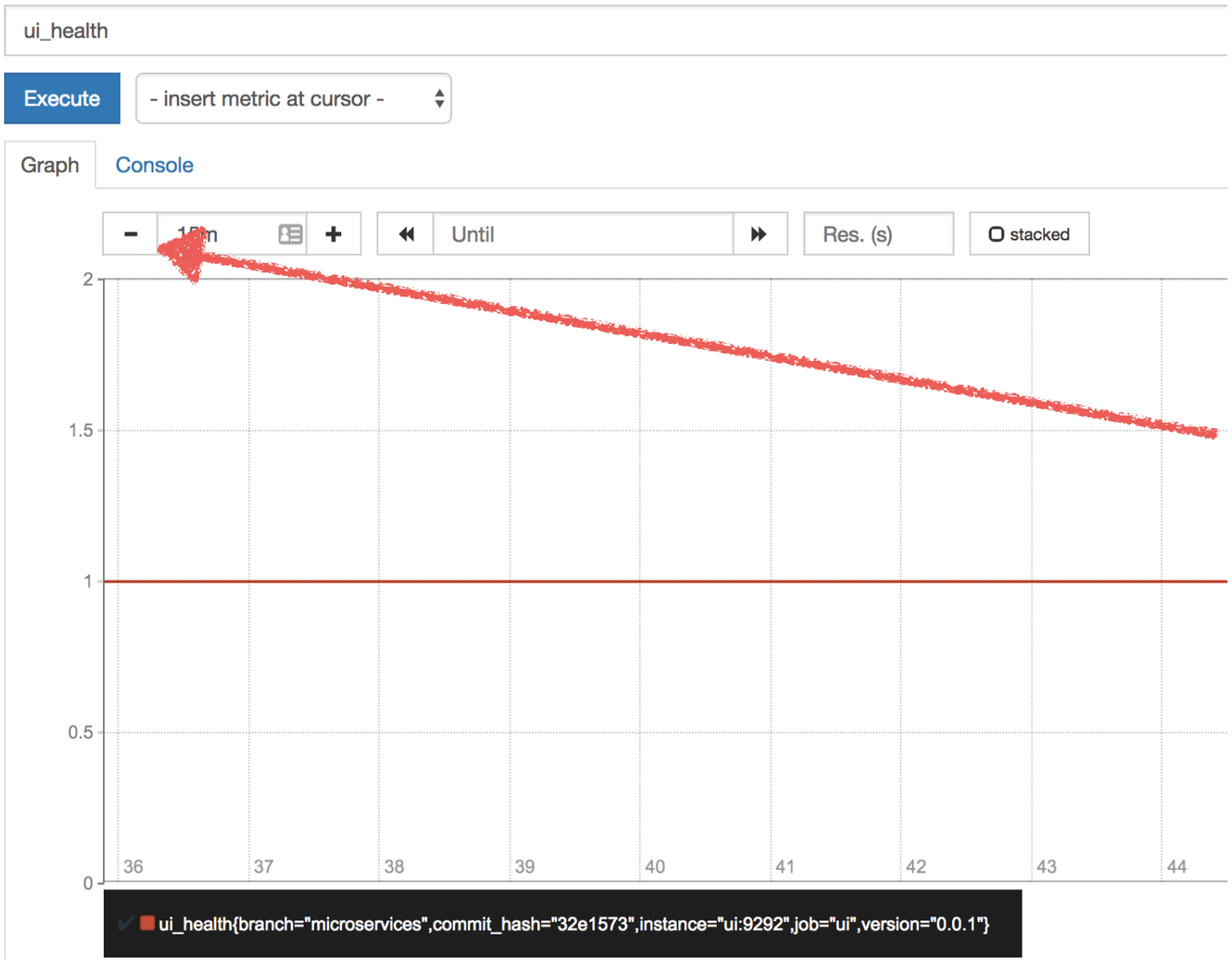
Построим график того, как менялось
значение метрики **ui_health** со временем



Обратим внимание, что, помимо имени метрики и ее значения, мы также видим информацию в лейблах о версии приложения, комите и ветке кода в Git-е.

Видим, что статус UI сервиса был стабильно 1, что означает, что сервис работал. Данный график оставьте открытым.

P.S. Если у вас статус не равен 1, проверьте какой сервис недоступен (слайд 32), и что у вас заданы все aliases для DB.



временной промежуток можно уменьшить, чтобы график выглядел лучше

Остановим post сервис

Мы говорили, что условились считать сервис здоровым, если все сервисы, от которых он зависит также являются здоровыми.

Попробуем остановить сервис post на некоторое время и проверим, как изменится статус ui сервиса, который зависим от post.

```
$ docker-compose stop post

Stopping starthealthchecks_post_1 ... done
```


Обновим наш график

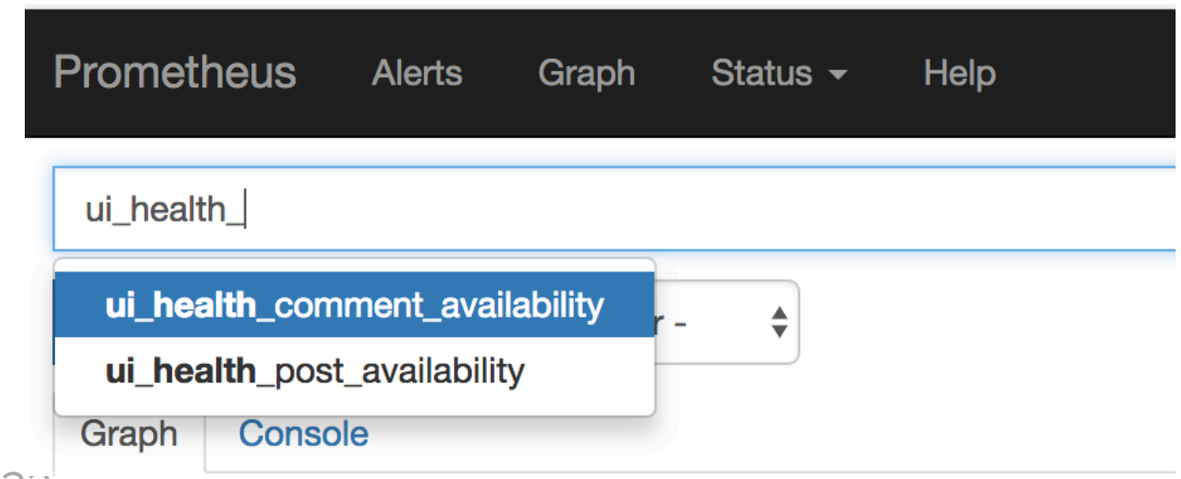
Метрика изменила свое значение на 0, что означает, что UI сервис стал нездоров

Поиск проблемы

Помимо статуса сервиса, мы также собираем статусы сервисов, от которых он зависит. Названия метрик, значения которых соответствует данным статусам, имеет формат ui_health_<service-name>.

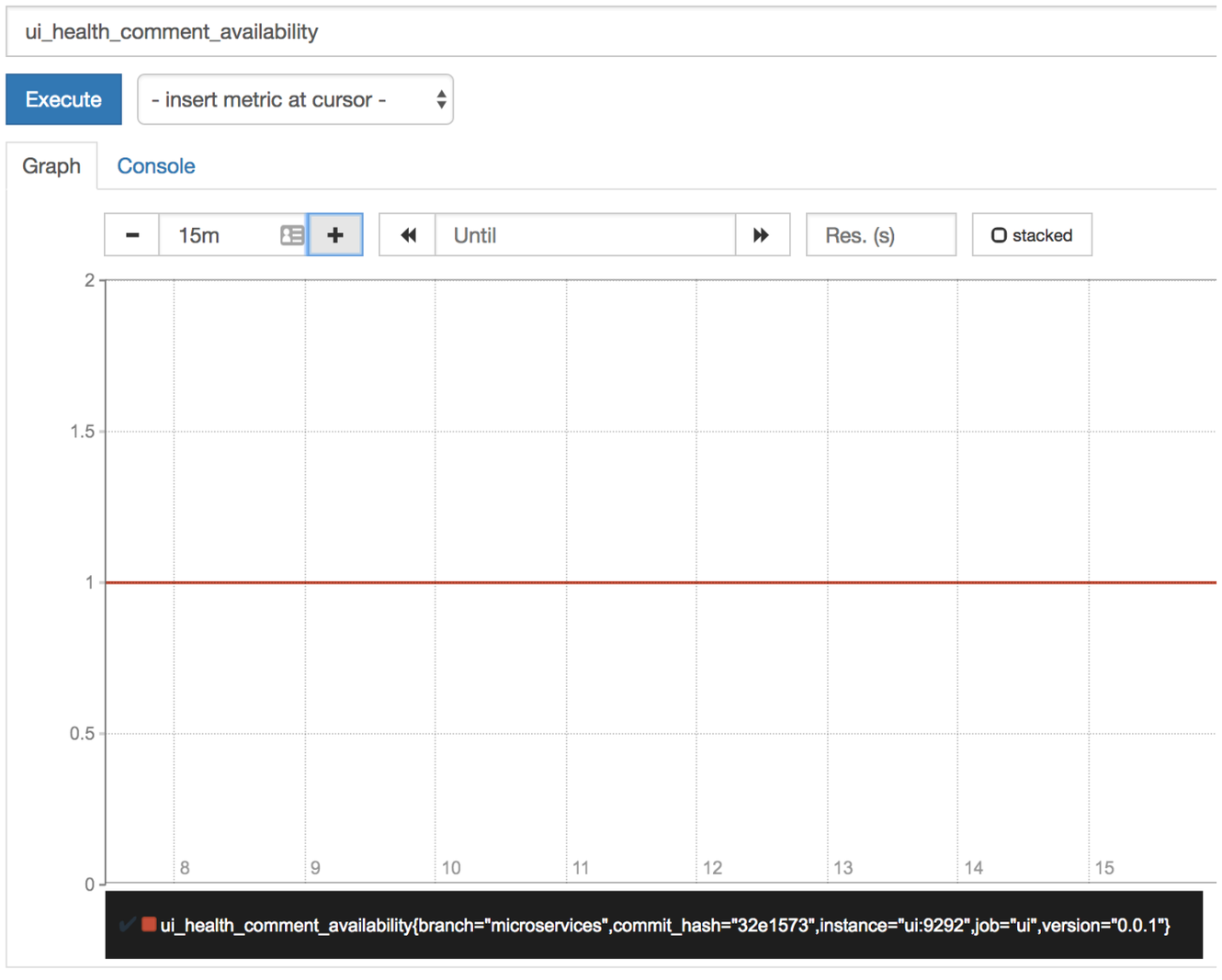
Посмотрим, не случилось ли чего плохого с сервисами, от которых зависит UI сервис.

Наберем в строке выражений ui_health_ и Prometheus нам предложит дополнить названия метрик.



Проверим comment сервис

Видим, что сервис свой статус не менял в данный промежуток времени



А с post сервисом все плохо

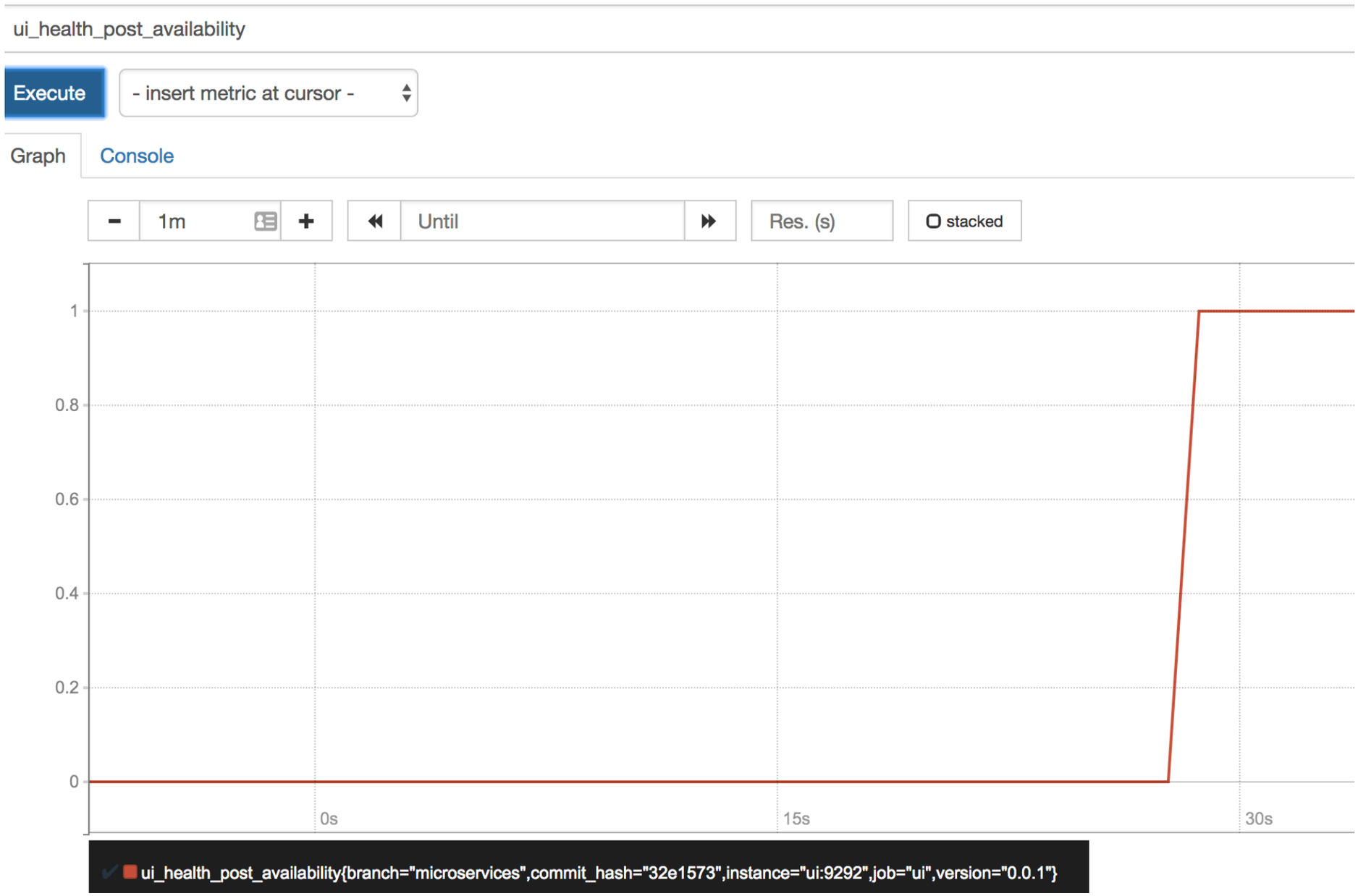
Чиним

Проблему мы обнаружили и знаем, как ее поправить (ведь мы же ее и создали :)). Поднимем post сервис.

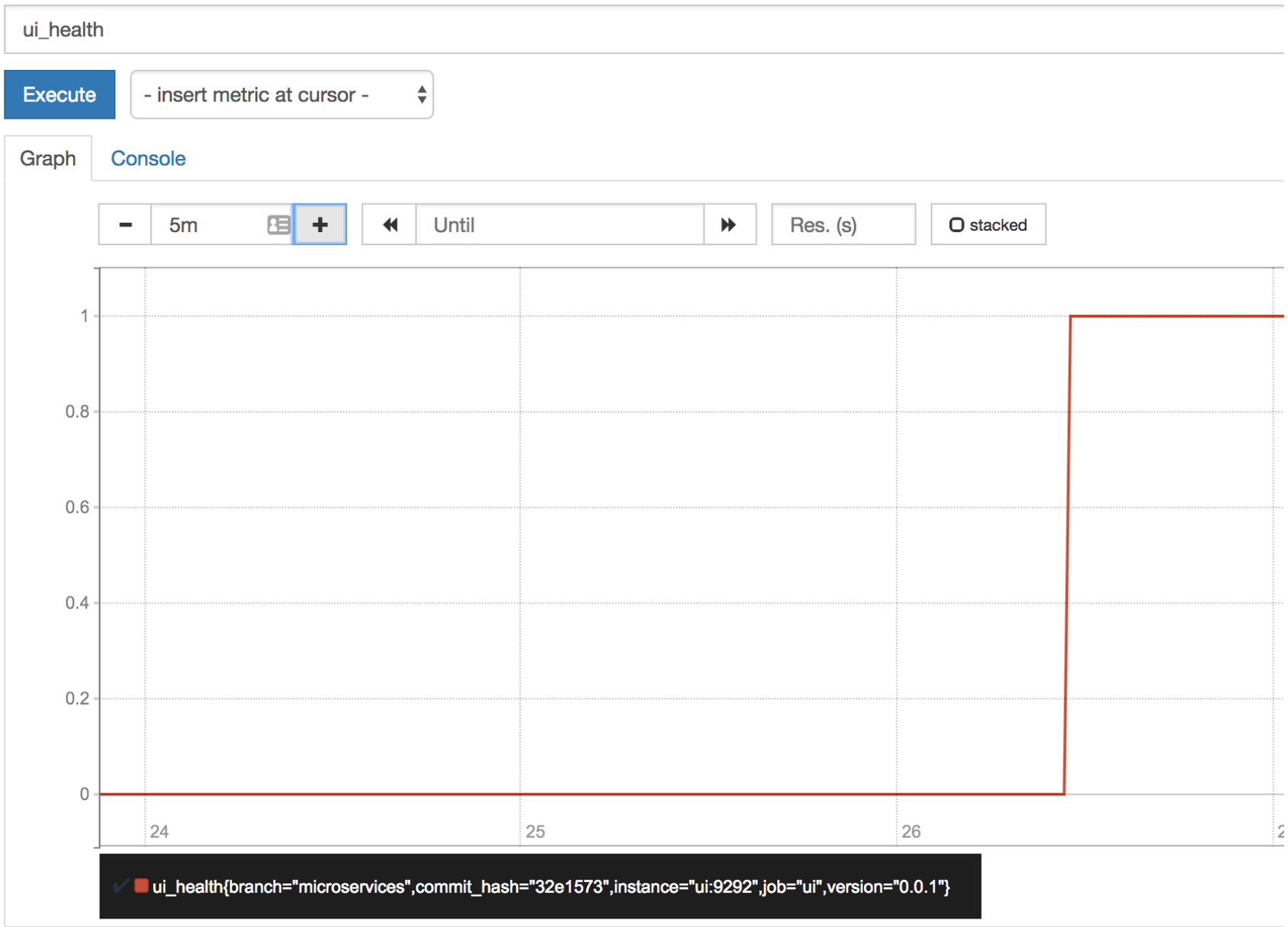
```
$ docker-compose start post

Starting post ... done
```

Post сервис поправился



UI сервис тоже



Самостоятельно

При желании, можно попробовать остановить comment сервис или БД для какого-то из сервисов и провести аналогичные операции по мониторингу ситуации.

Сбор метрик хоста

Exporters

Экспортер похож на вспомогательного агента для сбора метрик.

В ситуациях, когда мы не можем реализовать отдачу метрик Prometheus в коде приложения, мы можем использовать экспортер, который будет транслировать метрики приложения или системы в формате доступном для чтения Prometheus.

Exporters

- Программа, которая делает метрики доступными для сбора Prometheus

- Дает возможность конвертировать метрики в нужный для Prometheus формат

- Используется когда нельзя поменять код приложения

- Примеры: PostgreSQL, RabbitMQ, Nginx, Node exporter, cAdvisor

Node exporter

Воспользуемся Node экспортер для сбора информации о работе Docker хоста (виртуалки, где у нас запущены контейнеры) и предоставлению этой информации в Prometheus.

docker-compose.yml

Node экспортер будем запускать также в контейнере. Определим еще один сервис в docker/docker-compose.yml файле.

Не забудьте также добавить определение сетей для сервиса node-exporter, чтобы обеспечить доступ Prometheus к экспортеру.

(ссылка на gist)

```
services:
  ...

  node-exporter:
    image: prom/node-exporter:v0.15.2
    user: root
    volumes:
      - /proc:/host/proc:ro
      - /sys:/host/sys:ro
      - /:/rootfs:ro
    command:
      - '--path.procfs=/host/proc'
      - '--path.sysfs=/host/sys'
      - '--collector.filesystem.ignored-mount-points="^/(sys|proc|dev|host|etc)($$|/)"'
```

prometheus.yml

Чтобы сказать Prometheus следить за еще одним сервисом, нам нужно добавить информацию о нем в конфиг.

Добавим еще один job:

```
scrape_configs:
  ...

  - job_name: 'node'
    static_configs:
      - targets:
        - 'node-exporter:9100'
```

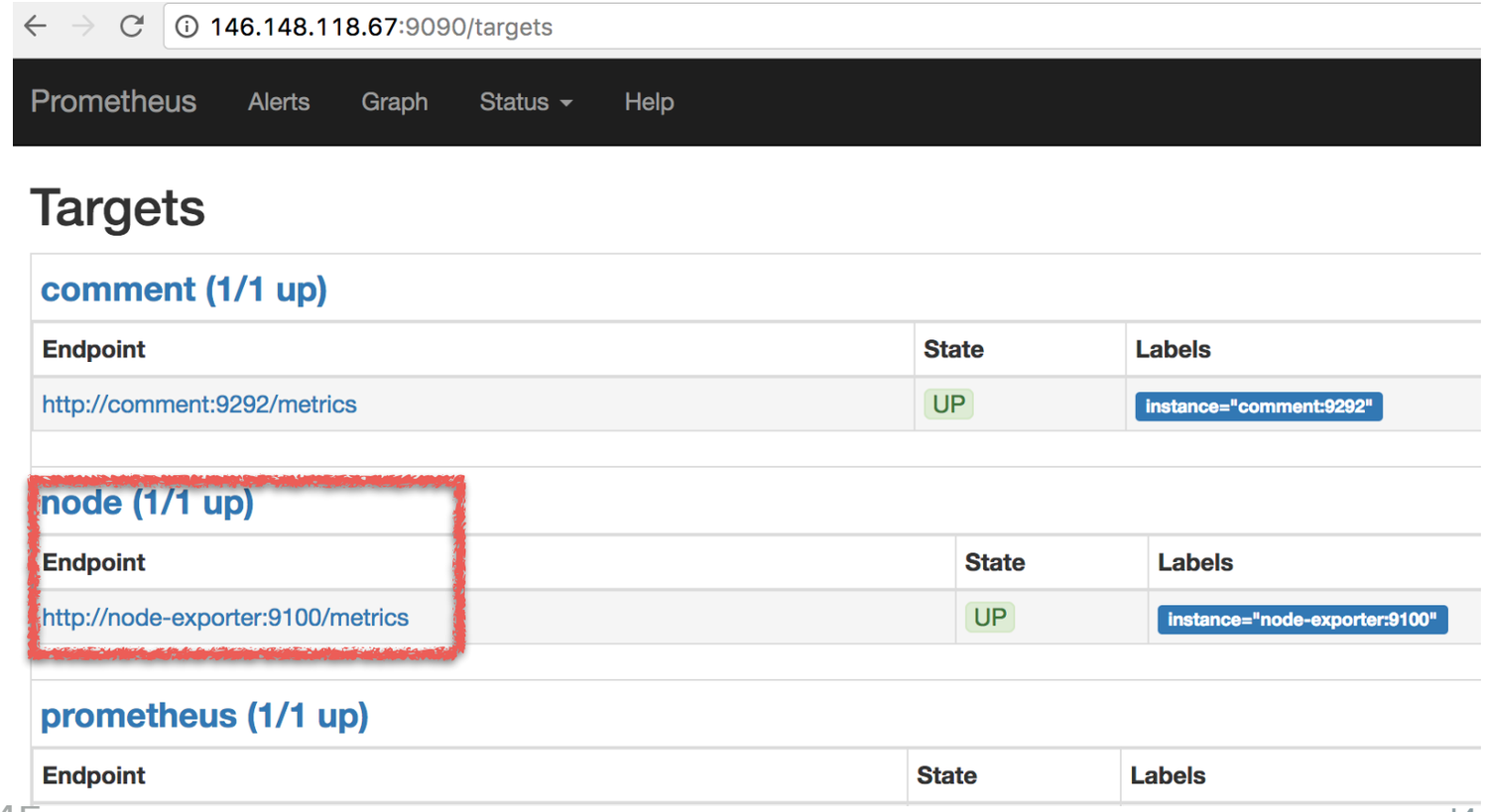
Не забудем собрать новый Docker для Prometheus:

```
monitoring/prometheus $ docker build -t $USER_NAME/prometheus .
```

Пересоздадим наши сервисы

```
$ docker-compose down
$ docker-compose up -d
```

посмотрим, список endpoint-ов Prometheus - должен появится еще один endpoint.



Получим информацию об использовании CPU

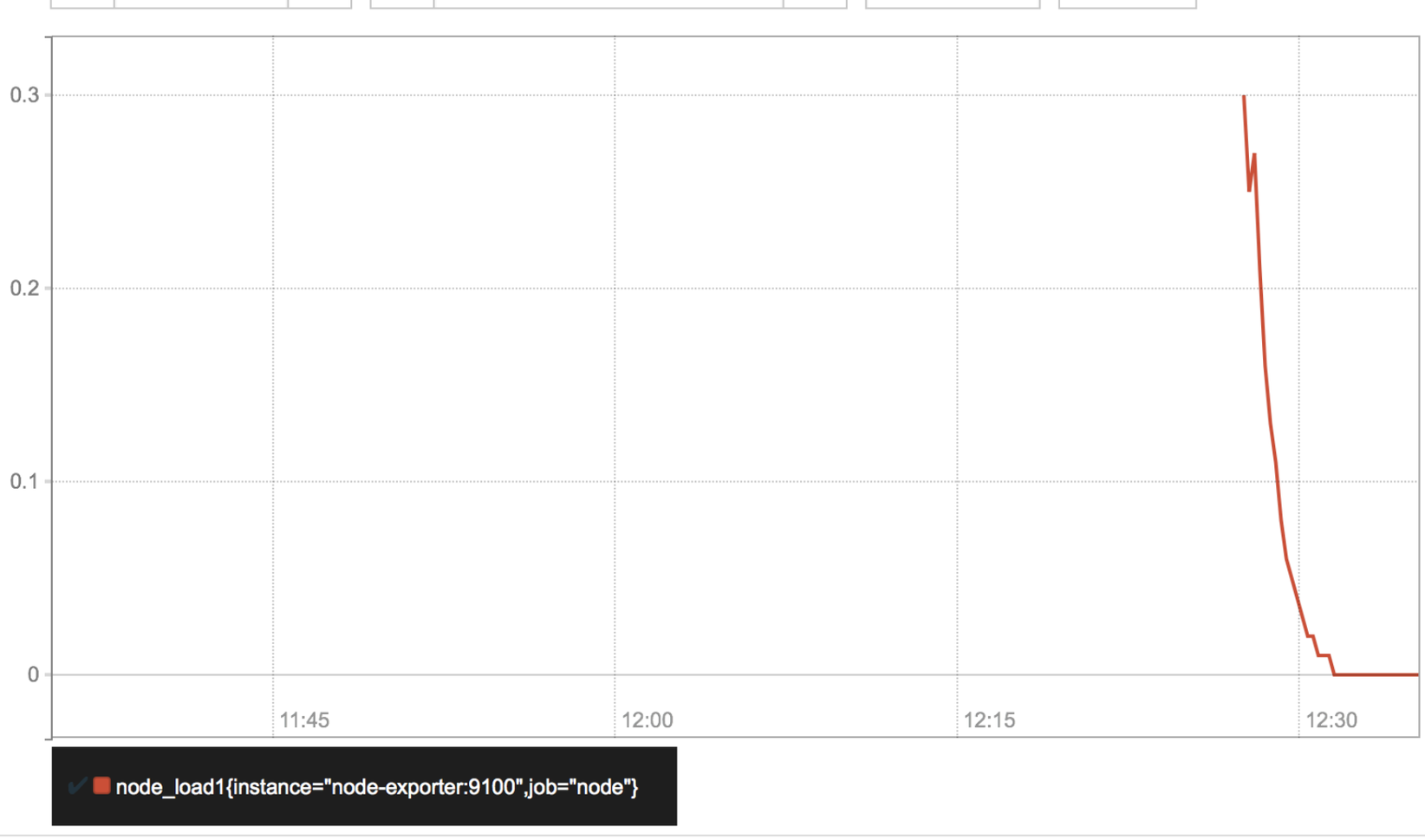
node_load1

Load time: 188ms
Resolution: 14s
Total time series: 1

Execute | - insert metric at cursor - |

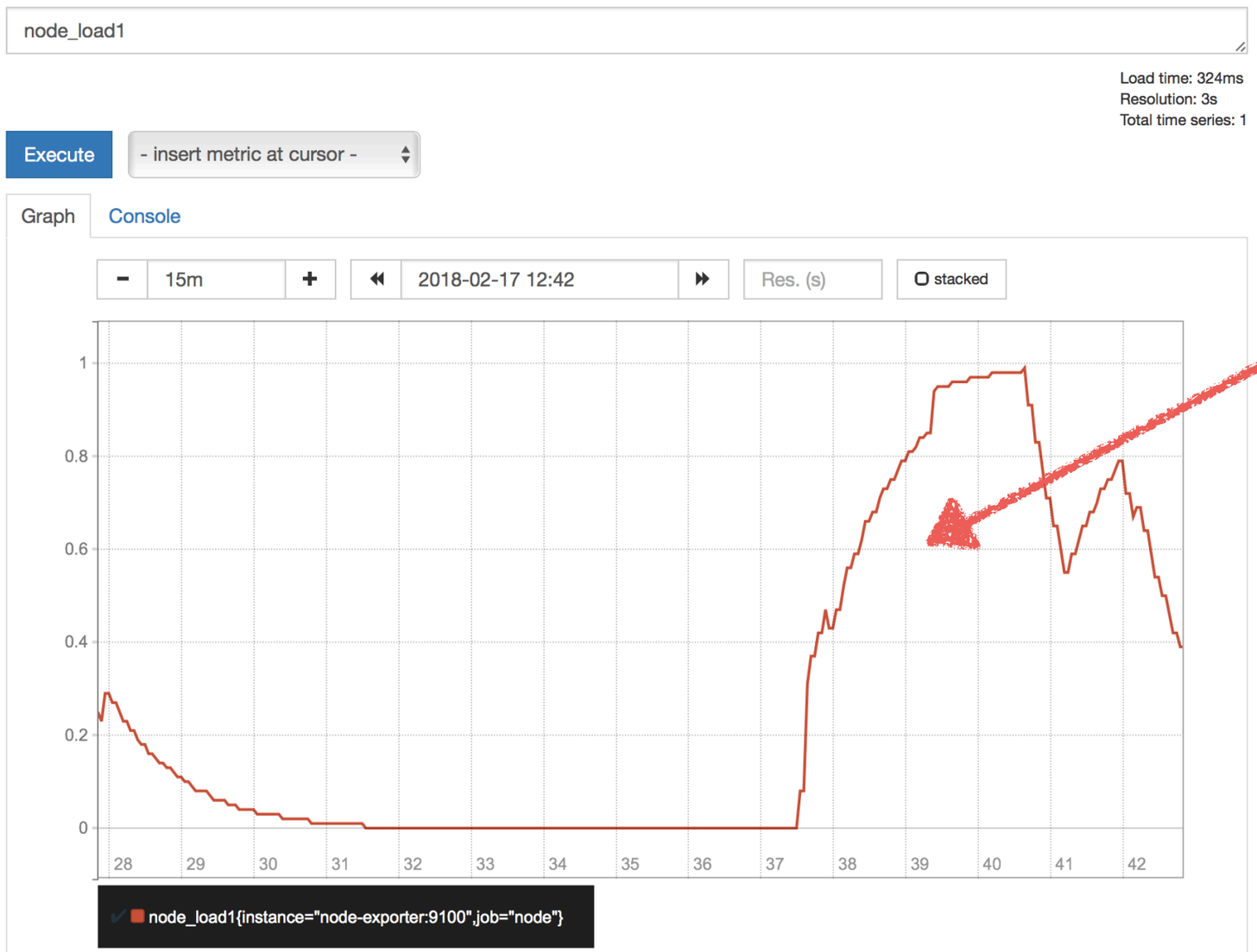
Graph | Console

− | 1h | + | ◀◀ | 2018-02-17 12:35 | ▶▶ | Res. (s) | stacked



Проверим мониторинг

- Зайдем на хост: `docker-machine ssh docker-host`
- Добавим нагрузки: `yes > /dev/null`



нагрузка выросла, мониторинг отображает повышение загруженности CPU

Завершение работы

- Запушьте собранные вами образы на DockerHub:

```
$ docker login
Login Succeeded

$ docker push $USER_NAME/ui
$ docker push $USER_NAME/comment
$ docker push $USER_NAME/post
$ docker push $USER_NAME/prometheus
```

- Удалите виртуалку:

```
$ docker-machine rm docker-host
```

- **Добавьте** ссылку на докер хаб с вашими образами в README.md и описание PR

Задание со *

Добавьте в Prometheus мониторинг **MongoDB** с использованием необходимого экспортера.

- Версию образа экспортера нужно фиксировать на последнюю стабильную
- Если будете добавлять для него **Dockerfile**, он должен быть в директории **monitoring**, а не в корне репозитория.

P.S. Проект dcu/mongodb_exporter не самый лучший вариант, т.к. у него есть проблемы с поддержкой (не обновляется)

Задание со *

Добавьте в Prometheus мониторинг сервисов **comment, post, ui** с помощью blackbox экспортера.

Blackbox exporter позволяет реализовать для Prometheus мониторинг по принципу черного ящика. Т.е. например мы можем проверить отвечает ли сервис по **http**, или принимает ли соединения порт.

- Версию образа экспортера нужно фиксировать на последнюю стабильную

- Если будете добавлять для него **Dockerfile**, он должен быть в директории **monitoring**, а не в корне репозитория.

Вместо blackbox_exporter можете попробовать использовать **Cloudprober** от Google.

Задание со *

Как вы могли заметить, количество компонент, для которых необходимо делать билд образов, растет. И уже сейчас делать это вручную не очень удобно.

Можно было бы конечно написать скрипт для автоматизации таких действий. Но гораздо лучше для этого использовать **Makefile**.

Задание: Напишите **Makefile**, который в минимальном варианте умеет:

1. Билдить любой или все образы, которые сейчас используются

2. Умеет пушить их в докер хаб

Дополнительно можете реализовать любой сценарии, которые вам кажутся полезными.

Документация: _1, _2, _3

Проверка ДЗ

- Результаты вашей работы находятся в ветке **monitoring-1** вашего microservices репозитория.
- В README внесите описание того, что сделано
- Создайте Pull Request к ветке master (**описание PR нужно заполнять**);
- В ревьюверы можно никого не добавлять;
- Добавьте "Labels" **monitoring** и **monitoring-1** к вашему Pull Request;

- После того, как один из преподавателей сделает approve пул реквеста, ветку с ДЗ можно смерджить и закрыть PR.