Docker swarm

План

- Построить кластер Docker Swarm
- Конфигурирование приложения и сервисов для Docker Swarm

Подготовка

Код микросервиса **ui** обновился для добавления функционала считывания переменных окружений. Проведенные изменения отражены в коммите. Чтобы скопировать измененные файлы, можно выполнить команду в папке приложений:

```
$ git pull
```

При этом Dockerfile для сборки образов сервисов лучше оставить свои.
После копирования выполните сборку образов микросервисов:

После копирования выполним сборку образа микросервиса:

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

Строим Swarm Cluster

Создадим машину **master-1** (ссылка на gist)

```
docker-machine create --driver google \
   --google-project  docker-182408  \
   --google-zone europe-west1-b \
   --google-machine-type g1-small \
   --google-machine-image $(gcloud compute images list
--filter ubuntu-1604-lts --uri) \
   master-1
```

По аналогии создайте машины **worker-1** и **worker-2**

P.S. не забудьте указать имя **своего** проекта GCP

Строим Swarm Cluster

```
$ eval $(docker-machine env master-1)
```

либо заходим по ssh

```
$ docker-machine ssh master-1
```

Инициализируем Swarm-mode

```
$ docker swarm init
```

P.S. если на сервере несколько сетевых интерфейсов или сервер находится за NAT, то необходимо указывать флаг **--advertise-addr** с конкретным адресом публикации. По-умолчанию это будет **<адрес интерфейса>:2377**

Строим Swarm Cluster

В результате выполнения **docker swarm init**:

- Текущая нода переключается в Swarm-режим
- Текущая нода назначается в качестве **Лидера** менеджеров кластера
- Ноде присваивается хостнейм машины
- Менеджер конфигурируется для прослушивания на порту **2377**
- Текущая нода получает статус **Active**, что означает возможность получать задачки от планировщика
- Запускается внутреннее распределенное хранилище данных Docker для работы оркестратора
- Генерируется самоподписный корневый (CA) сертификат для Swarm
- Генерируются токены для присоединения Worker и Manager нод к кластеру
- Создается Overlay-сеть **Ingress** для публикации сервисов наружу

Строим Swarm Cluster

Появится подобное сообщение:

```
Swarm initialized: current node (3ac3dc37xvup2v5hdmub5tlqc) is
now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token
SWMTKN-1-5dkxha7z0h9vfxqsoepxqybmehcs7mvfrtml00s8hxnn2nrgep-
chln12zdzd1805uensy5xouj7 10.132.0.6:2377
```

Кластер создан, в нем теперь есть 1 *manager* и можно добавить к нему новые ноды.
Выделенная команда позволит добавить только worker-ноды.
Также токен для добавления нод можно сгенерировать токен с помощью команды:
`$ docker swarm join-token manager/worker`

Строим Swarm Cluster

На хостах **worker-1** и **worker-2** выполнить:

```
$ docker swarm join --token <ваш токен> <advertise адрес manager'a>:237
```

Подключаемся к master-1 ноде (ssh или eval $(docker-machine ...))
Дальше работать будем только с ней. Команды в рамках Swarm-кластера можно запускать **только** на Manager-нодах.

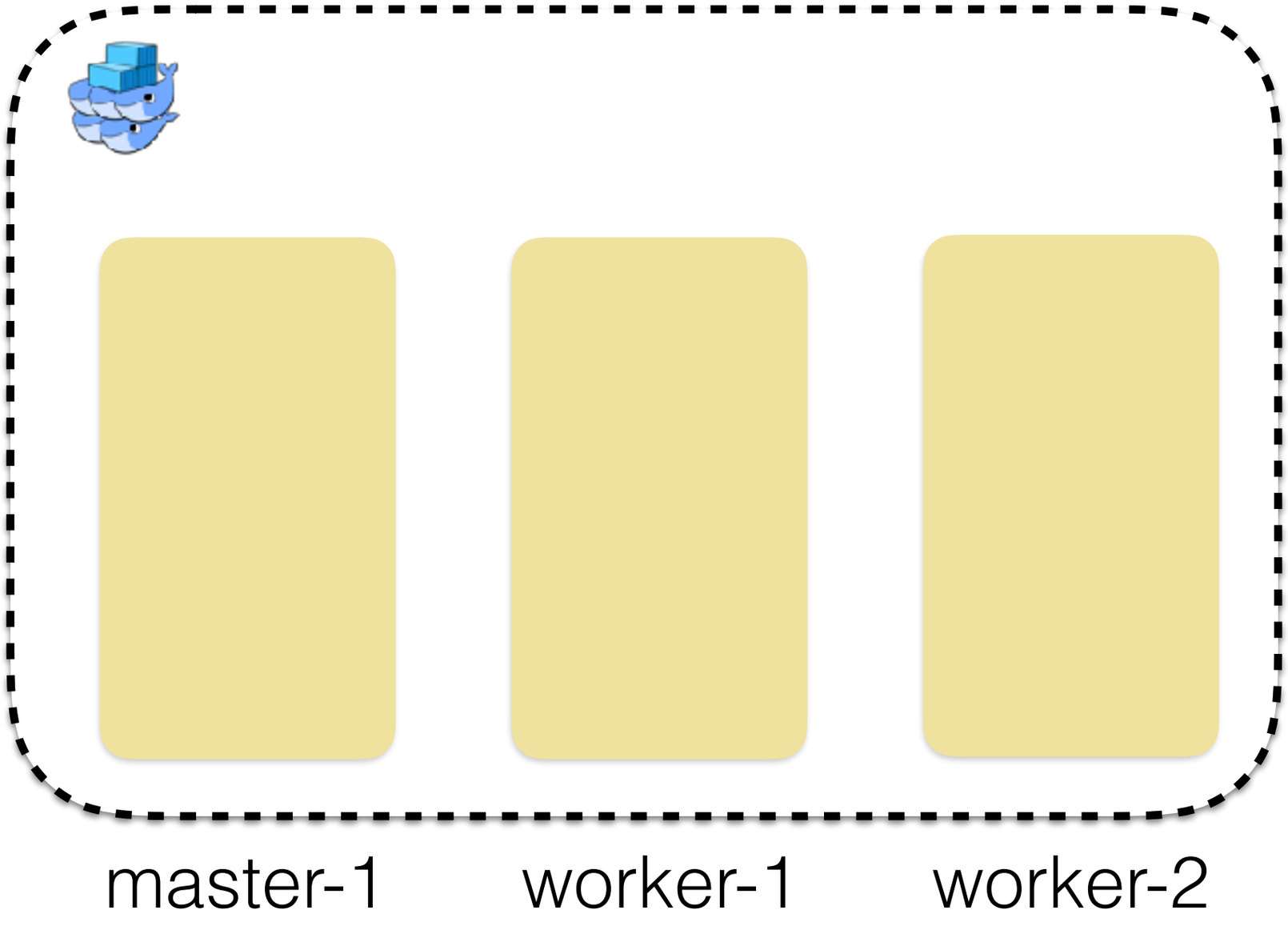
Проверим состояние кластера.

```
$ docker node ls
```

```
ID                            HOSTNAME            STATUS              AVAILABILITY        MANAGER STATUS
wzq2tosh1rqjg1cpthv6ofb0k *   master-1            Ready               Active              Leader
tosqmr3gyghdelxs85cowcrab     worker-1            Ready               Active
6dquf3w8ijuzq7dn035tn72lr     worker-2            Ready               Active
```

Строим Swarm Cluster

Теперь наш кластер выглядит вот так



Stack

- Сервисы и их зависимости объединяем в Stack
- Stack описываем в формате docker-compose (YML)

Управляем стеком с помощью команд:

```
$ docker stack deploy/rm/services/ls STACK_NAME
```

У нас уже есть первичное описание стека для запуска reddit-app в **docker-compose.yml**. Используйте свой или возьмите с gist.
Не забудьте определить файл **.env** (ссылка на gist)
P.S. Пока что используем только описание приложения и его зависимостей

Stack

```
version: '3.3'
services:
  mongo:
    image: mongo:${MONGO_VERSION}
    volumes:
      - mongo_data:/data/db
    networks:
      back_net:
        aliases:
          - post_db
          - comment_db

post:
  image: ${USER_NAME}/post:${POST_VERSION}
  networks:
    - front_net
    - back_net

comment:
  image: ${USER_NAME}/comment:${COMMENT_VERSION}
  networks:
    - front_net
    - back_net

  ui:
    image: ${USER_NAME}/ui:${UI_VERSION}
    ports:
      - "${UI_PORT}:9292/tcp"
    networks:
      - front_net

volumes:
  mongo_data: {}

networks:
  back_net: {}
  front_net: {}
```

Ссылка на gist

Stack

Проблема!

```
$ docker stack deploy  --compose-file docker-compose.yml ENV
```

не поддерживает переменные окружения и .env файлы

Workaround (ссылка на gist)

```
$ docker stack deploy --compose-file=<(docker-compose -f docker-
compose.yml config 2>/dev/null) DEV
```

имя стека

Stack

Посмотреть состояние стека:

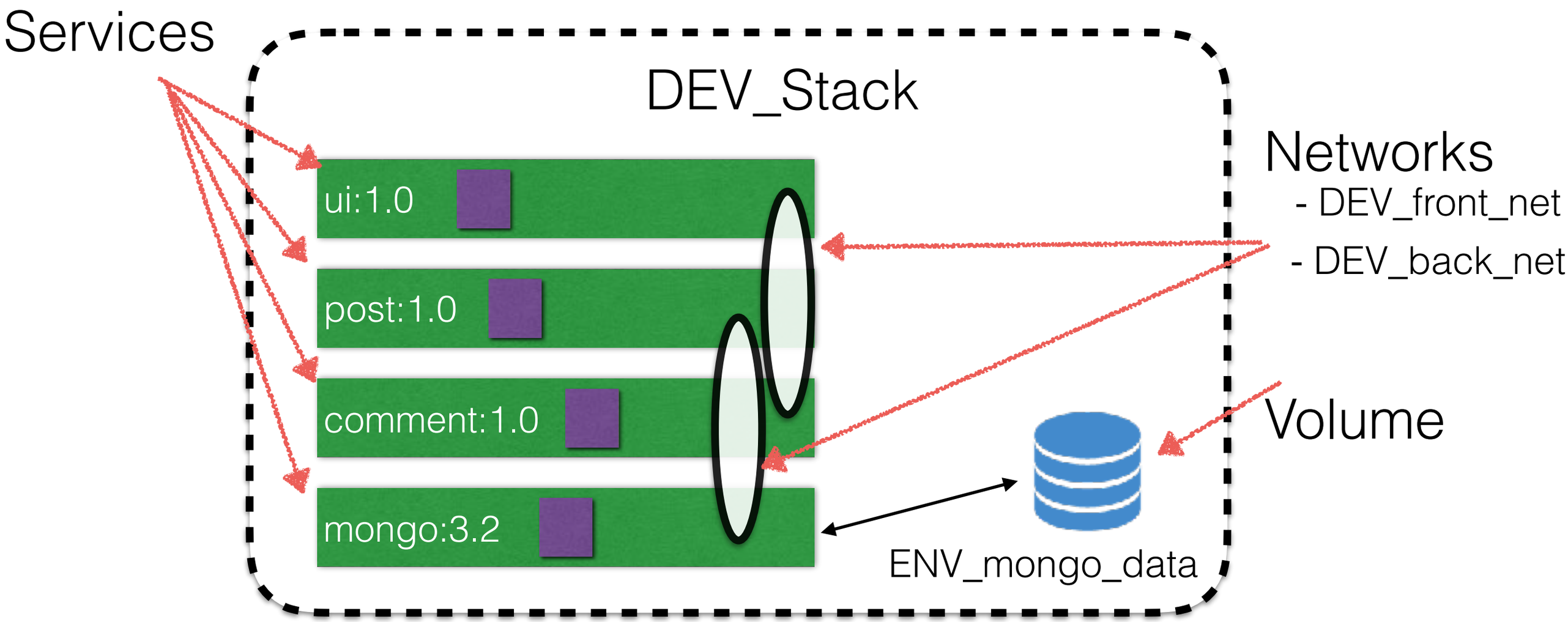
```
$ docker stack services DEV
```

Будете выведена своданая информация по сервисам (не по контейнерам)

```
ID             NAME         MODE         REPLICAS   IMAGE                  PORTS
buhy6df3tf50   DEV_post     replicated   1/1        chromko/post:latest
efvc9njr73m1   DEV_ui       replicated   1/1        chromko/ui:latest      *:9292->9292/tcp
mxlprrgh7p27   DEV_mongo    replicated   1/1        mongo:latest
yxxy4ivtyfxw   DEV_comment  replicated   1/1        chromko/comment:latest
```

Stack

Получим уже знакомую картину, но в Swarm-кластере



Размещаем сервисы

Ограничения размещения определяются с помощью логических действий со значениями label-ов (медатанных) **нод** и docker-**engine**’ов

Обращение к встроенным label’ам нод - **node.***

Обращение к заданным вручную label’ам нод - **node.labels***

Обращение к label’ам engine - **engine.labels.***

Примеры:

```
- node.labels.reliability == high
- node.role != manager
- engine.labels.provider == google
```

Labels

Добавим label к ноде (ссылка на gist)

```
$ docker node update --label-add reliability=high master-1
```

Swarm не умеет фильтровать вывод по label-ам нод пока что (ссылка на issue)

```
$ docker node ls --filter "label=reliability" - ничего не выдаст
```

Посмотреть label'ы всех нод можно так (ссылка на gist):

```
$ docker node ls -q | xargs docker node inspect   -f '{{ .ID }}
[{{ .Description.Hostname }}]: {{ .Spec.Labels }}'
```

Размещаем сервисы

Предположим, что нода master-1 надежнее и дороже, чем worker-ноды, поэтому поместим туда нашу базу.
Определим с помощью **placement constraints** ограничения размещения
(ссылка на gist)

```
mongo:
  image: mongo:${MONGO_VERSION}
  deploy:
    placement:
      constraints:
        - node.labels.reliability == high
…
```

Размещаем сервисы

Основную нагрузку пользователей reddit-app
будем принимать на worker-ноды, чтобы не перегружать master
с помощью label **node.role**

Каждому сервису (ui, post, comment) добавим placement
(ссылка на gist)

```
post:
    image: ${USER_NAME}/post:${POST_VERSION}
    deploy:
      placement:
        constraints:
          - node.role == worker
```

Размещаем сервисы

Деплоим (не забываем проверить в браузере работоспособность)

```
$ docker stack deploy …
```

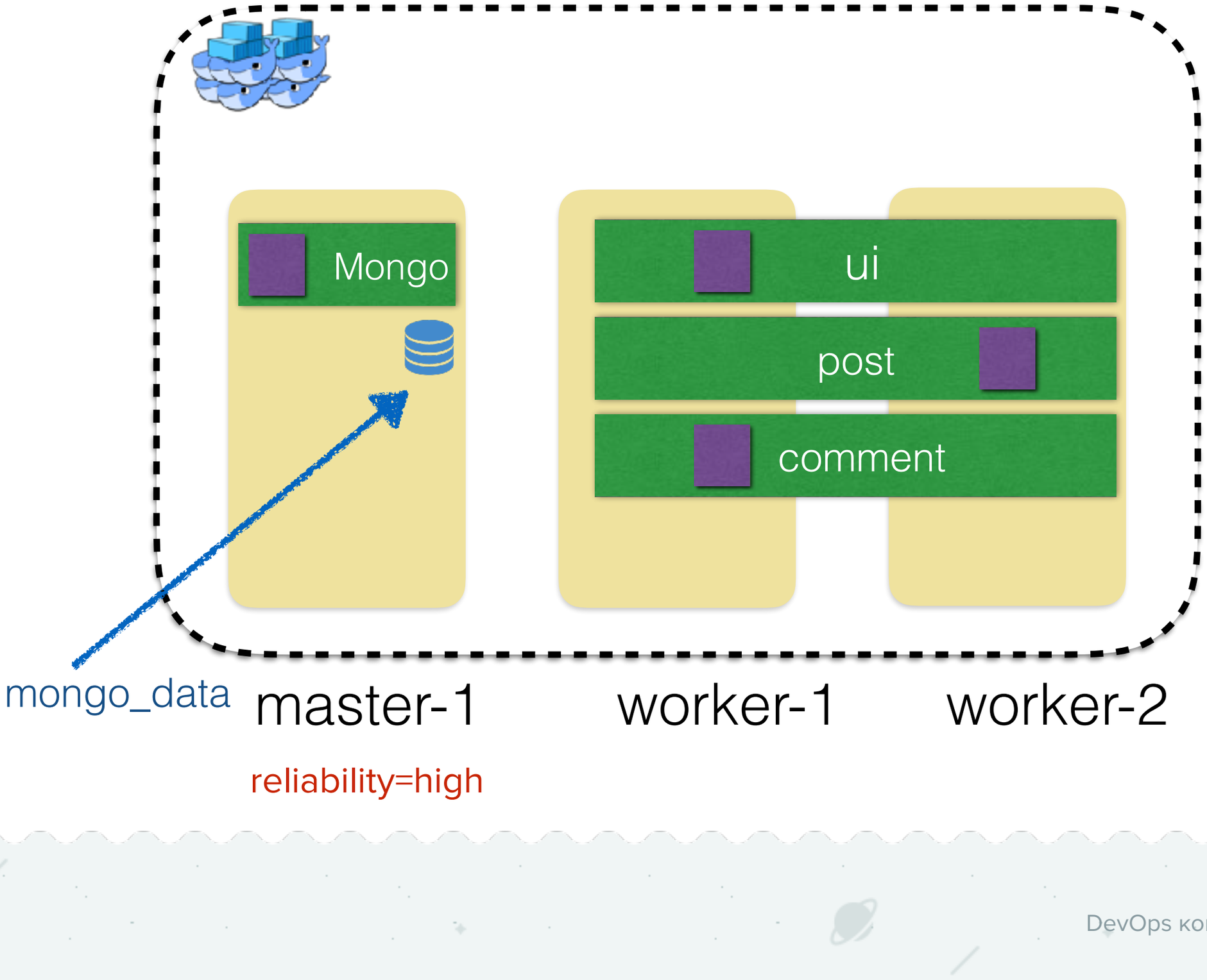
Посмотрим статусы текущих задач (конетейнеров) в стеке

```
$ docker stack ps DEV
```

Должны отобразиться 4 задачи
В поле **Node** задачи должны быть размещены согласно наложенным условиям:

- mongo - на master
- ui,post,comment - на worker-нодах

Будет примерно такая картина



Масштабируем сервисы

Для горизонтального масшатбирования и большей отказоустойчивости запустим микросервисы в нескольких экземплярах.

Существует 2 варианта запуска:

- replicated mode - запустить определенное число задач (default)
- global mode - запустить задачу на каждой ноде

!!! **Нельзя** заменить replicated mode на global mode (и обратно) без удаления сервиса

Масштабируем сервисы

Будем использовать **Replicated mode**

Запустим приложения ui, post и comment в **2**-х экземплярах (если хочется, можно больше)

(ссылка на gist)

```
ui:
    image: ${USER_NAME}/ui:${UI_VERSION}
    deploy:
      mode: replicated
      replicas: 2
```

не обязательно (т.к. это default)

Масштабируем сервисы

Деплоим (не забываем проверить в браузере работоспособность)

```
$ docker stack deploy …
```

Проверим что получилось

```
$ docker stack services DEV
```

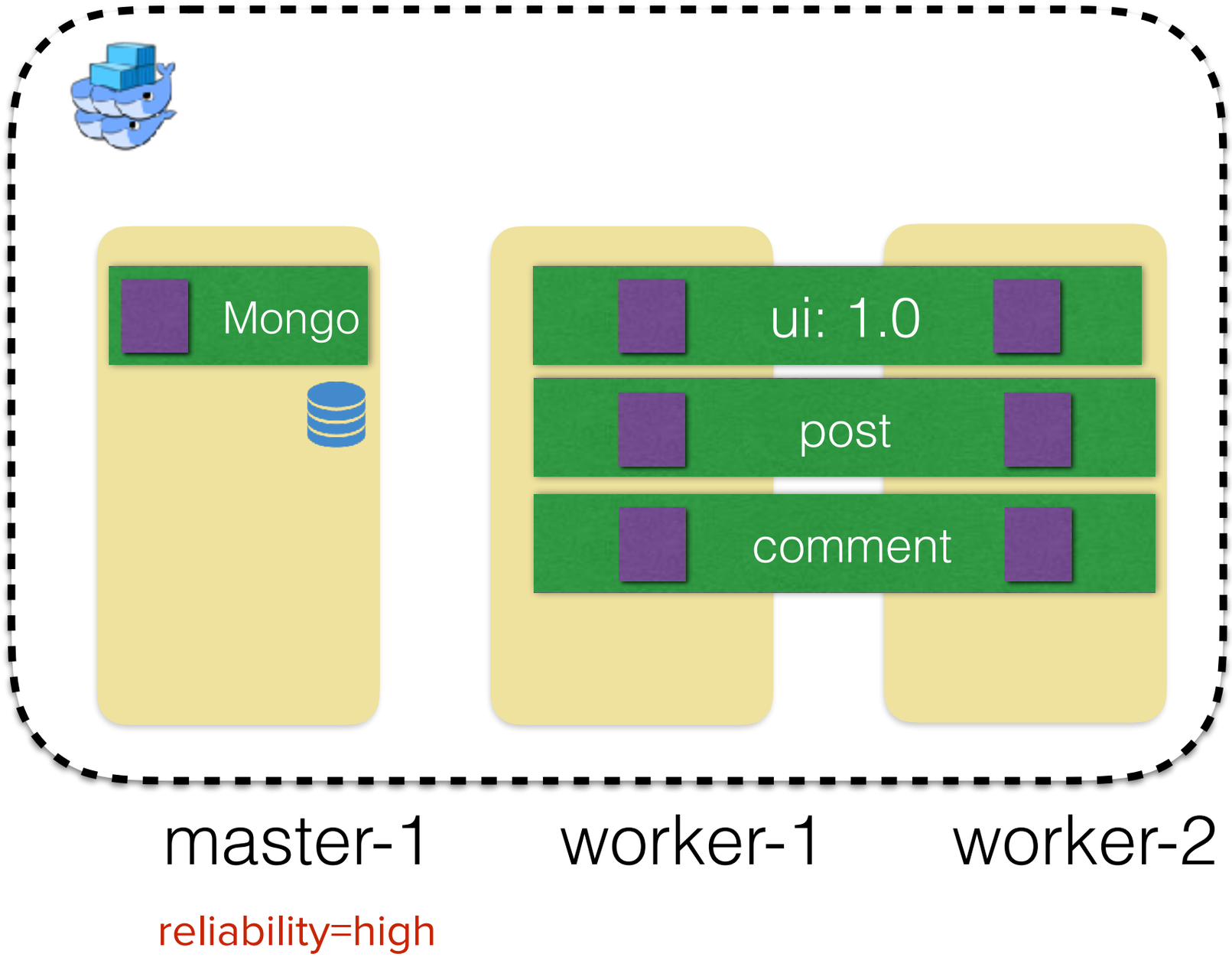
```
ID                  NAME                MODE                REPLICAS            IMAGE
PORTS
18d652ge5ajq        DEV_comment         replicated          2/2                 chromko/
comment:latest
…
```

Сервисы должны были распределиться равномерно по кластеру (стратегия spread) (проверяем)

```
$ docker stack ps DEV
```

```
$ docker stack ps DEV
```

Должно выглядеть так:



Масштабируем сервисы

Вы можете управлять кол-вом запускаемых сервисов в “на лету”

```
$ docker service scale DEV_ui=3
```

Или

```
$ docker service update --replicas 3 DEV_ui
```

Выключить все задачи сервиса:

```
$ docker service update --replicas 0 DEV_ui
```

Global

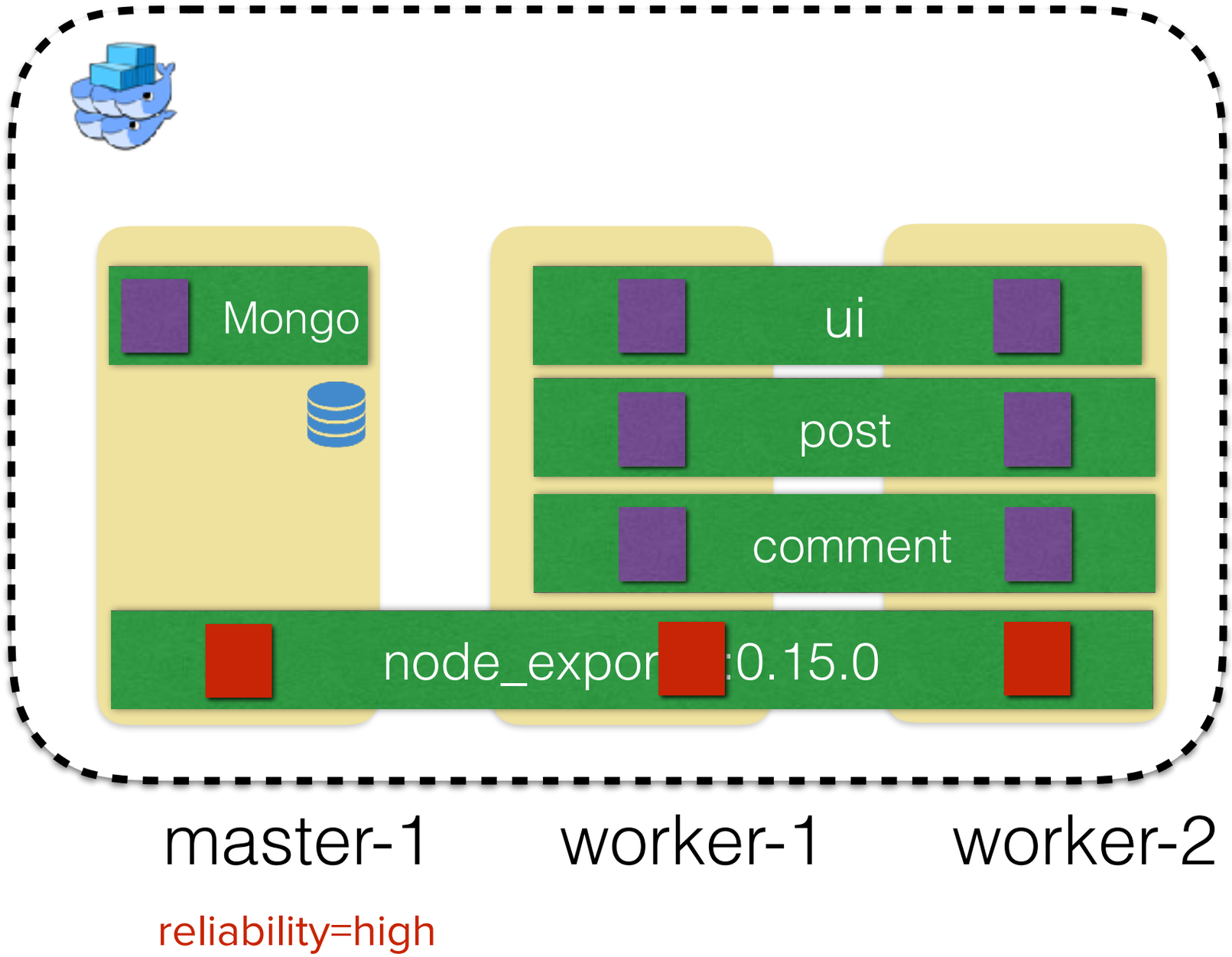
Для задач мониторинга кластера нам понадобится запускать node_exporter (только в 1-м экземпляре)
Используем **global mode**

(ссылка на gist)

```
node-exporter:
    image: prom/node-exporter:v0.15.0
    deploy:
      mode: global
        …
```

```
$ docker stack ps DEV
```

Должно выглядеть так:



Задание

1) Добавить в кластер еще 1 worker машину
2) Проследить какие контейнеры запустятся на ней
3) Увеличить число реплик микросервисов (3 - минимум)
4) Проследить какие контейнеры запустятся на новой машине. Сравнить с пунктом **2.**

Изменения должны быть отражены в compose-file.

* “Какой вывод можно сделать из результатов п.4?” - можете написать преподавателю **chromko** в slack.

Как мы общаемся с приложением?

У ui-компоненты приложения уже должен быть выставлен expose-порт, поэтому дополнять там ничего не нужно.

```
ui:
   image: ${USER_NAME}/ui:${UI_VERSION}
deploy:
  …
ports:
  - “${UI_PORT}:9292/tcp”
```

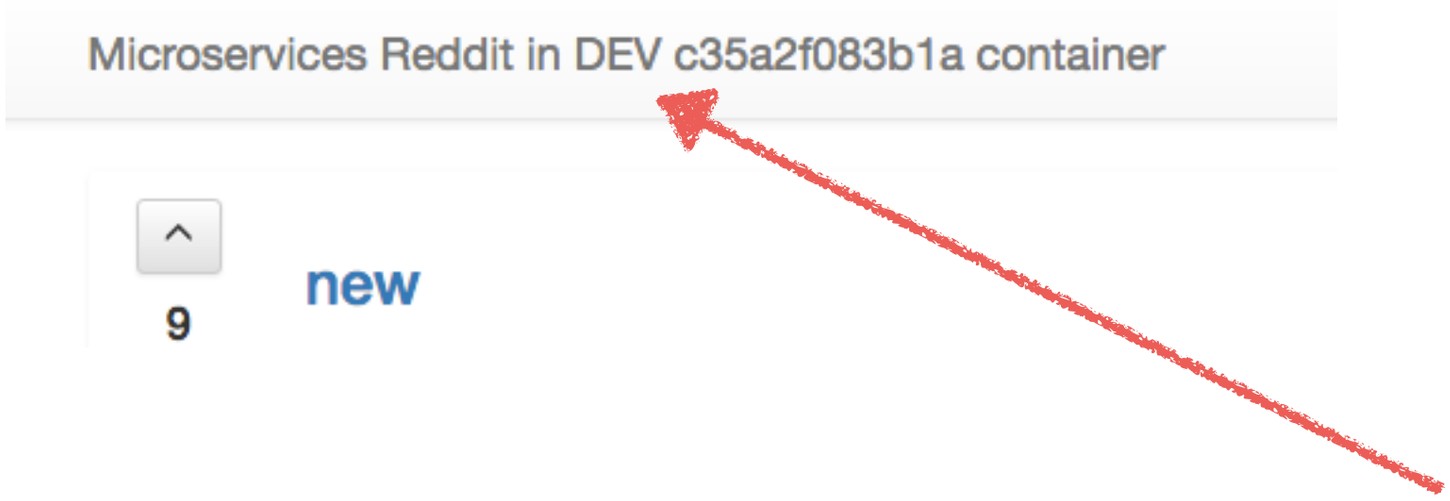
Как мы общаемся с приложением?

Однако отметим, что внутренний механизм Routing mesh обеспечивает балансировку запросов пользователей между контейнерами UI-сервиса.

1) В независимости от того, на какую ноду прийдет запрос, мы попадем на приложение (которое было опубликовано)
2) Каждое новое TCP/UDP-соединение будет отправлено на следующий endpoint (round-robin балансировка)

Как мы общаемся с приложением?

1) Посмотрите, где запущен UI-сервис:
   `$ docker service ps DEV_ui`
2) Получите список адресов:
   `$ docker-machine ip $(docker-machine ls  -q)`
3) Зайдите в браузере на каждую из машин (с интервалом в 10-15с)
4) Обратите внимание на id-контейнера



Чтобы отображалось имя окружения, добавьте переменную окружения **ENV** в сервис **ui** (ссылка на gist)

```
environment:
    - ENV=DEV
```

5) ID не сходятся, потому что рамках кластера минимальная единица - это задача (task). Контейнер - лишь конкретный экземпляр задачи.

ID контейнера можно увидеть, если выполнить

```
$ docker inspect $(docker stack ps DEV -q --filter "Name=DEV_ui.1") --format
"{{.Status.ContainerStatus.ContainerID}}"
```

Отфильтровать вывод по условию

Rolling Update

Чтобы обеспечить минимальное время простоя приложения во время обновлений (zero-downtime), сконфигурируем параметры деплоя параметром **update_config**

```
service:
   image: svc
deploy:
  update_config:
    parallelism: 2
    delay: 5s
    failure_action: rollback
    monitor: 5s
    max_failure_ratio: 2
    order: start-first
```

update_config

1) parallelism - сколько контейнеров (группу) обновить одновременно?
2) delay - задержка между обновлениями групп контейнеров
3) order - порядок обновлений (сначала убиваем старые и запускаем новые или наоборот) (только в compose 3.4)

Обработка ошибочных ситуаций

4) failure_action - что делать, если при обновлении возникла ошибка
5) monitor - сколько следить за обновлением, пока не признать его удачным или ошибочным
6) max_failure_ratio - сколько раз обновление может пройти с ошибкой перед тем, как перейти к failure_action

update_config

Определим, что приложение UI должно обновляться группами по 1 контейнеру с разрывом в 5 секунд.
В случае возникновения проблем деплой останавливается (Старые и новые версии работают вместе).

(ссылка на gist)

```
ui:
    image: ${USER_NAME}/ui:${UI_VERSION}
    deploy:
      replicas: 3
      update_config:
        delay: 5s
        parallelism: 1
        failure_action: pause
```

failure_action

Что делать, если обновление прошло с ошибкой?

- rollback - откатить все задачи на предыдущую версию
- pause (default) - приостановить обновление
- continue - продолжить обновление

```
ui:
  image: ${USER_NAME}/ui:${UI_VERSION}
  deploy:
    update_config:
      failure_action: rollback
```

Если вы перезаписали тег приложения рабочего, то **откатить его не получится**!!! Приложение будет сломано!

Rolling Update

Деплоим

```
$ docker stack deploy …
```

Отслеживаем изменения

```
$ docker service ps DEV_ui
```

Задание

Определить update_config для сервисов post и comment
так, чтобы они обновлялись группами по 2 сервиса с разрывом в 10 секунд, а в случае неудач осуществлялся rollback.

Все отразить в docker-compose.yml

Ограничиваем ресурсы

С помощью **resources limits** описываем максимум потребляемых приложениями CPU и памяти. Это обеспечит нам:

1) Представление о том, сколько ресурсов нужно приложению
2) Контроль Docker за тем, чтобы никто не превысил заданного порога (с помощью cgroups)
3) Защиту сторонних приложений от неконтролируемого расхода ресурса контейнером

```
ui:
   image: ${USER_NAME}/ui:${UI_VERSION}
   deploy:
     resources:
       limits:
         cpus: ‘0.25’
         memory: 150M
```

Максимум 150M памяти
и 25% процессорного времени

(ссылка на gist)

Задание

Задать ограничения ресурсов для сервисов post и comment, ограничив каждое в 300 мегабайт памяти и в 30% процессорного времени.

Все отразить в docker-compose.yml

Restart policy

Если контейнер в рамках задачи завершит свою работу, то планировщик Swarm автоматически запустит новый (даже если он вручную остановлен).

Мы можем поменять это поведение (для целей диагностики, например) так, чтобы контейнер перезапускался только при падении контейнера (on-failure).

По-умолчанию контейнер будет **бесконечно** перезапускаться. Это может оказать сильную нагрузку на машину в целом. Ограничим число попыток перезапуска 3-мя с интервалом в 3 секунды.

```
ui:
   image: ${USER_NAME}/ui:${UI_VERSION}
   deploy:
     restart_policy:
       condition: on-failure
        max_attempts: 3
        delay: 3s
```

(ссылка на gist)

Задание

Задайте политику перезапуска для **comment** и **post** сервисов так, чтобы Swarm пытался перезапустить их при падении с ошибкой 10-15 раз с интервалом в 1 секунду.

Все отразить в docker-compose.yml

Задание

Помимо сервисов приложения, у вас может быть инфраструктура, описанная в compose-файле (prometheus, node-exporter, grafana ...)

Нужно выделить ее в отдельный compose-файл. С названием **docker-compose.infra.yml.**
В него выносится все что относится к этим сервисам (volumes, services)

Запускать приложение вместе с инфрой можно следующей командой (ссылка на gist)

```
$ docker stack deploy --compose-file=<(docker-compose
-f docker-compose.infra.yml -f docker-compose.yml
config 2>/dev/null)  DEV
```

Задание

***

Как вы видите управление несколькими окружениями с помощью .env-файлов и compose-файлов в Swarm?

Создайте такие .env-файлы и параметризуйте что считаете нужным в compose-файлах.

Напишите команды, с помощью которых вы запустите эти несколько окружений рядом (в кластере) в README-файле.

Проверка

- Создайте PR с вашими наработками
- Пригласите одного из следующих преподавателей на review вашего PR:
  - chromko
  - Artemmkin
  - Nklya