В прошлой статье я рассказал о том, как установить кластер Kubernetes на свое железо. Сегодня я покажу, как можно с ним работать — запускать различные контейнеры, устанавливать лимиты, регулировать запуск и восстановление и т.д. По сути, опишу быстрый старт и настройку в kubernetes — на конкретных примерах покажу, с чего начинать освоение kubernetes и как его использовать на практике.

Если у вас есть желание научиться строить и поддерживать высокодоступные и надежные системы, рекомендую познакомиться с **онлайн-курсом «Администратор Linux»** в OTUS. Курс не для новичков, для поступления нужно пройти .

Содержание:

Цели статьи

1. Рассказать об основных абстракциях в kubernetes.
2. На примерах показать, как запускать, управлять контейнерами в кластере.
3. Описать механизм отказоустойчивости, реализованный в кубернетис.
4. Настроить обработку запросов в кластер из вне и запустить его в работу.
5. Задеплоить реальное микросервисное приложение.

Введение

Сразу поделюсь ссылкой на официальную документацию по kubernetes. Она хорошо структурирована и наполнена. Пользоваться ей удобно. С ее помощью настраивать кластер проще. Напоминаю, что мы будем работать с кластером, который установили по предыдущей статье — установка kubernetes. Перед тем, как начать работать с кластером, расскажу об одной полезной возможности. Есть команда:

```
# kubectl completion bash
```

Она формирует конфиг для настройки автодополнения команд в bash. Вывод этой команды нужно добавить в ваш *~/.bashrc* Можно это сделать автоматически.

```
# kubectl completion bash >> ~/.bashrc
```

Чтобы автодополнение работало, нужен пакет **bash-completion**.

```
# yum install bash-completion
```

Запуск контейнера nginx в kubernetes

Начнем с самого простого — создадим один pod, в котором запустим контейнер с последней версией nginx. **POD** в терминологии kubernetes — это набор контейнеров, объединенных между собой общим linux namespace. Самое важное тут то, что все контейнеры в одном поде связаны между собой сетью в

рамках общего localhost. Они не могут использовать один и тот же порт. По сути, кубернетис работает не с докер контейнерами, а именно с подами.

Создаем файл pod-nginx.yaml следующего содержания:

```
---
apiVersion: v1
kind: Pod
metadata:
  name: pod-nginx
spec:
  containers:
  - image: nginx:1.16
    name: nginx
    ports:
    - containerPort: 80
```

Внимательно следите за пробелами в начале строк. Нужны именно пробелы, а не табуляция. В Yaml файлах очень важна структура. Запускаем получившийся pod.

```
# kubectl create -f pod-nginx.yaml
pod/pod-nginx created
```

Сразу сделаю пояснение насчет команды **create**. Вместо нее можно, а чаще всего и нужно, так как удобнее, использовать команду **apply**. Create создает новую абстракцию. Если вы ее измените и снова создадите с тем же именем, то получите ошибку. Вам придется сначала ее удалить, а потом создать заново. Команда apply сначала проверяет наличие абстракции, в данном случае pod-nginx и если ее нет, то создает. А если она уже есть, то просто перезапускает существующую с новыми параметрами.

Проверяем, что получилось.

```
# kubectl get pod
NAME        READY   STATUS    RESTARTS   AGE
pod-nginx   1/1     Running   0          2m32s
```

В настоящий момент мы запустили в кластере kubernetes один контейнер с nginx. Существует команда **edit**, которая позволяет редактировать сущности кластера kubernetes в режиме реального времени. Применение изменений произойдет сразу же после сохранения изменений. Пример:

```
# kubectl edit pod pod-nginx
```

```
kubectl-edit-dutgw.yaml   [----]  0 L:[  1+ 0   1/ 90] *(0   /2687b) 0035 0x023
# Please edit the object below. Lines beginning with a '#' will be ignored,
# and an empty file will abort the edit. If an error occurs while saving this file will be
# reopened with the relevant failures.
#
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"v1","kind":"Pod","metadata":{"annotations":{},"name":"pod-nginx","namespace":"default"},"spec":{"containers":[{"image":"nginx:1.16","name":"nginx","ports":[{"containerPort":80}]}]}}
  creationTimestamp: "2019-09-24T13:17:40Z"
  name: pod-nginx
  namespace: default
  resourceVersion: "1301284"
  selfLink: /api/v1/namespaces/default/pods/pod-nginx
  uid: 733f27d5-5b90-4c66-93bf-6d68ac85dd3b
spec:
  containers:
  - image: nginx:1.16
    imagePullPolicy: IfNotPresent
    name: nginx
    ports:
    - containerPort: 80
      protocol: TCP
    resources: {}
    terminationMessagePath: /dev/termination-log
    terminationMessagePolicy: File
    volumeMounts:
    - mountPath: /var/run/secrets/kubernetes.io/serviceaccount
      name: default-token-2x2gw
      readOnly: true
  dnsPolicy: ClusterFirst
  enableServiceLinks: true
  nodeName: kub-node-2
  priority: 0
  restartPolicy: Always
  schedulerName: default-scheduler
  securityContext: {}
  serviceAccount: default
  serviceAccountName: default
  terminationGracePeriodSeconds: 30
  tolerations:
  - effect: NoExecute
    key: node.kubernetes.io/not-ready
    operator: Exists
    tolerationSeconds: 300
  - effect: NoExecute
    key: node.kubernetes.io/unreachable
    operator: Exists
    tolerationSeconds: 300
  volumes:
  - name: default-token-2x2gw
    secret:
      defaultMode: 420
      secretName: default-token-2x2gw
status:
  conditions:
```

serveradmin.ru

Вы увидите полный конфиг пода, который использует кластер. В нем намного больше информации, нежели в вашем yaml файле.

Удалить созданный pod можно следующей командой:

```
# kubectl delete pod pod-nginx
pod "pod-nginx" deleted
```

Или сразу все поды:

```
# kubectl delete pod --all
```

Отказоустойчивость подов с помощью ReplicaSet

Теперь создадим несколько подов с nginx. Для этого нужно использовать другую абстракцию кубернетис. **ReplicaSet** следит за количеством подов по шаблону, который мы указываем. В этом шаблоне мы можем указать метку нашего приложения, в моем примере это my-nginx, и количество запущенных реплик.

```
---
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: replicaset-nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: my-nginx
  template:
    metadata:
      labels:
        app: my-nginx
```

```
    spec:
      containers:
      - image: nginx:1.16
        name: nginx
        ports:
        - containerPort: 80
```

Запускаем replicaset.

```
# kubectl apply -f replicaset-nginx.yaml
```

Проверяем, что получилось.

```
# kubectl get replicaset
NAME               DESIRED   CURRENT   READY   AGE
replicaset-nginx   2         2         2       18m
```

Нам нужно было 2 реплики и мы их получили. С помощью **edit** мы можем на ходу редактировать replicaset, к примеру, добавляя количество реплик.

```
# kubectl edit replicaset replicaset-nginx
```

Репликасет сама следит за количеством запущенных подов. Если один из них по какой-то причине упадет или будет удален, она поднимет его заново. Можете проверить это сами, вручную удалив один из подов. Спустя некоторое время он появится снова.

```
# kubectl get pod
NAME                     READY   STATUS    RESTARTS   AGE
pod-nginx                1/1     Running   0          15m
replicaset-nginx-f87qf   1/1     Running   0          22m
replicaset-nginx-mr4kw   1/1     Running   0          22m

# kubectl delete pod replicaset-nginx-f87qf
```

```
pod "replicaset-nginx-f87qf" deleted

# kubectl get replicaset
NAME               DESIRED   CURRENT   READY   AGE
replicaset-nginx   2         2         2       23m

# kubectl get pod
NAME                     READY   STATUS    RESTARTS   AGE
pod-nginx                1/1     Running   0          16m
replicaset-nginx-g4l58   1/1     Running   0          14s
replicaset-nginx-mr4kw   1/1     Running   0          23m
```

Я удалил replicaset-nginx-f87qf, вместо него тут же был запущен новый replicaset-nginx-g4l58. Наглядный пример одного из механизмов отказоустойчивости в кластере kubernetes на уровне подов. Кубернетис будет следить за количеством реплик на основе их label. Если вы через replicaset указали запустить 2 реплики приложения с меткой my-nginx, ни меньше, ни больше подов с этой меткой вы запустить не сможете. Если вы вручную запустите pod с меткой my-nginx, он будет тут же завершен, если у вас уже есть 2 пода с такими метками от replicaset.

Replicaset запускает поды на разных нодах. Проверить это можно, посмотрев расширенную информацию о подах.

```
# kubectl get pod -o wide
NAME                     READY   STATUS    RESTARTS   AGE   IP            NODE         NOMINATED NODE   READINESS GATES
replicaset-nginx-cmfnh   1/1     Running   0          2m26s   10.233.67.6   kub-node-1   <none>           <none>
replicaset-nginx-vgxfl   1/1     Running   0          2m26s   10.233.68.4   kub-node-2   <none>           <none>
```

Таким образом, если у вас одна нода выйдет из строя, кластер автоматически запустит умершие поды на новых нодах. При этом, если нода вернется обратно с запущенными подами на ней, kubernetes автоматически прибьет лишние поды, чтобы их максимальное количество соответствовало информации из шаблона replicaset.

Удаляются replicaset так же как и поды.

```
# kubectl delete rs replicaset-nginx
replicaset.extensions "replicaset-nginx" deleted
```

Вместо длинного replicaset я использовал сокращение rs. Это бывает удобно для абстракций с длинными названиями. Для всех них кубернетис поддерживает сокращения.

Deployment — деплой в кластер

Переходим к следующей абстракции Kubernetes — **Deployment**. Они управляют наборами replicaset для непрерывного обновления подов. Покажу на примере, о чем идет речь. Допустим, у вас вышло обновление приложения в новом контейнере. Вам нужно не останавливая сервис выкатить обновление в прод. Если вы измените версию контейнера в шаблоне replicaset, автоматически он у вас не обновится. Да, если pod со старой версией контейнера упадет, новый будет создан уже с новой версией. Но все работающие поды останутся на старой версии.

Deployment как раз и нужен для управления репликасетами, задавая им стратегию обновления. У вас вышла новая версия контейнера, вы заменяете в текущем deployment версию контейнера и он по заранее настроенным правилам начинает перезапуск репликасетов и соответственно подов в них. Покажу на примере nginx, который мы откатим на предыдущую версию. Создаем yaml файл с deployment.

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deployment-nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: my-nginx
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
    type: RollingUpdate
  template:
```

```
    metadata:
      labels:
        app: my-nginx
    spec:
      containers:
      - image: nginx:1.16
        name: nginx
        ports:
        - containerPort: 80
```

Запускаем его.

```
# kubectl apply -f deployment-nginx.yaml
deployment.apps/deployment-nginx created
```

Смотрим список подов и репликасетов.

```
# kubectl get pod
NAME                                READY   STATUS    RESTARTS   AGE
deployment-nginx-785b6d8d9f-dr4cv   1/1     Running   0          55s
deployment-nginx-785b6d8d9f-m47tr   1/1     Running   0          55s
# kubectl get rs
NAME                          DESIRED   CURRENT   READY   AGE
deployment-nginx-785b6d8d9f   2         2         2       58s
```

Проверяем версию nginx в одном из подов.

```
# kubectl describe pod deployment-nginx-785b6d8d9f-dr4cv
```

```
[root@kub-master-1 ~]# kubectl describe pod deployment-nginx-785b6d8d9f-dr4cv
Name:           deployment-nginx-785b6d8d9f-dr4cv
Namespace:      default
Priority:       0
Node:           kub-node-1/10.1.4.32
Start Time:     Tue, 24 Sep 2019 17:53:46 +0300
Labels:         app=my-nginx
                pod-template-hash=785b6d8d9f
Annotations:    <none>
Status:         Running
IP:             10.233.67.8
Controlled By:  ReplicaSet/deployment-nginx-785b6d8d9f
Containers:
  nginx:
    Container ID:   docker://897bf9627bbcdbfe3874fdeae4f4927e9b1d452046fd158f52ee1a6cee1502d4
    Image:          nginx:1.16
    Image ID:       docker-pullable://nginx@sha256:0d0af9bc6ca2db780b532a522a885bef7fcaddd52d11817fc4cb6a3ead3eacc0
    Port:           80/TCP
    Host Port:      0/TCP
    State:          Running
      Started:      Tue, 24 Sep 2019 17:53:47 +0300
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-2x2gw (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Volumes:
  default-token-2x2gw:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-2x2gw
    Optional:    false
QoS Class:       BestEffort
Node-Selectors:  <none>
Tolerations:     node.kubernetes.io/not-ready:NoExecute for 300s
                 node.kubernetes.io/unreachable:NoExecute for 300s
Events:
  Type    Reason     Age    From                Message
  ----    ------     ----   ----                -------
  Normal  Scheduled  3m1s   default-scheduler   Successfully assigned default/deployment-nginx-785b6d8d9f-dr4cv to kub-node-1
  Normal  Pulled     2m47s  kubelet, kub-node-1  Container image "nginx:1.16" already present on machine
  Normal  Created    2m46s  kubelet, kub-node-1  Created container nginx
  Normal  Started    2m46s  kubelet, kub-node-1  Started container nginx
[root@kub-master-1 ~]#
```

serveradmin.ru

Теперь откатимся на предыдущую версию nginx 1.15. Для этого вносим изменения в **Deployment**.

```
# kubectl set image deployment deployment-nginx nginx=nginx:1.15
```

Проверяем список подов и репликасетов.

```
# kubectl get pod
NAME                                READY   STATUS    RESTARTS   AGE
deployment-nginx-68d778658b-fz5lt   1/1     Running   0          11s
deployment-nginx-68d778658b-mpkg5   1/1     Running   0          10s
[root@kub-master-1 ~]# kubectl get rs
NAME                          DESIRED   CURRENT   READY   AGE
deployment-nginx-68d778658b   2         2         2       15s
deployment-nginx-785b6d8d9f   0         0         0       5m57s
```

Название подов и репликасета изменились. Появился новый replicaset, а старый остался с погашенными подами, у него параметр **replicas** стал 0. Проверяем версию nginx в поде.

```
# kubectl describe pod deployment-nginx-68d778658b-fz5lt
```

```
[root@kub-master-1 ~]# kubectl describe pod deployment-nginx-68d778658b-fz51t
Name:           deployment-nginx-68d778658b-fz51t
Namespace:      default
Priority:       0
Node:           kub-node-2/10.1.4.33
Start Time:     Tue, 24 Sep 2019 17:58:54 +0300
Labels:         app=my-nginx
                pod-template-hash=68d778658b
Annotations:    <none>
Status:         Running
IP:             10.233.68.7
Controlled By:  ReplicaSet/deployment-nginx-68d778658b
Containers:
  nginx:
    Container ID:   docker://4e01fe5330lecc1e62f89b6e1099f687871c64b607a945496918b96d53e4e8d3
    Image:          nginx:1.15
    Image ID:       docker-pullable://nginx@sha256:23b4dcdf0d34d4a129755fc6f52e1c6e23bb34ea011b315d87e193033bcd1b68
    Port:           80/TCP
    Host Port:      0/TCP
    State:          Running
      Started:      Tue, 24 Sep 2019 17:58:56 +0300
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-2x2gw (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Volumes:
  default-token-2x2gw:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-2x2gw
    Optional:    false
QoS Class:       BestEffort
Node-Selectors:  <none>
Tolerations:     node.kubernetes.io/not-ready:NoExecute for 300s
                 node.kubernetes.io/unreachable:NoExecute for 300s
Events:
  Type    Reason     Age    From                Message
  ----    ------     ----   ----                -------
  Normal  Pulled     2m22s  kubelet, kub-node-2  Container image "nginx:1.15" already present on machine
  Normal  Created    2m22s  kubelet, kub-node-2  Created container nginx
  Normal  Started    2m21s  kubelet, kub-node-2  Started container nginx
  Normal  Scheduled  2m3s   default-scheduler    Successfully assigned default/deployment-nginx-68d778658b-fz51t to kub-node-2
[root@kub-master-1 ~]#
```

serveradmin.ru

Версия nginx изменилась во всех подах. Стратегия обновления описана в шаблоне деплоймента в разделе **strategy**. В данном случае используется тип **RollingUpdate**, когда deployment постепенно уменьшает количество реплик старой версии и увеличивает реплики новой версии, пока они все не заменят старые. При этом replicaset с предыдущей версией контейнера осталась. Она не удаляется для того, чтобы можно было потом оперативно вернуться на предыдущую версию, если с новой будет что-то не так. Для этого достаточно будет по очереди погасить поды новой replicaset и запустить старые. Делается это следующим образом.

```
# kubectl rollout undo deployment deployment-nginx
deployment.extensions/deployment-nginx rolled back
```

Проверяем наши replicaset.

```
# kubectl get rs
NAME                          DESIRED   CURRENT   READY   AGE
deployment-nginx-68d778658b   0         0         0       12m
deployment-nginx-785b6d8d9f   2         2         2       18m
```

Видим, что был снова запущен предыдущий replicaset с прошлой версией контейнера. По-умолчанию хранятся 10 версий прошлых replicaset.

Проверки доступности Probes

С деплоем и перезапуском подов не все так просто. Есть тяжелые приложения, которые стартуют очень долго, либо зависят от других приложений. Прежде чем погасить старую версию, нужно убедиться, что новая уже запущена и готова к работе. Для этого существует **liveness** и **readiness** проверки. Покажу на примере. Берем предыдущий deployment и добавляем туда проверки.

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deployment-nginx
spec:
  replicas: 2
```

```
  selector:
    matchLabels:
      app: my-nginx
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: my-nginx
    spec:
      containers:
      - image: nginx:1.16
        name: nginx
        ports:
        - containerPort: 80
        readinessProbe:
          failureThreshold: 5
          httpGet:
            path: /
            port: 80
          periodSeconds: 10
          successThreshold: 2
          timeoutSeconds: 3
        livenessProbe:
          failureThreshold: 3
          httpGet:
            path: /
            port: 80
          periodSeconds: 10
          successThreshold: 1
```

```
        timeoutSeconds: 3
        initialDelaySeconds: 10
```

В данном примере используется проверка httpGet по корневому урлу на порт 80.

- **readinessProbe** проверяет способность приложения начать принимать трафик. Она делает 5 проверок (failureThreshold). Если хотя бы 2 (successThreshold) из них будут удачными, считается, что приложение готово. Метод httpGet проверяет код ответа веб сервера. Если он 200 или 3xx, то считается, что все в порядке. Эта проверка выполняется до тех пор, пока не будет выполнено заданное на успех условие — successThreshold. После этого прекращается.
- **livenessProbe** выполняется постоянно, следя за приложением во время его жизни. В моем примере проверка будет неудачной, если 3 (failureThreshold) проверки подряд провалились. При этом, если хотя бы одна (successThreshold) будет удачной, то счетчик неудачных сбрасывается. Параметр initialDelaySeconds задает задержку после старта пода для начала **liveness** проверок.

Вот еще один пример liveness проверки, но уже по наличию файла. Проверяется файл */tmp/healthy*, если он существует, проверка удачна, если его нет, то ошибка.

```
    livenessProbe:
      exec:
        command:
        - cat
        - /tmp/healthy
      initialDelaySeconds: 5
      periodSeconds: 5
```

Создавать этот файл может само приложение во время работы.

Resources — настройка ресурсов

Расскажу, как ограничиваются выделяемые для подов вычислительные ресурсы. Речь идет про CPU и Оперативную память. Задать верхнюю планку использования ресурсов можно с помощью **Limits**. А с помощью **Requests** мы можем зарезервировать необходимые ресурсы для пода на ноде. Если в Requests у пода параметры выше, чем есть свободных ресурсов у ноды, то под не сможет приехать на эту ноду.

Важно понимать, что реквесты никак не следят за реальным использованием ресурсов. То есть это просто пожелание к ресурсам ноды, где будет размещаться под. При этом после размещения он сможет занять ресурсов больше, чем указано в **Requests**. Кластер kubernetes за этим не следит. Если реквесты вообще не указать, то под может приехать на ноду, где свободно очень мало ресурсов, а ему для работы надо больше. В итоге он будет падать. Таким образом, requests используются для планирования ресурсов кластера.

Далее пример деплоймента с указанными параметрами ресурсов. Дополняем предыдущие примеры.

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deployment-nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: my-nginx
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: my-nginx
    spec:
      containers:
      - image: nginx:1.16
        name: nginx
        ports:
        - containerPort: 80
        readinessProbe:
```

```
        failureThreshold: 5
        httpGet:
          path: /
          port: 80
        periodSeconds: 10
        successThreshold: 2
        timeoutSeconds: 3
      livenessProbe:
        failureThreshold: 3
        httpGet:
          path: /
          port: 80
        periodSeconds: 10
        successThreshold: 1
        timeoutSeconds: 3
        initialDelaySeconds: 10
      resources:
        requests:
          cpu: 100m
          memory: 256Mi
        limits:
          cpu: 300m
          memory: 512Mi
```

Память выделяется в мегабайтах, а вот CPU в милли cpu, что равно 1/1000 от процессоронго ядра. То есть 1000 миллицпу это одно ядро процессора ноды. Запустим наш deployment и посмотрим на один из подов.

```
# kubectl describe pod deployment-nginx-79cd6dc79c-rlhtg
```

```
[root@kub-master-1 ~]# kubectl describe pod deployment-nginx-79cd6dc79c-rlhtg
Name:           deployment-nginx-79cd6dc79c-rlhtg
Namespace:      default
Priority:       0
Node:           kub-node-1/10.1.4.32
Start Time:     Wed, 25 Sep 2019 15:54:54 +0300
Labels:         app=my-nginx
                pod-template-hash=79cd6dc79c
Annotations:    <none>
Status:         Running
IP:             10.233.67.14
Controlled By:  ReplicaSet/deployment-nginx-79cd6dc79c
Containers:
  nginx:
    Container ID:   docker://86dd2f353e0979a724ae81841cf938212c8637e954ff8ec38059e4905fad103e
    Image:          nginx:1.16
    Image ID:       docker-pullable://nginx@sha256:0d0af9bc6ca2db780b532a522a885bef7fcaddd52d11817fc4cb6a3ead3eacc0
    Port:           80/TCP
    Host Port:      0/TCP
    State:          Running
      Started:      Wed, 25 Sep 2019 15:54:55 +0300
    Ready:          True
    Restart Count:  0
    Limits:
      cpu:     300m
      memory:  512Mi
    Requests:
      cpu:        100m
      memory:     256Mi
    Liveness:     http-get http://:80/ delay=10s timeout=3s period=10s #success=1 #failure=3
    Readiness:    http-get http://:80/ delay=0s timeout=3s period=10s #success=2 #failure=5
    Environment:  <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-2x2gw (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Volumes:
  default-token-2x2gw:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-2x2gw
    Optional:    false
QoS Class:       Burstable
Node-Selectors:  <none>
Tolerations:     node.kubernetes.io/not-ready:NoExecute for 300s
                 node.kubernetes.io/unreachable:NoExecute for 300s
Events:
  Type    Reason     Age    From                Message
  ----    ------     ----   ----                -------
  Normal  Scheduled  6m16s  default-scheduler   Successfully assigned default/deployment-nginx-79cd6dc79c-rlhtg to kub-node-1
  Normal  Pulled     6m2s   kubelet, kub-node-1  Container image "nginx:1.16" already present on machine
  Normal  Created    6m2s   kubelet, kub-node-1  Created container nginx
  Normal  Started    6m2s   kubelet, kub-node-1  Started container nginx
[root@kub-master-1 ~]#
```

serveradmin.ru

Вот они, наши проверки и лимиты с реквестами.

Монтирование конфигов через ConfigMap

Абстракция kubernetes **configmap** придумана для того, чтобы отвязать конфиги контейнеров docker от deployment, чтобы не раздувать их размер. К примеру, нам нужно во все контейнеры с nginx положить конфиг default.conf. Мы для этого создаем configmap, в нем настраиваем желаемый конфиг и потом подключаем его в deployment. Показываю на примере. Создаем файл *configmap.yaml*.

```
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: configmap-nginx
data:
  default.conf: |
    server {
        listen       80 default_server;
        server_name  _;

        default_type text/plain;

        location / {
            return 200 '$hostname\n';
        }
    }
```

В данном случае это простейший конфиг для nginx, который при обращении к серверу будет отдавать 200-й код ответа и имя сервера. Теперь подключаем его к нашему deployment из предыдущих примеров.

```
---
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
  name: deployment-nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: my-nginx
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: my-nginx
    spec:
      containers:
      - image: nginx:1.16
        name: nginx
        ports:
        - containerPort: 80
        readinessProbe:
          failureThreshold: 5
          httpGet:
            path: /
            port: 80
          periodSeconds: 10
          successThreshold: 2
          timeoutSeconds: 3
        livenessProbe:
          failureThreshold: 3
          httpGet:
```

```
          path: /
          port: 80
        periodSeconds: 10
        successThreshold: 1
        timeoutSeconds: 3
        initialDelaySeconds: 10
      resources:
        requests:
          cpu: 100m
          memory: 256Mi
        limits:
          cpu: 300m
          memory: 512Mi
      volumeMounts:
      - name: config
        mountPath: /etc/nginx/conf.d/
    volumes:
    - name: config
      configMap:
        name: configmap-nginx
```

Мы создаем монтирование с именем config по пути */etc/nginx/conf.d/* и связываем это монтирование с configmap, который ранее создали. Теперь применяем сначала configmap, а затем deployment.

```
# kubectl apply -f configmap.yaml
configmap/configmap-nginx created
# kubectl apply -f deployment-nginx-confmap.yaml
deployment.apps/deployment-nginx configured
```

Проверяем, что получилось. Я подключусь к одному из подов и прочитаю там конфигурацию nginx.

```
[root@kub-master-1 ~]# kubectl get pod
NAME                                READY   STATUS    RESTARTS   AGE
deployment-nginx-848cc4c754-cxx2z   1/1     Running   0          2m15s
deployment-nginx-848cc4c754-w7q9s   1/1     Running   0          2m15s
[root@kub-master-1 ~]# kubectl exec deployment-nginx-848cc4c754-cxx2z cat /etc/nginx/conf.d/default.conf
server {
    listen       80 default_server;
    server_name  _;

    default_type text/plain;

    location / {
        return 200 '$hostname\n';
    }
}
[root@kub-master-1 ~]#
```

serveradmin.ru

Видим, что применился наш configmap. На втором поде будет то же самое.

Проброс порта в pod

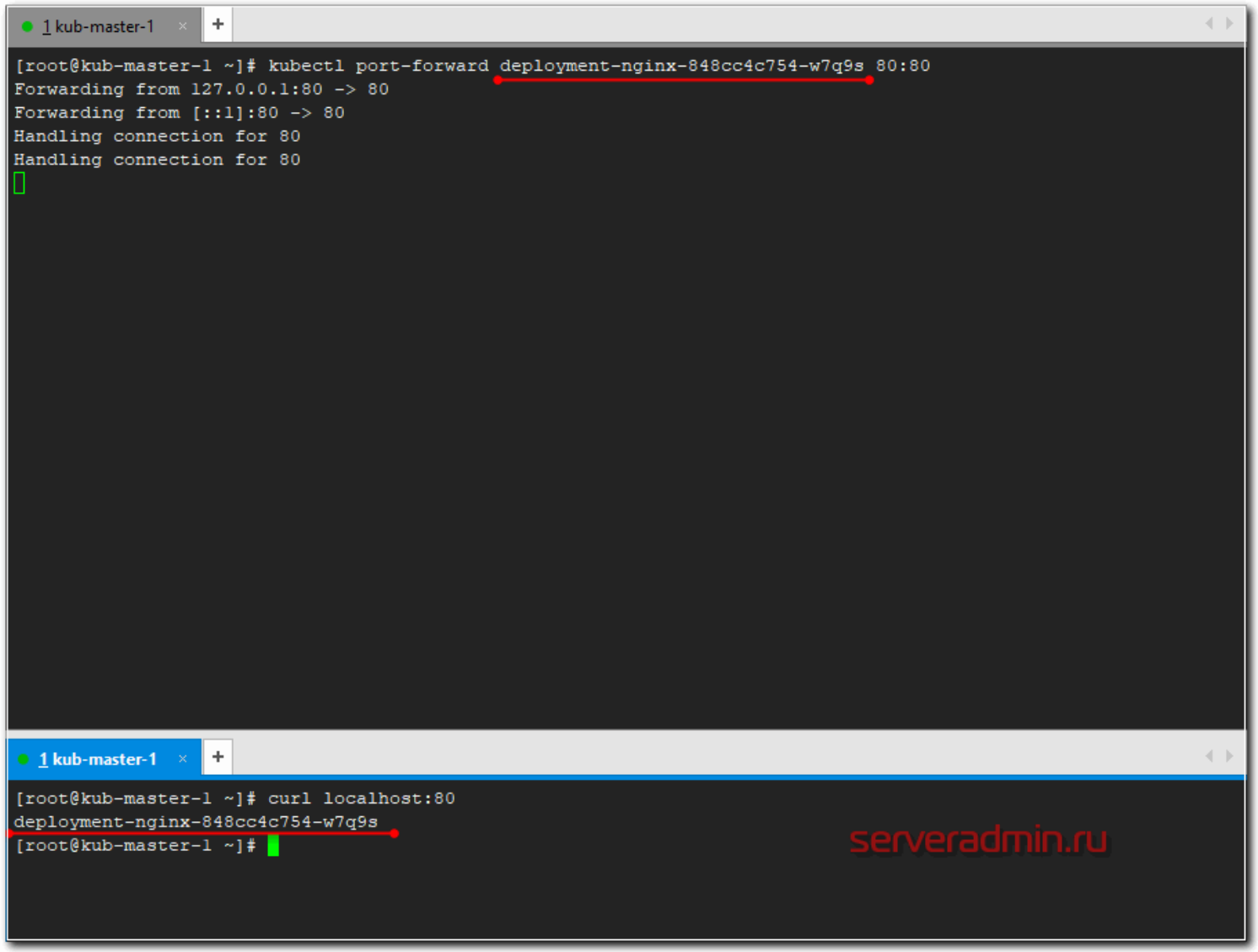
А сейчас пробросим 80-й порт мастера в конкретный под и проверим, что nginx действительно работает в соответствии с установленным конфигом. Делается это следующим обарзом.

```
# kubectl port-forward deployment-nginx-848cc4c754-w7q9s 80:80
Forwarding from 127.0.0.1:80 -> 80
Forwarding from [::1]:80 -> 80
```

Перемещаемся в соседнюю консоль мастера и там проверяем через curl.

```
# curl localhost:80
deployment-nginx-848cc4c754-w7q9s
```

Если сделать проброс в другой под и проверить подключение, вы получите в ответ на запрос curl на 80-й порт мастера имя второго пода. На практике, я не знаю, как можно использовать данную возможность. А вот для тестов в самый раз.


1 kub-master-1
[root@kub-master-1 ~]# kubectl port-forward deployment-nginx-848cc4c754-w7q9s 80:80
Forwarding from 127.0.0.1:80 -> 80
Forwarding from [::1]:80 -> 80
Handling connection for 80
Handling connection for 80
1 kub-master-1
[root@kub-master-1 ~]# curl localhost:80
deployment-nginx-848cc4c754-w7q9s
[root@kub-master-1 ~]#
serveradmin.ru

Service — балансировка сети в kubernetes

Service в Kubernetes по своей сути это некий балансировщик на уровне L3. С помощью сервисов мы можем попасть в нужные нам поды, обратившись к ним по имени или по ip адресу. Перераспределение запросов идет по алгоритму round-robin. Давайте сделаем сервис к нашему deployment из предыдущих примеров.

```
---
apiVersion: v1
kind: Service
metadata:
  name: service-nginx
spec:
  ports:
  - port: 80
    targetPort: 80
  selector:
    app: my-nginx
  type: ClusterIP
```

Параметр **ClusterIP** означает, что будет использоваться виртуальная сеть кластера. Доступ к сервису будет доступен только внутри кластера. Извне доступ закрыт. С помощью selector app my-nginx мы привязали сервис к приложению с именем my-nginx, которое мы использовали в предыдущих примерах. Применяем service в кластере.

```
# kubectl apply -f service.yaml
service/service-nginx created
```

И смотрим, что получилось.

```
# kubectl get service
NAME            TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
kubernetes      ClusterIP   10.233.0.1     <none>        443/TCP   7d12h
```

```
service-nginx   ClusterIP   10.233.43.79   <none>        80/TCP    29s
```

Чтобы проверить работу сети в кластере, удобно запустить в нем специальный docker образ, где внутри собраны всякие сетевые утилиты, с помощью которых можно попинговать, посмотреть маршруты и т.д. Можете какой-то свой браз для этого использовать, или воспользоваться готовым — **amouat/network-utils**. Запустим его напрямую в кластере с параметрами, чтобы он удалился сразу после выхода из него. Ключи здесь такие же как в докере.

```
# kubectl run -t -i --rm --image amouat/network-utils testnet bash
kubectl run --generator=deployment/apps.v1 is DEPRECATED and will be removed in a future version. Use kubectl run --
generator=run-pod/v1 or kubectl create instead.
If you don't see a command prompt, try pressing enter.
```

Нажимайте enter и попадете в контейнер. А дальше пробуйте пинговать.

```
[root@kub-master-1 ~]# kubectl run -t -i --rm --image amouat/network-utils testnet bash
kubectl run --generator=deployment/apps.v1 is DEPRECATED and will be removed in a future version. Use kubectl run --generator=run-pod/v1 or kubectl create instead.
If you don't see a command prompt, try pressing enter.
root@testnet-9d4b6c988-8lmv8:/#
root@testnet-9d4b6c988-8lmv8:/# curl service-nginx
deployment-nginx-848cc4c754-cxx2z
root@testnet-9d4b6c988-8lmv8:/# curl service-nginx
deployment-nginx-848cc4c754-w7q9s
root@testnet-9d4b6c988-8lmv8:/# curl service-nginx
deployment-nginx-848cc4c754-w7q9s
root@testnet-9d4b6c988-8lmv8:/# curl service-nginx
deployment-nginx-848cc4c754-w7q9s
root@testnet-9d4b6c988-8lmv8:/# curl service-nginx
deployment-nginx-848cc4c754-cxx2z
root@testnet-9d4b6c988-8lmv8:/# curl service-nginx
deployment-nginx-848cc4c754-w7q9s
root@testnet-9d4b6c988-8lmv8:/# curl service-nginx
deployment-nginx-848cc4c754-w7q9s
root@testnet-9d4b6c988-8lmv8:/# curl service-nginx
deployment-nginx-848cc4c754-cxx2z
root@testnet-9d4b6c988-8lmv8:/#
```

serveradmin.ru

После выхода из контейнера, он удалится.

```
# exit
exit
Session ended, resume using 'kubectl attach testnet-9d4b6c988-8lmv8 -c testnet -i -t' command when the pod is running
deployment.apps "testnet" deleted
```

Настройка Ingress

Описанные выше сервисы решают задачу взаимодействия внутри кластера kubernetes, но нам нужно еще и с внешними пользователями взаимодействовать. Для этого настроим **Ingress**, который мы установили ранее в виде отдельной роли на ноде. По своей сути это обычный nginx, который будет получать конфигурацию из yaml файла. Рисуем конфиг для ingress, который будет пробрасывать запросы из вне на сервис service-nginx, который мы создали на предыдущем шаге.

```
---
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-nginx
spec:
  rules:
  - host: nginx.cluster.local
    http:
      paths:
      - backend:
          serviceName: service-nginx
          servicePort: 80
```

Загружаем конфиг в кластер кубернетиса.

```
# kubectl apply -f ingress.yaml
ingress.extensions/ingress-nginx configured
```

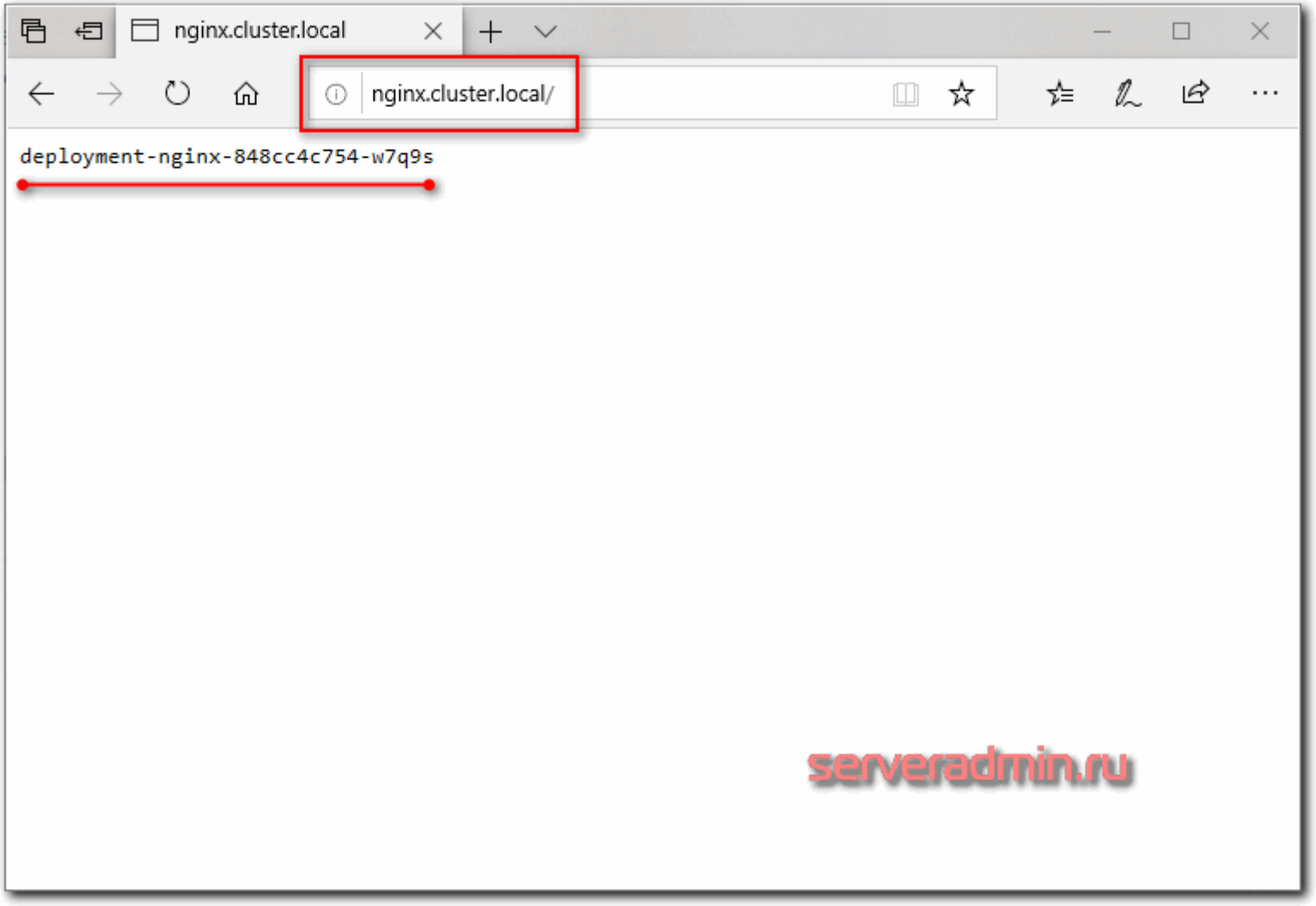
Смотрим, что получилось.

```
# kubectl get ingress -o wide
NAME            HOSTS                 ADDRESS     PORTS   AGE
ingress-nginx   nginx.cluster.local   10.1.4.39   80      3m20s
```

10.1.4.39 — ip адрес ноды ingress. В принципе, это сразу же может быть внешний ip, который будет принимать на себя все запросы. Так как это nginx, должно быть безопасно и надежно. На практике, я не знаю, делают ли так, но не вижу каких-то весомых причин в типовых сценариях этого не делать. Чтобы проверить работу ingress, нам надо добавить dns запись.

```
10.1.4.39 nginx.cluster.local
```

Я просто в локальный hosts машины добавил и проверил.



Если обновлять страничку, имя пода будет меняться в соответствии с настройкой балансировки. Она выполняется по алгоритму least_conn.

На этом я прерываюсь и заканчиваю текущее повествование по работе с кластером kubernetes. В таком виде в нем уже можно осмысленно что-то запускать и эксплуатировать. Надеюсь, вам было хоть немного понятно и вы сможете начать экспериментировать с кластером и пытаться на нем запускать какую-то полезную нагрузку. В таком виде он уже пригоден к ограниченной эксплуатации.

Deploy реального приложения в кластер

Давайте теперь на реальноми примере попробуем что-то запустить в кластере kubernetes. Я предлагаю для этого использовать демо магазин носков из этого репозитория — https://github.com/microservices-demo/microservices-demo. Там есть длиннющий yaml файл, который содержит в себе все необходимое (deployments, service и т.д.) для запуска магазина. Магазин состоит из множества компонентов, так что мы на практике убедимся, как легко и быстро можно деплоить сложные приложения в кластер.

Магазин настроен на работе в отдельном namespace — sock-shop. Его предварительно надо создать.

```
# kubectl create namespace sock-shop
```

Запускаем деплой всего проекта одной командой.

```
# kubectl apply -n sock-shop -f
"https://raw.githubusercontent.com/microservices-demo/microservices-demo/master/deploy/kubernetes/complete-demo.yaml"
```

Наблюдать за поднятием подов можно командой в реальном времени.

```
# kubectl get pods -n sock-shop -w
```

После того, как они все станут Running можно проверять работу. В этом проекте не используется ingress, поэтому чтобы понять, как подключиться к магазину, надо провести небольшое расследование. Для начала посмотрим запущенные service.

```
# kubectl get service -n sock-shop -o wide
```

```
[root@kub-master-1 ~]# kubectl get service -n sock-shop -o wide
NAME           TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE     SELECTOR
carts          ClusterIP   10.233.47.168   <none>        80/TCP           8m33s   name=carts
carts-db       ClusterIP   10.233.53.68    <none>        27017/TCP        8m33s   name=carts-db
catalogue      ClusterIP   10.233.29.109   <none>        80/TCP           8m33s   name=catalogue
catalogue-db   ClusterIP   10.233.30.68    <none>        3306/TCP         8m33s   name=catalogue-db
front-end      NodePort    10.233.62.23    <none>        80:30001/TCP     8m33s   name=front-end
orders         ClusterIP   10.233.59.85    <none>        80/TCP           8m32s   name=orders
orders-db      ClusterIP   10.233.18.9     <none>        27017/TCP        8m32s   name=orders-db
payment        ClusterIP   10.233.63.245   <none>        80/TCP           8m32s   name=payment
queue-master   ClusterIP   10.233.19.127   <none>        80/TCP           8m32s   name=queue-master
rabbitmq       ClusterIP   10.233.16.51    <none>        5672/TCP         8m32s   name=rabbitmq
shipping       ClusterIP   10.233.41.96    <none>        80/TCP           8m32s   name=shipping
user           ClusterIP   10.233.59.72    <none>        80/TCP           8m31s   name=user
user-db        ClusterIP   10.233.7.227    <none>        27017/TCP        8m32s   name=user-db
```

serveradmin.ru

Нас интересует тип **NodePort**, так как к нему можно подключаться из вне. Видим, что порт используется 30001 и имя приложения front-end. Посмотрим, где оно запущено.

```
# kubectl get pod -n sock-shop -o wide
```

```
[root@kub-master-1 ~]# kubectl get pod -n sock-shop -o wide
NAME                            READY   STATUS    RESTARTS   AGE     IP              NODE         NOMINATED NODE   READINESS GATES
carts-56c6fb966b-kgvq4          1/1     Running   0          8m44s   10.233.67.18    kub-node-1   <none>           <none>
carts-db-5678cc578f-6g9p4       1/1     Running   0          8m44s   10.233.68.16    kub-node-2   <none>           <none>
catalogue-644549d46f-lhgqn      1/1     Running   0          8m44s   10.233.67.19    kub-node-1   <none>           <none>
catalogue-db-6ddc796b66-qjd6l   1/1     Running   0          8m44s   10.233.68.17    kub-node-2   <none>           <none>
front-end-5594987df6-qgggw      1/1     Running   0          8m44s   10.233.68.18    kub-node-2   <none>           <none>
orders-749cdc8c9-9xnns          1/1     Running   0          8m43s   10.233.68.19    kub-node-2   <none>           <none>
orders-db-5cfc68c4cf-j5nm5      1/1     Running   0          8m43s   10.233.67.20    kub-node-1   <none>           <none>
payment-54f55b96b9-fdngt        1/1     Running   0          8m43s   10.233.67.21    kub-node-1   <none>           <none>
queue-master-6fff667867-p2kgs   1/1     Running   0          8m43s   10.233.68.20    kub-node-2   <none>           <none>
rabbitmq-bdfd84d55-b2zd9        1/1     Running   0          8m43s   10.233.67.22    kub-node-1   <none>           <none>
shipping-78794fdb4f-462st       1/1     Running   0          8m42s   10.233.68.21    kub-node-2   <none>           <none>
user-77cff48476-dbthl           1/1     Running   0          8m42s   10.233.68.22    kub-node-2   <none>           <none>
user-db-99685d75b-49fdj         1/1     Running   0          8m42s   10.233.67.23    kub-node-1   <none>           <none>
```

serveradmin.ru

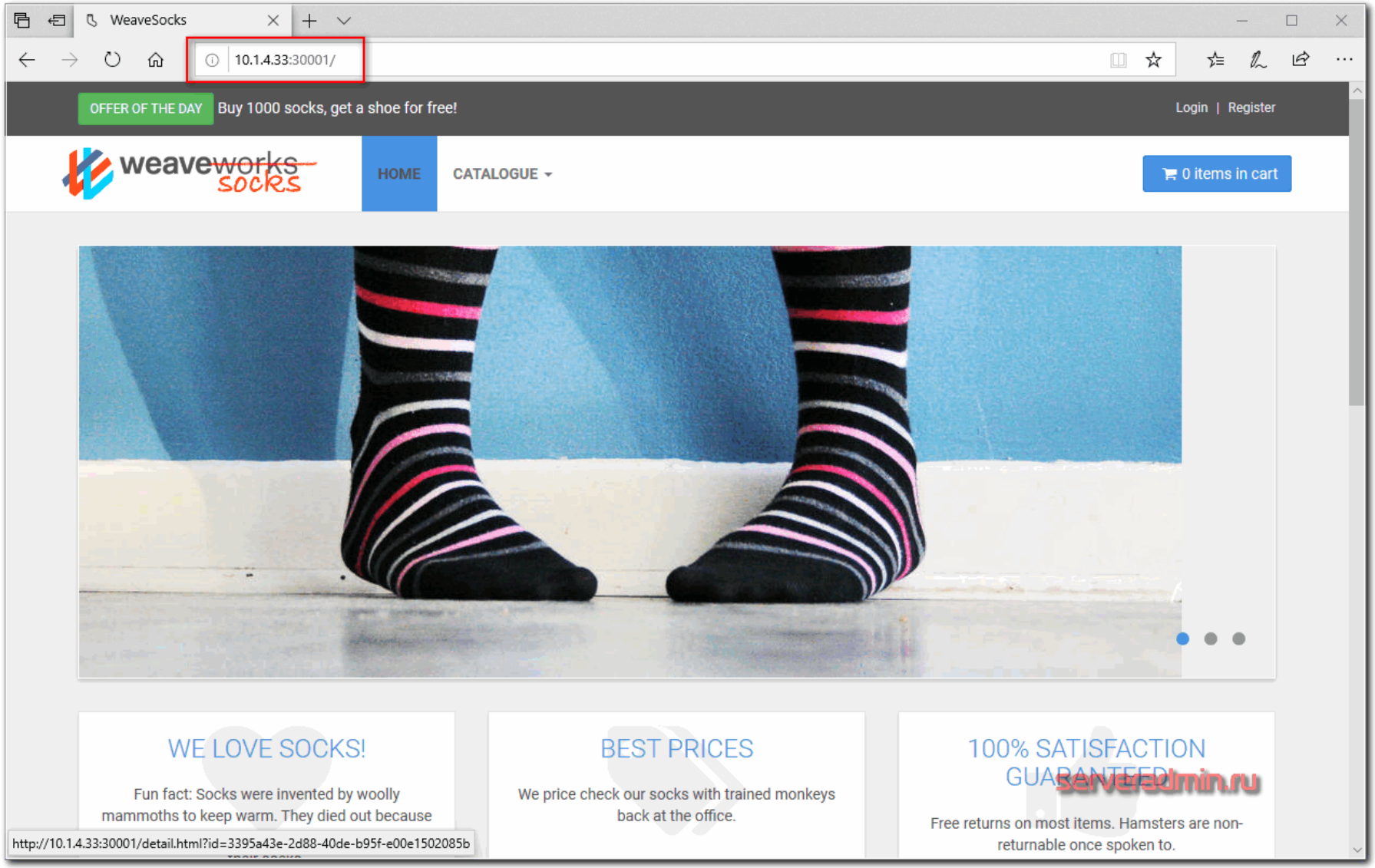
Этот pod запущен на kub-node-2. Посмотрим ее ip.

```
# kubectl get node -o wide
```

```
[root@kub-master-1 ~]# kubectl get node -o wide
NAME            STATUS   ROLES     AGE     VERSION   INTERNAL-IP   EXTERNAL-IP   OS-IMAGE                KERNEL-VERSION                CONTAINER-RUNTIME
kub-ingress-1   Ready    ingress   7d15h   v1.15.3   10.1.4.39     <none>        CentOS Linux 7 (Core)   3.10.0-1062.1.1.el7.x86_64   docker://18.9.7
kub-master-1    Ready    master    7d15h   v1.15.3   10.1.4.36     <none>        CentOS Linux 7 (Core)   3.10.0-1062.1.1.el7.x86_64   docker://18.9.7
kub-master-2    Ready    master    7d15h   v1.15.3   10.1.4.37     <none>        CentOS Linux 7 (Core)   3.10.0-1062.1.1.el7.x86_64   docker://18.9.7
kub-master-3    Ready    master    7d15h   v1.15.3   10.1.4.38     <none>        CentOS Linux 7 (Core)   3.10.0-1062.1.1.el7.x86_64   docker://18.9.7
kub-node-1      Ready    node      7d15h   v1.15.3   10.1.4.32     <none>        CentOS Linux 7 (Core)   3.10.0-1062.1.1.el7.x86_64   docker://18.9.7
kub-node-2      Ready    node      7d15h   v1.15.3   10.1.4.33     <none>        CentOS Linux 7 (Core)   3.10.0-1062.1.1.el7.x86_64   docker://18.9.7
[root@kub-master-1 ~]#
```

serveradmin.ru

Ее ip адрес — 10.1.4.33. Значит для проверки магазина надо идти по урлу http://10.1.4.33:30001/


WeaveSocks
10.1.4.33:30001/
OFFER OF THE DAY
Buy 1000 socks, get a shoe for free!
Login | Register
weaveworks socks
HOME
CATALOGUE
0 items in cart
WE LOVE SOCKS!
Fun fact: Socks were invented by woolly mammoths to keep warm. They died out because
BEST PRICES
We price check our socks with trained monkeys back at the office.
100% SATISFACTION GUARANTEED
Free returns on most items. Hamsters are non-returnable once spoken to.
http://10.1.4.33:30001/detail.html?id=3395a43e-2d88-40de-b95f-e00e1502085b

Вот он, наш магазин. Для удобства, можем сами доделать доступ через ingress по доменному имени. Настраиваем конфиг ingress.

```
---
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-sock
  namespace: sock-shop
spec:
  rules:
  - host: sock-shop.cluster.local
    http:
      paths:
      - backend:
          serviceName: front-end
          servicePort: 80
```

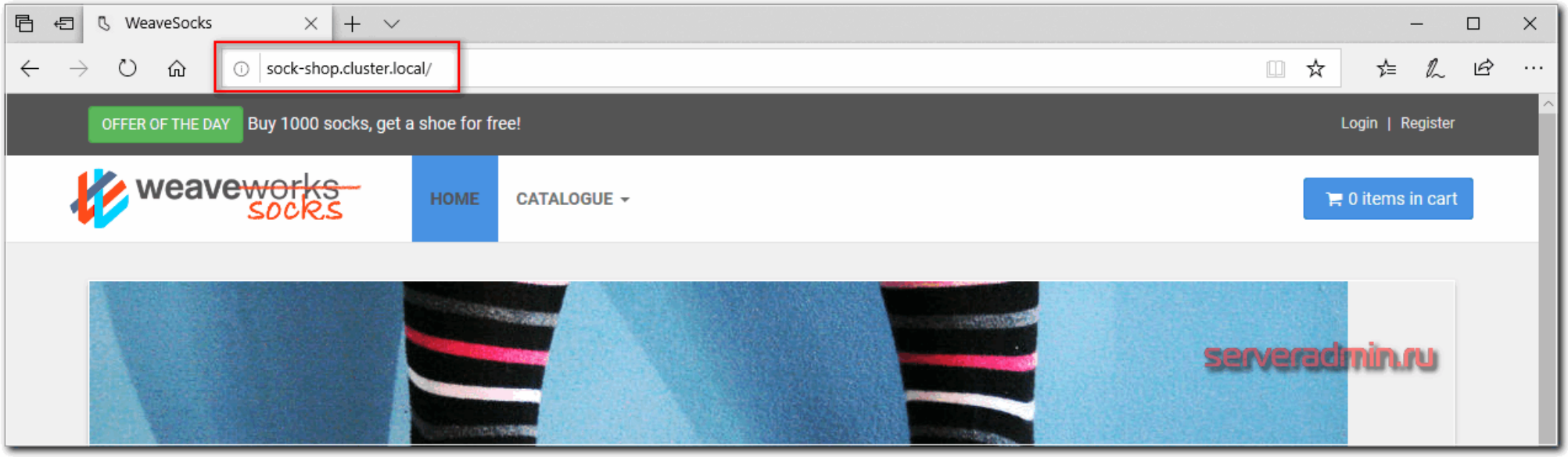
Не забывайте указывать нужный namespace и правильное имя сервиса, для которого настраиваем ingress. Применяем конфиг.

```
# kubectl apply -f ingress-sock.yaml
```

Смотрим, что получилось.

```
# kubectl get ingress -n sock-shop -o wide
NAME           HOSTS                    ADDRESS     PORTS   AGE
ingress-sock   sock-shop.cluster.local   10.1.4.39   80      29s
```

Редактируем файл hosts и идем в браузер проверять.



Поздравляю, ваш кластер работает, а вы теперь администратор кластера Kubernetes и инженер yaml файлов :) У вас теперь будет большая дружба с лапшеподобным синтаксисом. Идите к руководству и требуйе прибавки к зарплате минимум на 30%.

Заключение

> Не понравилась статья и хочешь научить меня администрировать? Пожалуйста, я люблю учиться. Комментарии в твоем распоряжении. Расскажи, как сделать правильно!

Я рассмотрел основные абстракции kubernetes, которые нужны для того, чтобы на нем хоть что-то запустить и начать работать. Кластер в таком виде уже способен принимать запросы из вне. Для полноты картины не хватает одного объемного раздела — фаловые хранилища. Эта отдельная большая тема, поэтому я решил ее не затрагивать здесь, а вынести в отдельную статью, которая будет следующей в этом цикле. В таком виде, как описано здесь,

используются локальные диски нод кластера, где запущены контейнеры docker.

Если вам подходит такой формат работы — пользуйтесь. По большому счету, он вполне жизнеспособен, если у вас все полезные данные, к примеру, хранятся в самих контейнерах в реджистри и в базе данных, которая работает не в кластере, а медиаконтент на внешних CDN. В таком случае kubernetes будет работать как масштабируемый вычислительный кластер.

Напоминаю, что подробно, с примерами и практическими заданиями изучить кластер kubernetes можно на обучении Слёрм, которое я прошел лично и могу рекомендовать, как хороший и эффективный курс.

Онлайн курс по Linux

Если у вас есть желание научиться строить и поддерживать высокодоступные и надежные системы, рекомендую познакомиться с **онлайн-курсом «Администратор Linux»** в OTUS. Курс не для новичков, для поступления нужны базовые знания по сетям и установке Linux на виртуалку. Обучение длится 5 месяцев, после чего успешные выпускники курса смогут пройти собеседования у партнеров. Что даст вам этот курс:

- Знание архитектуры Linux.
- Освоение современных методов и инструментов анализа и обработки данных.
- Умение подбирать конфигурацию под необходимые задачи, управлять процессами и обеспечивать безопасность системы.
- Владение основными рабочими инструментами системного администратора.
- Понимание особенностей развертывания, настройки и обслуживания сетей, построенных на базе Linux.
- Способность быстро решать возникающие проблемы и обеспечивать стабильную и бесперебойную работу системы.

Проверьте себя на вступительном тесте и смотрите подробнее программу по .

Помогла статья? Есть возможность отблагодарить автора