МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
"МИРЭА - Российский технологический университет"

РТУ МИРЭА

Институт информационных технологий (ИТ)
Кафедра инструментального и прикладного программного обеспечения

ОТЧЕТ
ПО ПРАКТИЧЕСКОЙ РАБОТЕ №6

по дисциплине

«Технологии виртуализации клиент-серверных приложений»

Выполнил студент группы ИКБО-01-22 Прокопчук Р.О.

Принял преподаватель кафедры ИиППО Волков М.Ю.

Практические работы выполнены «__»_____2025 г.

«Зачтено» «__»_____2025 г.

Москва
2025

Теоретическое введение

Прежде чем погружаться в подробности развертывания данного приложения в Kubernetes, следует поговорить о том, как управлять конфигурацией. В Kubernetes все представлено в декларативном виде. Это значит, что для определения всех аспектов своего приложения вы сначала описываете, каким должно быть его состояние в кластере (обычно в формате YAML или JSON).

Декларативный подход куда более предпочтителен по сравнению с императивным, в котором состояние кластера представляет собой совокупность внесенных в него изменений.

В случае императивной конфигурации очень сложно понять и воспроизвести состояние, в котором находится кластер. Это существенно затрудняет диагностику и исправление проблем, возникающих в приложении. Предпочтительный формат для объявления состояния приложения — YAML, хотя Kubernetes поддерживает и JSON. Дело в том, что YAML выглядит несколько более компактно и лучше подходит для редактирования вручную.

Однако стоит отметить: этот формат чувствителен к отступам, из-за чего многие ошибки в конфигурационных файлах связаны с некорректными пробельными символами. Если что-то идет не так, то имеет смысл проверить отступы. Поскольку декларативное состояние, хранящееся в этих YAMLфайлах, служит источником истины, из которого ваше приложение черпает информацию, правильная работа с состоянием — залог успеха.

В ходе его редактирования вы должны иметь возможность управлять изменениями, проверять их корректность, проводить аудит их авторов и откатывать изменения в случае неполадок. К счастью, в контексте разработки ПО для всего этого уже есть подходящие инструменты. В частности, практические рекомендации, относящиеся к управлению версиями и аудиту изменений кода, можно смело использовать в работе с декларативным состоянием приложения.

Процесс сборки образов как таковой уязвим к «атакам на поставщиков». В ходе подобных атак злоумышленник внедряет код или двоичные файлы в одну из зависимостей, которая хранится в доверенном источнике и участвует в сборке вашего приложения. Поскольку это создает слишком высокий риск, в сборке должны участвовать только хорошо известные и доверенные провайдеры образов. В качестве альтернативы все образы можно собирать с нуля; для некоторых языков (например, Go) это не составляет труда, поскольку они могут создавать статические исполняемые файлы, но в интерпретируемых языках, таких как Python, JavaScript и Ruby, это может быть большой проблемой.

Некоторые рекомендации касаются выбора имен для образов. Теоретически тег с версией образа контейнера в реестре можно изменить, но вы никогда не должны этого делать. Хороший пример системы именования — сочетание семантической версии и SHA-хеша фиксации кода, из которой собирается образ (например, v1.0.1-bfeda01f). Если версию не указать, то по умолчанию используется значение latest. Оно может быть удобно в процессе разработки, но в промышленных условиях данного значения лучше избегать, так как оно явно изменяется при создании каждого нового образа.

Рекомендуется следить за тем, чтобы содержимое кластера в точности соответствовало содержимому репозитория. Для этого лучше всего использовать методику GitOps и брать код для промышленной среды только из определенной ветки системы контроля версий. Данный процесс можно автоматизировать с помощью непрерывной интеграции (Continuous Integration, CI) и непрерывной доставки (Continuous Delivery, CD). Это позволяет гарантировать соответствие между репозиторием и промышленной системой. Для простого приложения полноценный процесс CI/CD может показаться избыточным, но автоматизация как таковая, даже если не брать во внимание повышение надежности, которое она обеспечивает, обычно стоит затраченных усилий. Внедрение CI/CD в уже существующий проект с императивным развертыванием — чрезвычайно сложная задача.

Kubernetes — эффективная система, которая может показаться сложной. Однако процесс развертывания обычного приложения легко упростить, если следовать общепринятым рекомендациям.

— Большинство сервисов нужно развертывать в виде ресурса Deployment. Объекты Deployment создают идентичные реплики для масштабирования и обеспечения избыточности.

— Для доступа к объектам Deployment можно использовать объект Service, который, в сущности, является балансировщиком нагрузки. Service может быть доступен как изнутри (по умолчанию), так и снаружи. Если вы хотите, чтобы к вашему HTTP-приложению можно было обращаться, то используйте контроллер Ingress для добавления таких возможностей, как маршрутизация запросов и SSL.

— Рано или поздно ваше приложение нужно будет параметризировать, чтобы сделать его конфигурацию более пригодной к использованию в разных средах. Для этого лучше всего подходят диспетчеры пакетов, такие как Helm (helm.sh).

Постановка задачи

В данной практической будет рассмотрена конфигурация развертывания систем.

Для организации компонентов приложения обычно стоит использовать структуру папок файлов системы. Сервис приложения обычно хранится в отдельном каталоге, а его компоненты — в подкаталогах.

В этом примере мы структурируем файлы таким образом:

journal/

frontend/

redis/

fileserver/

Внутри каждого каталога находятся конкретные YAML-файлы, необходимые для определения сервиса. Как вы позже сами увидите, по мере развертывания нашего приложения в разных регионах или кластерах эта структура каталогов будет все более усложняться.

В качестве этой практике будет выступать приложение для Node.js, написанное на языке TypeScript. Его код (https://github.com/brendandburns/kbpsample) На порте 8080 работает HTTP-сервис, который обслуживает запросы к /api/* и использует сервер Redis для добавления, удаления и вывода актуальных записей журнала. Вы можете собрать это приложение в виде образа контейнера, используя включенный в его код файл Dockerfile, и загрузить его в собственный репозиторий образов. Затем вы сможете подставить его имя в YAML-файлы, приведенные ниже.

Наше клиентское приложение является stateless (не хранит свое состояние), делегируя данную функцию серверу Redis. Благодаря этому его можно реплицировать произвольным образом без воздействия на трафик. И хотя наш пример вряд ли будет испытывать серьезные нагрузки, все же неплохо использовать как минимум две реплики (копии): это позволяет справляться с неожиданными сбоями и выкатывать новые версии без простоя.

В Kubernetes есть объект ReplicaSet, отвечающий за репликацию контейнеризованных приложений, но его лучше не использовать напрямую. Для наших задач подойдет объект Deployment, который сочетает в себе возможности объекта ReplicaSet, систему управления версиями и поддержку поэтапного развертывания обновлений. Объект Deployment позволяет применять встроенные в Kubernetes механизмы для перехода от одной версии к другой.

Так, для идентификации экземпляров ReplicaSet, объекта Deployment и создаваемых им pod используются метки (labels). Мы добавили метку layer: frontend для всех этих объектов, благодаря чему они теперь находятся в общем слое (layer) и их можно просматривать вместе с помощью одного запроса. Обратите внимание и на то, что для контейнеров в ресурсе Deployment установлены запросы ресурсов Request и Limit с одинаковыми значениями. Request гарантирует выделение определенного объема ресурсов на сервере, на котором запущено приложение. Limit — максимальное потребление ресурсов, разрешенное к использованию контейнером. в. Когда Request и Limit равны, ваше приложение не тратит слишком много процессорного времени и не потребляет лишние ресурсы при бездействии.

Для настройки внешнего доступа для HTTP-трафика мы воспользуемся двумя ресурсами. Первый — это Service, который распределяет (балансирует) трафик, поступающий по TCP или UDP. В нашем примере мы задействуем протокол TCP. Второй ресурс — объект Ingress, обеспечивающий балансировку нагрузки с гибкой маршрутизацией запросов в зависимости от доменных имен и HTTP-путей.

Прежде чем определять ресурс Ingress, следует создать Kubernetes Service, на который он будет указывать. А чтобы связать этот Service с pod, созданными в предыдущем разделе, мы воспользуемся метками. Определение Service выглядит намного проще, чем ресурс Deployment.

Теперь можно определить ресурс Ingress. Он, в отличие от Service, требует наличия в кластере контейнера с подходящим контроллером.

Контроллеры бывают разные: одни из них предоставляются облачными провайдерами, а другие основываются на серверах с открытым исходным кодом. Если вы выбрали открытую реализацию Ingress, то для ее установки и обслуживания лучше использовать диспетчер пакетов Helm (helm.sh). Популярностью пользуются такие реализации, как nginx и haproxy.

Конфигурация позволяет быстро (и даже динамически) активизировать и деактивизировать возможности в зависимости от потребностей пользователей или программных сбоев. В Kubernetes такого рода конфигурация представлена в ConfigMap, в формате «ключ — значение». Эта информация предоставляется подам с помощью файлов или переменных среды. Для того, чтобы создать ConfigMap с переменной journalEntries нужно выполнить команду. Затем вы должны предоставить конфигурационную информацию в виде переменной среды в самом приложении.

В любом реальном проекте соединение между сервисами должно быть защищенным. Для аутентификации в сервере Redis используется обычный пароль, который будет храниться в объекте Secret. Для того чтобы создать этот объект Secret воспользуемся командой.

Сохранив пароль к Redis в виде объекта Secret, вы должны привязать его к приложению, которое развертывается в Kubernetes. Для этого можно использовать ресурс Volume (том). В случае с секретными данными том создается в виде файловой системы tmpfs в оперативной памяти, и затем подключается к контейнеру.

Чтобы добавить секретный том в объект Deployment, вам нужно указать в YAML-файле последнего два дополнительных раздела. Первый раздел, volumes, добавляет том в pod.

Затем том нужно подключить к определенному контейнеру. Для этого в описании контейнера следует указать поле volumeMounts.

Благодаря этому том становится доступным для клиентского кода в каталоге redis-passwd.

Для развертывания сервиса Redis необходим StatefulSet. Это дополнение к ReplicaSet, которое предоставляет более строгие гарантии, такие как согласованные имена и определенный порядок увеличения и уменьшения количества pod (scale-up, scale-down).

Чтобы запросить постоянный том для нашего сервиса Redis, используется PersistentVolumeClaim. Это своеобразный запрос ресурсов. Сервис объявляет, что ему нужно хранилище размером 50 Гбайт, а кластер определяет, как выделить подходящий постоянный том. Данный механизм нужен по двум причинам. Во-первых, он позволяет создать ресурс StatefulSet, который можно переносить между разными облаками и размещать локально, не заботясь о конкретных физических дисках. Во-вторых, несмотря на то, что том типа PersistentVolume можно подключить лишь к одному pod, запрос тома позволяет написать шаблон, доступный для реплицирования, но при этом каждому pod будет назначен отдельный постоянный том.

Когда мы добавляем в объект StatefulSet новый неуправляемый (headless) сервис, для него автоматически создается DNS-запись redis-0.redis; это IP-адрес первой реплики. Вы можете воспользоваться этим для написания сценария, пригодного для запуска во всех контейнерах.

Этот сценарий можно оформить в виде ConfigMap.

Затем объект ConfigMap нужно добавить в StatefulSet и использовать его как команду для управления контейнером. Добавим также пароль для аутентификации, который создали ранее.

Итак, мы развернули stateful-сервис Redis; теперь его нужно сделать доступным для нашего клиентского приложения. Для этого создадим два разных Service Kubernetes. Первый будет читать данные из Redis. Поскольку они реплицируются между всеми тремя участниками StatefulSet, для нас несущественно, к какому из них будут направляться наши запросы на чтение. Следовательно, для этой задачи подойдет простой Service.

Выполнение записи потребует обращения к ведущей реплике Redis (под номером 0). Создайте для этого неуправляемый (headless) Service. У него

нет IP-адреса внутри кластера; вместо этого он задает отдельную DNS-запись для каждого pod в StatefulSet. То есть мы можем обратиться к нашей ведущей реплике по доменному имени redis-0.redis.

Таким образом, если нам нужно подключиться к Redis для сохранения каких-либо данных или выполнения транзакции с чтением/записью, то мы можем собрать отдельный клиент, который будет подключаться к серверу redis0.redis-write Заключительный компонент нашего приложения — сервер статических файлов, который отвечает за раздачу HTML-, CSS-, JavaScript-файлов и изображений. Ingress позволяет очень легко организовать такую архитектуру в стиле микросервисов. Как и в случае с клиентским приложением, можно реплицируемый сервер NGINX с помощью ресурса Deployment. Соберем статические образы в контейнер NGINX и развернем их в каждой реплике. Ресурс Deployment будет выглядеть следующим образом.

Теперь, запустив реплицируемый статический веб-сервер, вы можете аналогичным образом создать ресурс Service, который будет играть роль балансировщика нагрузки.

Итак, у вас есть Service для сервера статических файлов. Добавим в ресурс Ingress новый путь. Необходимо отметить, что путь / должен идти после /api, иначе запросы API станут направляться серверу статических файлов.

В результате выполнения работы вам необходимо отобразить в отчете все этапы конфигурации программной системы и показать работоспособность системы, доступ к которой настроен с помощью Ingress, с помощью браузера.

Ход работы

Разработанные yaml файлы представлены на рисунках 1-8.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: journal-frontend
  labels:
    app: journal
    layer: frontend
spec:
  replicas: 2
  selector:
    matchLabels:
      app: journal
      layer: frontend
  template:
    metadata:
      labels:
        app: journal
        layer: frontend
    spec:
      containers:
        - name: frontend
          image: remsely/kbp-sample:1.0.0
          ports:
            - containerPort: 8080
          resources:
            requests:
              memory: "128Mi"
              cpu: "100m"
            limits:
              memory: "128Mi"
              cpu: "100m"
          env:
            - name: JOURNAL_ENTRIES
              valueFrom:
                configMapKeyRef:
                  name: journal-config
                  key: journalEntries
            - name: REDIS_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: redis-passwd
                  key: passwd
          volumeMounts:
            - name: redis-passwd
              mountPath: /etc/redis-passwd
              readOnly: true
      volumes:
        - name: redis-passwd
          secret:
            secretName: redis-passwd
```

Рисунок 1 – Файл frontend/deployment.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: journal-frontend
  labels:
    app: journal
    layer: frontend
spec:
  selector:
    app: journal
    layer: frontend
  ports:
    - port: 8080
      targetPort: 8080
  type: ClusterIP
```

Рисунок 2 – Файл frontend/service.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: fileserver
  labels:
    app: fileserver
spec:
  replicas: 2
  selector:
    matchLabels:
      app: fileserver
  template:
    metadata:
      labels:
        app: fileserver
    spec:
      containers:
        - name: nginx
          image: nginx:1.21
          ports:
            - containerPort: 80
          resources:
            requests:
              memory: "64Mi"
              cpu: "50m"
            limits:
              memory: "64Mi"
              cpu: "50m"
```

Рисунок 3 – Файл fileserver/deployment.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: fileserver
  labels:
    app: fileserver
spec:
  selector:
    app: fileserver
  ports:
    - port: 80
      targetPort: 80
  type: ClusterIP
```

Рисунок 4 – Файл fileserver/service.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: redis
  labels:
    app: redis
spec:
  selector:
    app: redis
  ports:
    - port: 6379
      targetPort: 6379
  type: ClusterIP
```

Рисунок 5 – Файл redis/service-read.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: redis-write
  labels:
    app: redis
spec:
  selector:
    app: redis
  ports:
    - port: 6379
      targetPort: 6379
  clusterIP: None
```

Рисунок 6 – Файл redis/service-write.yaml

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: redis
spec:
  serviceName: redis
  replicas: 3
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
        - name: redis
          image: redis:6.2
          ports:
            - containerPort: 6379
          command:
            - sh
            - -c
            - |
              if [ "$(hostname)" = "redis-0" ]; then
                redis-server --requirepass $(cat /etc/redis-passwd/passwd)
              else
                redis-server --requirepass $(cat /etc/redis-passwd/passwd) --slaveof redis-0.redis 6379 --masterauth \
                  $(cat /etc/redis-passwd/passwd)
              fi
          volumeMounts:
            - name: data
              mountPath: /data
            - name: redis-passwd
              mountPath: /etc/redis-passwd
              readOnly: true
      volumes:
        - name: redis-passwd
          secret:
            secretName: redis-passwd
  volumeClaimTemplates:
    - metadata:
        name: data
      spec:
        accessModes: [ "ReadWriteOnce" ]
        resources:
          requests:
            storage: 1Gi
```

Рисунок 7 – Файл redis/statefulset.yaml

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: journal-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /$2
spec:
  ingressClassName: nginx
  rules:
    - host: journal.local
      http:
        paths:
          - path: /api(/|$)(.*)
            pathType: ImplementationSpecific
            backend:
              service:
                name: journal-frontend
                port:
                  number: 8080
          - path: /
            pathType: Prefix
            backend:
              service:
                name: fileserver
                port:
                  number: 80
```

Рисунок 8 – Файл ingress.yaml

Выполнение команд для развертывания систем представлено на рисунке 9. Команды для просмотра статуса развертываемых систем представлены на рисунке 10.

```
PS C:\Users\Remsely\dev\mirea\csavt> minikube start
* minikube v1.37.0 на Microsoft Windows 11 Iot Enterprise Ltsc 2024 10.0.26100.4061 Build 26100.4061
* Используется драйвер hyperv на основе существующего профиля
* Starting "minikube" primary control-plane node in "minikube" cluster
* Перезагружается существующий hyperv VM для "minikube" ...
! Failing to connect to https://registry.k8s.io/ from inside the minikube VM
* To pull new external images, you may need to configure a proxy: https://minikube.sigs.k8s.io/docs/reference/networking/
* Подготавливается Kubernetes v1.34.0 на Docker 28.4.0 ...
* Configuring bridge CNI (Container Networking Interface) ...
* Компоненты Kubernetes проверяются ...
  - Используется образ gcr.io/k8s-minikube/storage-provisioner:v5
* Включенные дополнения: storage-provisioner, default-storageclass

! C:\Program Files\Docker\Docker\resources\bin\kubectl.exe is version 1.31.4, which may have incompatibilities with Kuber
  - Want kubectl v1.34.0? Try 'minikube kubectl -- get pods -A'
* Готово! kubectl настроен для использования кластера "minikube" и "default" пространства имён по умолчанию
PS C:\Users\Remsely\dev\mirea\csavt> minikube addons enable ingress
* ingress is an addon maintained by Kubernetes. For any concerns contact minikube on GitHub.
You can view the list of minikube maintainers at: https://github.com/kubernetes/minikube/blob/master/OWNERS
  - Используется образ registry.k8s.io/ingress-nginx/kube-webhook-certgen:v1.6.2
  - Используется образ registry.k8s.io/ingress-nginx/kube-webhook-certgen:v1.6.2
  - Используется образ registry.k8s.io/ingress-nginx/controller:v1.13.2
* Verifying ingress addon...
* The 'ingress' addon is enabled
PS C:\Users\Remsely\dev\mirea\csavt> kubectl create configmap journal-config --from-literal=journalEntries=10
configmap/journal-config created
PS C:\Users\Remsely\dev\mirea\csavt> kubectl create secret generic redis-passwd --from-literal=passwd=mySecurePassword123
secret/redis-passwd created
PS C:\Users\Remsely\dev\mirea\csavt> cd .\practice-6-kubernetes-deployment\
PS C:\Users\Remsely\dev\mirea\csavt\practice-6-kubernetes-deployment> kubectl apply -f journal/redis/statefulset.yaml
statefulset.apps/redis created
PS C:\Users\Remsely\dev\mirea\csavt\practice-6-kubernetes-deployment> kubectl apply -f journal/redis/service-read.yaml
service/redis created
PS C:\Users\Remsely\dev\mirea\csavt\practice-6-kubernetes-deployment> kubectl apply -f journal/redis/service-write.yaml
Warning: spec.SessionAffinity is ignored for headless services
service/redis-write created
PS C:\Users\Remsely\dev\mirea\csavt\practice-6-kubernetes-deployment> kubectl apply -f journal/frontend/deployment.yaml
deployment.apps/journal-frontend created
PS C:\Users\Remsely\dev\mirea\csavt\practice-6-kubernetes-deployment> kubectl apply -f journal/frontend/service.yaml
service/journal-frontend created
PS C:\Users\Remsely\dev\mirea\csavt\practice-6-kubernetes-deployment> kubectl apply -f journal/fileserver/deployment.yaml
deployment.apps/fileserver created
PS C:\Users\Remsely\dev\mirea\csavt\practice-6-kubernetes-deployment> kubectl apply -f journal/fileserver/service.yaml
service/fileserver created
PS C:\Users\Remsely\dev\mirea\csavt\practice-6-kubernetes-deployment> kubectl apply -f journal/ingress.yaml
ingress.networking.k8s.io/journal-ingress unchanged
```

Рисунок 9 – Команды для развёртывания систем

```
PS C:\Users\Remsely\dev\mirea\csavt\practice-6-kubernetes-deployment> kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
fileserver-5b648c79f4-s6nkh         1/1     Running   0          37s
fileserver-5b648c79f4-w75pt         1/1     Running   0          37s
journal-frontend-5f9c8559cb-4wm7n   1/1     Running   0          50s
journal-frontend-5f9c8559cb-hs9zb   1/1     Running   0          50s
redis-0                             1/1     Running   0          103s
redis-1                             1/1     Running   0          102s
redis-2                             1/1     Running   0          101s
PS C:\Users\Remsely\dev\mirea\csavt\practice-6-kubernetes-deployment> kubectl get services
NAME             TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
fileserver       ClusterIP   10.99.16.207     <none>        80/TCP     39s
journal-frontend ClusterIP   10.109.185.125   <none>        8080/TCP   51s
kubernetes       ClusterIP   10.96.0.1        <none>        443/TCP    6m51s
redis            ClusterIP   10.96.12.189     <none>        6379/TCP   105s
redis-write      ClusterIP   None             <none>        6379/TCP   99s
PS C:\Users\Remsely\dev\mirea\csavt\practice-6-kubernetes-deployment> kubectl get ingress
NAME              CLASS    HOSTS           ADDRESS         PORTS   AGE
frontend-ingress  nginx    *               172.31.46.86    80      45h
journal-ingress   <none>   journal.local   172.31.46.86    80      18m
PS C:\Users\Remsely\dev\mirea\csavt\practice-6-kubernetes-deployment> minikube ip
172.31.46.86
```

Рисунок 10 – Команды для развёртывания систем

Состояние системы в DashBard представлено на рисуках 11-12.

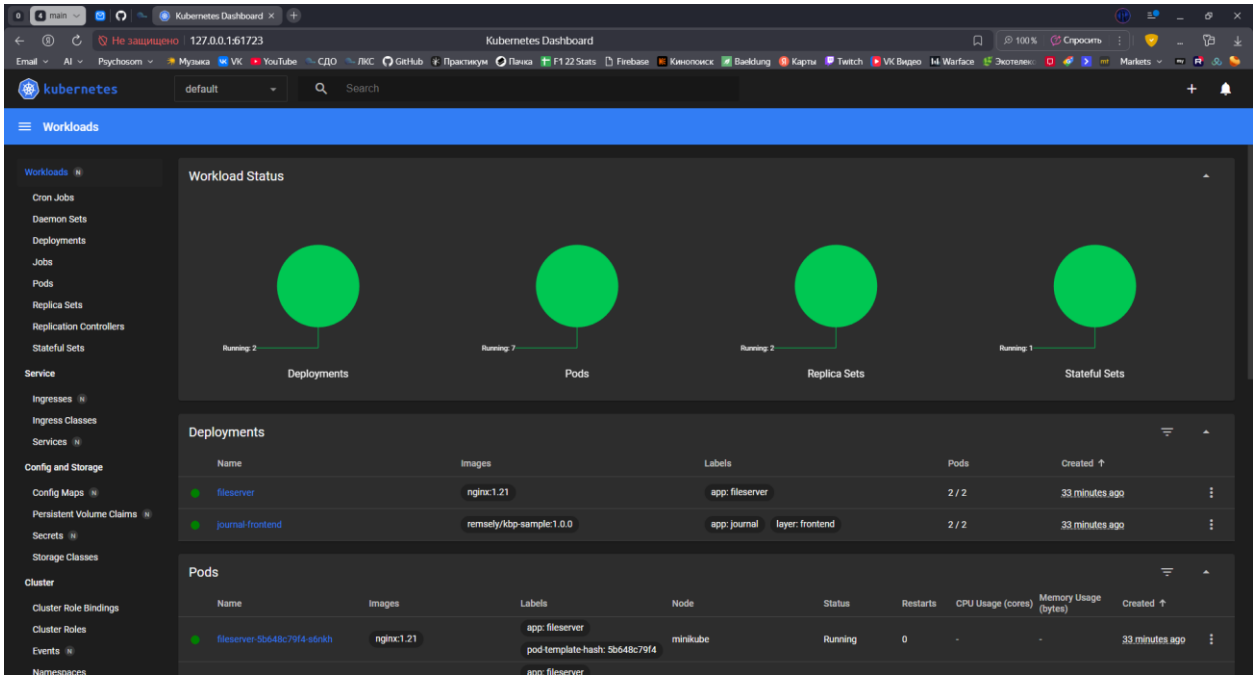

Рисунок 11 – Состояние Workload в Dashboard

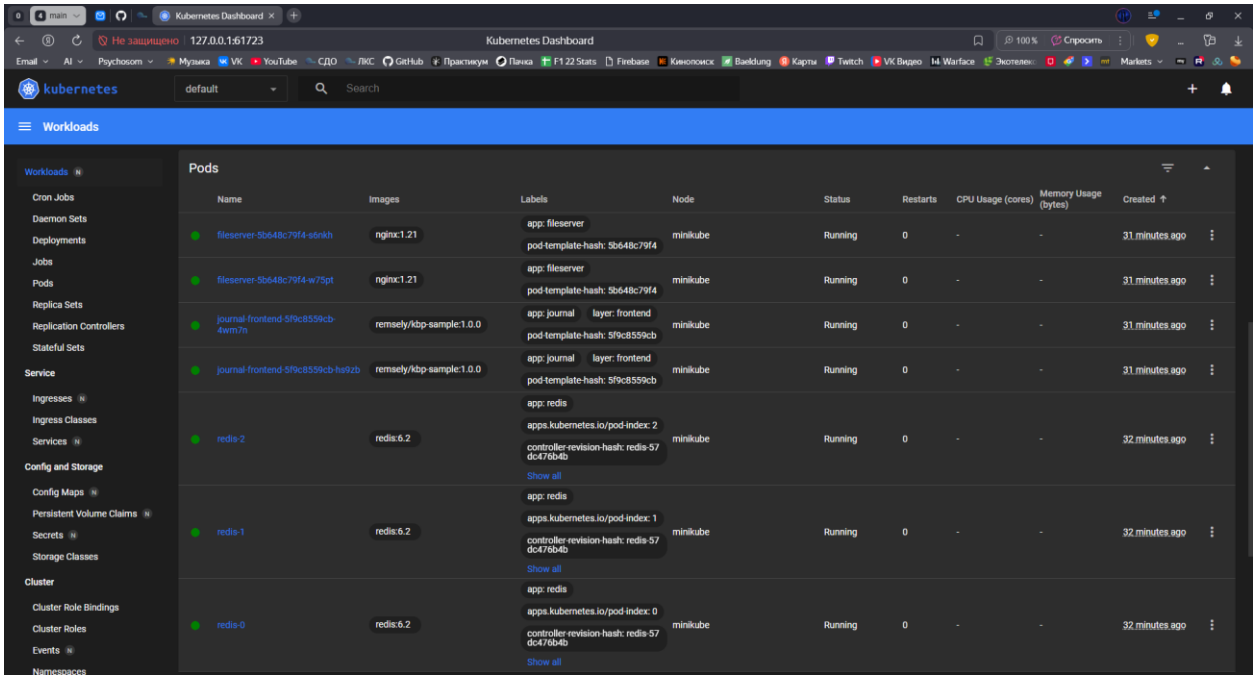

Рисунок 12 – Состояние под в Kubernetes Dashboard

Вывод

В результате выполнения данной практической было выполнено развертывание примера в кластере Kubernetes.

Ответы на вопросы к практической работе

1. Для чего нужен ресурс Deployment?

Deployment управляет репликами приложения, обеспечивает их масштабирование, обновление без простоя и откат к предыдущим версиям.

2. Почему не стоит хранить пароли в ConfigMap?

ConfigMap хранит данные в незашифрованном виде. Для паролей используют Secret, который предоставляет дополнительную защиту (хранение в памяти, шифрование).

3. Что необходимо для настройки внешнего доступа для HTTP-трафика? Назовите шаги.

Создать Service для балансировки внутри кластера, создать Ingress для маршрутизации внешних запросов, установить Ingress Controller.

4. Чем отличается развертывание stateful от развертывания клиентского приложения?

Stateless (Deployment) – реплики идентичны, можно добавлять/удалять в любом порядке.

Stateful (StatefulSet) – каждая реплика уникальна, имеет постоянное имя и собственный диск.

5. Где хранится том с секретными данными?

В оперативной памяти (tmpfs), а не на диске, для повышенной безопасности.

6. Как работает связка PersistentVolume и PersistentVolumeClaim?

PersistentVolumeClaim – это "запрос" на диск с нужными характеристиками. Kubernetes автоматически находит или создает подходящий PersistentVolume и связывает их.

Список источников информации

1. Kubernetes: лучшие практики. — СПб.: Питер, 2021. — 288 с.: ил. — (Серия «Для профессионалов»). K8S для начинающих. Первая часть — Текст: электронный [сайт]. — URL: https://habr.com/ru/post/589415/

2. Kubernetes или с чего начать, чтобы понять что это и зачем он нужен — Текст: электронный [сайт]. — URL: https://habr.com/ru/company/otus/blog/537162/

3. Основы Kubernetes — Текст: электронный [сайт]. — URL: https://habr.com/ru/post/258443/