Лекция 16: Введение в оркестрацию. Управляем 'флотом' наших приложений с помощью Kubernetes

Введение: Проблема масштаба

Здравствуйте! На прошлой лекции мы построили идеальный "заводской конвейер" (CI/CD), который автоматически собирает и тестирует наше приложение, упаковывая его в стандартный Docker-контейнер. Мы даже научились автоматически доставлять этот контейнер на сервер.

Но до сих пор мы рассматривали наш "дом" как единое целое, стоящее на одном участке земли (на одном сервере). Что произойдет, если:

- **Сервер выйдет из строя?** (отключится электричество, сгорит диск) — наше приложение полностью перестанет работать.
- **Нагрузка вырастет?** Один сервер перестанет справляться с тысячами пользователей, и приложение начнет "тормозить".
- **Нам нужно обновить приложение?** Придется остановить старую версию и запустить новую, что приведет к простою (downtime).

Инструменты вроде Docker Compose отлично решают эти задачи на машине одного разработчика, но они не предназначены для управления приложениями в производственной среде, состоящей из множества серверов. Нам нужен "дирижер" для нашего "оркестра" контейнеров. Этот класс инструментов называется **оркестраторами**.

1. Что такое оркестрация?

Оркестрация контейнеров — это автоматизированное управление жизненным циклом большого количества контейнеров в распределенной среде (кластере серверов).

Аналогия:

- **Docker Compose** — это **менеджер одного ресторана**. Он хорошо справляется с персоналом и поставками в рамках одного заведения.
- **Оркестратор (Kubernetes)** — это **центральный офис (штаб-квартира) всей сети ресторанов**. Он не управляет каждым поваром лично, а задает общие правила ("в каждом ресторане должно быть 3 повара", "если один заболел — немедленно найти замену") и следит за их выполнением в масштабах всей страны.

Ключевые задачи оркестратора:

- **Планирование (Scheduling):** Решает, на каком из серверов кластера запустить контейнер.
- **Масштабирование (Scaling):** Увеличивает или уменьшает количество копий приложения.
- **Самовосстановление (Self-healing):** Если контейнер или сервер "умирает", оркестратор автоматически перезапускает его на другом месте.
- **Балансировка нагрузки:** Распределяет трафик между копиями приложения.

- **Управление обновлениями (Rolling Updates):** Обновляет приложение без простоя.

2. Kubernetes: Король оркестрации

В середине 2010-х было несколько конкурирующих оркестраторов, но к сегодняшнему дню **Kubernetes (сокращенно K8s)** стал неоспоримым промышленным стандартом. Разработанный изначально в Google на основе их 15-летнего опыта эксплуатации контейнеров, Kubernetes — это невероятно мощная и гибкая платформа с открытым исходным кодом.

Ключевая идея Kubernetes — декларативный подход. Мы не даем ему пошаговых **императивных** инструкций ("запусти контейнер на сервере А"). Вместо этого мы описываем **декларативно**, в виде YAML-файлов, **желаемое состояние** системы ("Я хочу, чтобы в мире всегда существовало 3 копии моего приложения с версией 1.2"). А Kubernetes сам, как умный термостат, постоянно сверяет желаемое состояние с реальным и предпринимает все необходимые действия, чтобы их синхронизировать.

3. Анатомия Kubernetes: "Строительные блоки"

Мы общаемся с Kubernetes, описывая различные **ресурсы** или **объекты**. Давайте разберем четыре самых главных "строительных блока", из которых собирается любое приложение.

- **Pod ('Стручок'): Атом приложения.**
    - Это самая маленькая развертываемая единица в Kubernetes. Pod — это группа из одного или нескольких тесно связанных контейнеров (в 99% случаев — одного).
    - Kubernetes управляет не контейнерами напрямую, а подами.
    - **Важно:** Поды — **эфемерны**. Они могут быть уничтожены и пересозданы в любой момент с новым IP-адресом. Мы никогда не управляем ими напрямую.
- **Deployment: Контроллер репликации и обновлений.**
    - Это наш главный инструмент. Deployment — это "чертеж", который описывает, как должны "жить" наши поды.
    - Мы говорим ему: "Обеспечь, чтобы всегда было запущено **3 реплики** пода, созданного из образа my-app:1.0".
    - **Что он делает:**
        - **Масштабирование:** Меняем replicas: 3 на replicas: 5, и Deployment сам создаст 2 новых пода.
        - **Самовосстановление:** Если один под "умирает", Deployment тут же создает новый ему на замену.
        - **Обновления:** Меняем my-app:1.0 на my-app:1.1, и Deployment плавно, один за другим, заменит старые поды на новые (**Rolling Update**).
- **Service: Стабильный сетевой адрес.**
    - **Проблема:** Поды непостоянны и имеют меняющиеся IP-адреса. Как другим частям нашего приложения (или внешним пользователям) найти их?

    - **Решение:** Service — это объект, который создает **стабильный, виртуальный IP-адрес** и DNS-имя для группы подов.
    - Он находит нужные поды по **меткам (labels)** и работает как **внутренний балансировщик нагрузки**, распределяя запросы между ними.
- **Ingress: Умный маршрутизатор для внешнего мира.**
    - **Проблема:** Как предоставить доступ к нашим сервисам из интернета? Сервис типа LoadBalancer создает отдельный, дорогой балансировщик для каждого сервиса.
    - **Решение:** Ingress — это набор **правил**, который описывает, как маршрутизировать внешние HTTP/HTTPS запросы к внутренним сервисам на основе хоста или пути. api.com/users -> user-service, api.com/products -> product-service. Он позволяет использовать один внешний балансировщик для множества сервисов.

4. Практика: Жизненный цикл приложения в Kubernetes

Давайте посмотрим, как это работает на практике.

Шаг 1: Описываем желаемое состояние в YAML-файлах.
Мы создаем два файла: deployment.yml и service.yml.

deployment.yml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-app-container
        image: ghcr.io/my-repo/my-app:v1.0.0
        ports:
        - containerPort: 3000
```

service.yml:

```
apiVersion: v1
kind: Service
metadata:
  name: my-app-service
spec:
  type: LoadBalancer
  selector:
```

```
    app: my-app
  ports:
    - port: 80
      targetPort: 3000
```

Шаг 2: Применяем конфигурацию.
С помощью утилиты kubectl мы отправляем эти "приказы" в кластер:

```
kubectl apply -f deployment.yml
kubectl apply -f service.yml
```

Kubernetes читает эти файлы и начинает свою работу: скачивает образы, создает 3 пода, создает сервис и внешний балансировщик. Через несколько минут наше приложение доступно в интернете.

Шаг 3: Демонстрация самовосстановления.
Мы можем вручную удалить один из подов: kubectl delete pod my-app-deployment-xyz. И если мы тут же запросим список подов (kubectl get pods --watch), мы увидим, как старый под переходит в состояние Terminating, а Kubernetes **немедленно** начинает создавать новый, чтобы восстановить желаемое состояние в 3 реплики.

Шаг 4: Демонстрация обновления.
Мы собираем новый образ my-app:v1.1.0. Меняем одну строчку в deployment.yml и снова выполняем kubectl apply -f deployment.yml. Kubernetes запускает процесс **Rolling Update**: он постепенно, один за другим, будет заменять старые поды на новые, гарантируя, что в любой момент времени приложение остается доступным.

Заключение

Сегодня мы сделали большой шаг от управления одним контейнером к управлению целым "флотом". Мы поняли, что **оркестрация** решает фундаментальные проблемы **масштабируемости, отказоустойчивости и обновлений**.

Мы познакомились с **Kubernetes** — промышленным стандартом в этой области, и его декларативным подходом. Мы разобрали его ключевые "строительные блоки" — Pod, Deployment, Service и Ingress.

Запускать и поддерживать свой собственный Kubernetes-кластер — это сложная задача. К счастью, сегодня нам редко приходится это делать. На следующей лекции мы познакомимся с **облачными провайдерами**, которые предоставляют **управляемые Kubernetes-кластеры** и множество других полезных сервисов, снимая с нас бремя сложного администрирования.