Механика запуска и взаимодействия контейнеров

Не забудь включить запись!

План

- От `run` до `Running`:
  - Подготовка к запуску пода
  - Планирование ресурсов
  - Настройка окружения Pod
- Управление группами Podов:
  - ReplicaSet, ReplicationController, Job
  - CronJob, DaemonSet, Deployment

Запуск Pod

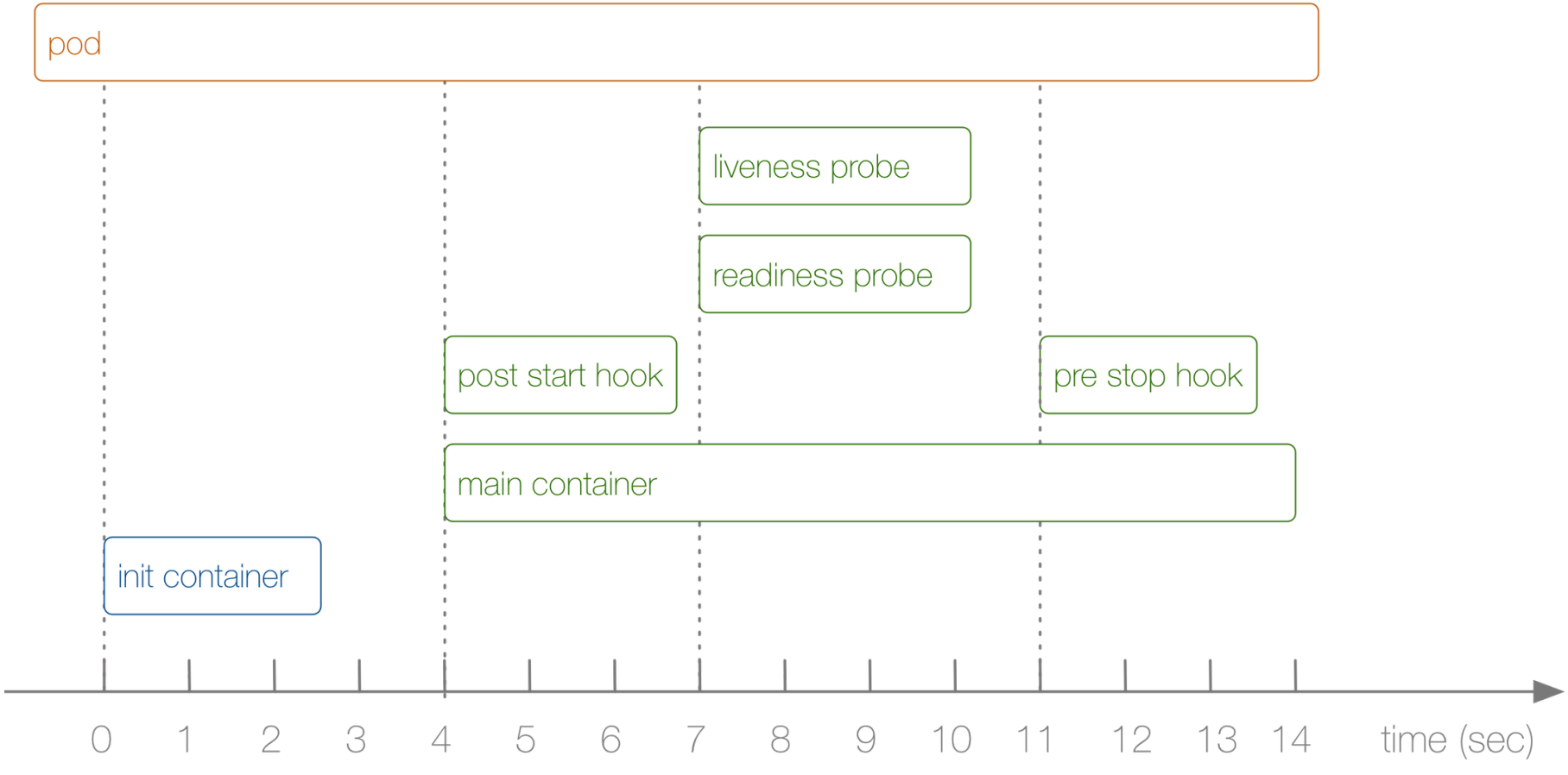
Обычная последовательность при старте Pod:

1. Подготовка и валидация runtime-объекта
2. Идентификация и авторизация запроса
3. Обработка запроса Admission Controllers
4. Сохранение ресурса в *etcd*

Запуск Pod

6. Основной цикл обработки ресурсов (e.g. Deployment Controller)
7. Запуск планировщика
8. Магия `kubelet` 🦄
9. Магия runtime. Подготовка изолированного окружения
10. Магия runtime. Инициализация сети
11. Магия runtime. Запуск контейнеров
12. И снова `kubelet`... Запуск хуков

Запуск Pod



Источник: блог OKD, ссылка в конце презентации

Подготовка API-запроса

- Актуальные схемы в формате OpenAPI можно найти на `kube-apiserver` (`/apis`)
- `kubectl` кэширует информацию о схеме API
  - Кэш лежит тут - `~/.kube/cache/discovery`
  - Кэш придумали не зря. `kubectl -v` с пустым кэшем наглядно это покажет

Тот самый AAA

- После идентификации пользователя на `kube-apiserver` начинается самое интересное
- Запрос проходит через цепочку Authorizers (RBAC, ABAC, Node):
  - Достаточно первого ответа (можно/нельзя)
  - Если ни один авторизатор не ответил внятно, запрос отклоняется
- Далее, вызываются Admission Controllers, которые смотрят на параметры runtime-объекта
  - Можно/нельзя (e.g. `PodSecurityPolicy`)
  - Меняют параметры объекта (e.g. `AlwaysPullImages`)
  - Создают новые объекты (e.g. `NamespaceAutoProvision`)

Планирование ресурсов

1. Допущенные к запуску и окончательно мутировавшие Pods попадают в очередь планировщика
2. Планировщик (`kube-scheduler`) сортирует очередь
3. После сортировки планировщик выбирает ноды, подходящие для запуска Pod и делает оценку "пригодности" каждой ноды
4. В финале, происходит binding - привязка Poda к узлу (и измененный объект Pod сохраняется в etcd)

Что делает Kubelet

1. `kubelet` опрашивает `kube-apiserver` и забирает список подов со своим `NodeName`
2. Полученный список подов сверяется со списком запущенных
3. Далее он удаляет и создает Pods, чтобы достичь соответствия между описанным и текущим состоянием.
4. `NodeAuthorizer` ограничивает права `kubelet` объектами, связанными с его Podами.
5. `kubelet` регистрирует базовые метрики связанные с Pod (например, тайминги запуска)
6. `kubelet` выполняет дополнительные проверки (`AdmissionHandlers`) для полученного Pod

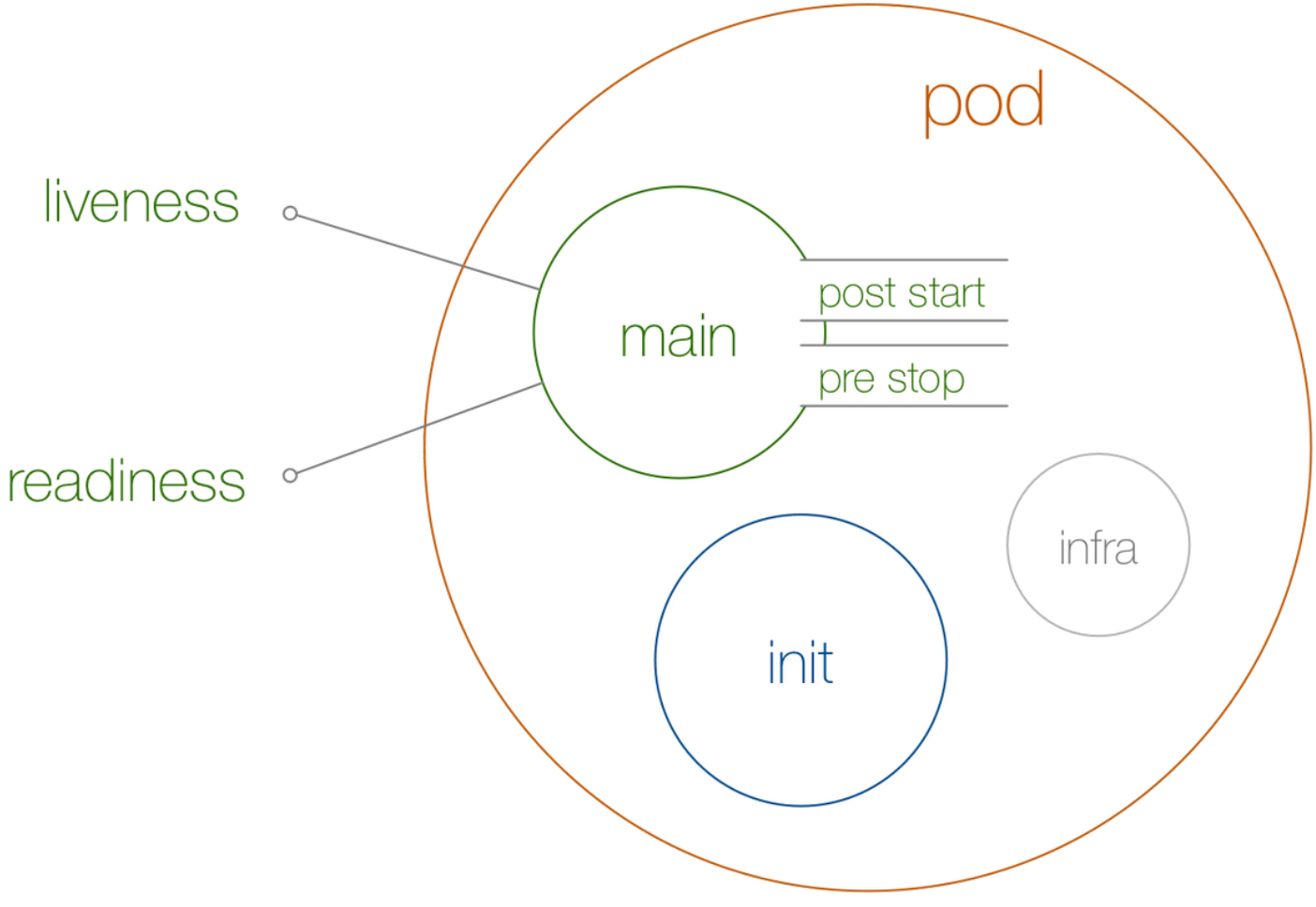
Kubelet | AdmissionHandlers, PodSyncHandlers

1. Проверки Poda на совместимость с текущей версией и конфигурацией Runtime
   - `NoNewPrivileges`, sysctl, AppArmor/SELinux профили
2. Доступные ресурсы
   - Для подов с гарантированными ресурсами всегда "зеленый свет" 🚦 (ресурсы освобождаются потом)
3. Также `kubelet` делает периодические проверки запущенных Podов (начиная с момента запуска)
   - Например, проверка соответствия `ActiveDeadline` для Jobs
   - Поды не прошедшие проверку идут на... eviction

Kubelet | 🦄

❗ Проще говоря, Pods, которые прошли Server Dry-run, AdmissionController-ы и планировщик НЕ ОБЯЗАТЕЛЬНО будут запущены

Структура Pod



Источник: блог OKD, ссылка в конце презентации

PodStatus | Phase

- ⏲ Pending - объект был создан и может быть даже, назначена Node для него. Если Pod не прошел AdmissionHandlers, он останется в этом состоянии
- 🏃 Running - все контейнеры Poda запущены и post-start hook отработал
- 👍 Succeeded - все контейнеры корректно завершили работу
- ☠ Failed - все контейнеры остановлены, как минимум один контейнер некорректно завершил работу (или не прошел проверки в PodSyncHandler)
- 🙃 Unknown - `kube-apiserver` не в курсе, что там с этим подом (обычно, из-за проблем со связью с `kubelet`)

PodStatus || PodConditions

- **PodScheduled** - планировщик выбрал ноду для запуска пода
- **Initialized** - все Init-контейнеры успешно отработали
- **Ready** - Pod готов к работе (остальные условия выполнены) и может быть добавлен в балансировку нагрузки
- **ContainersReady** - все контейнеры запущены

> Используя функционал ReadinessGate можно добавлять свои условия в PodSpec

- **Unschedulable** - планировщик не смог найти подходящую ноду

PodPhase || Pending

Пока Pod висит в статусе Pending `kubelet` делает следующее:

1. Создает cgroups и настраивает ограничения ресурсов
2. Создает служебные папки для Poda
3. Подключает дисковые тома внутрь папки `volumes`
4. Получает реквизиты `ImagePullSecrets`
5. И наконец-то переходит к запуску контейенеров...

PodPhase || Still pending...

1. `kubelet` отправляет CRI-плагину запрос **RunPodSandbox**:
   - в случае с VM-плагинами, создается виртуальная машина
   - в случае с контейнерами, запускается *тот-самый-pause-container* и создаются namespace
2. Когда песочница создана, CRI вызывает сетевой плагин (CNI) и просит его инициализировать сетевое подключение
3. Наконец-то `kubelet` может позапускать контейнеры...

PodPhase || What, still pending???

Теперь `kubelet` может запустить свой первый контейнер:

1. Но сначала запускаются Init-контейнеры. Зачем?
   - Для проверки внешних условий запуска контейнера ("Монго ли я?", "Одинок ли я в этом кластере?")
   - Для предварительной конфигурации и загрузки данных (быстрая копия данных с реплики)
   - Чтобы не выдавать основному контейнеру лишние секреты
2. Теперь запускаем основные контейнеры, описанные в PodSpec
   - И в то же время запускается Post-Start Hook

Наконец-то Pod переходит в статус Running 🎉

Тот-самый-pause-контейнер

Служебный контейнер:

1. Нужен, чтобы зарезервировать Kernel Namespace и сетевую конфигурацию для Pod
2. Умеет убивать zombie-процессы. Но уже не надо:
   - Docker/Moby автоматом запускают tini
   - Shared PID Namespace для пода сначала был включен, а теперь снова выключен (т.к. несекьюрно)
3. Потребление памяти - примерно 180 байт.

> Подробный рассказ есть тут

Проверки Pod

`kubelet` может опрашивать состояние контейнеров в поде. Для этого можно задать два параметра:

- `livenessProbe` - если завершается неуспехом, контейнер перезапускается в соответствии с *RestartPolicy*
- `readinessProbe` - влияет на *PodCondition:* **Ready**

Виды проверок:

- HTTP - проверяет факт подключения и *код возврата HTTP* (успех - это коды от 200 до 399, включительно)
- TCP - проверяет факт подключения по TCP
- Exec - выполняет команду внутри контейнера и смотрит на код возврата

Проверки Pod

- TCP и HTTP-проверки выполняются `kubelet`-ом (по умолчанию, подключение выполняется на IP-адрес Poda и указанный `containerPort`).

> **!** Осторожно, грабли! До версии **1.14** `kubelet` при выполнении HTTP-проверок честно использовал переменные окружения `http_proxy` (или `HTTP_PROXY`) для выполнения `livenessProbe` и `readinessProbe`.

- Проверки Exec выполняются внутри запущенного контейнера, и им доступен весь контекст выполнения Poda.

Проверки Pod | Конфигурация

Проверки определяются непосредственно в спецификации контейнера:

```
apiVersion: v1
kind: Pod
metadata: { name: web, labels: { app: web }}
spec:
  containers:
  - name: web
    image: example/web:1.0
    readinessProbe:
      httpGet: { path: /health, port: 80 }
      periodSeconds: 10
    livenessProbe:
      exec: ['sh', '-c', 'pgrep app-worker-thread']
      initialDelaySeconds: 60
```

Проверки Pod | Конфигурация

- Поддерживается только одна `livenessProbe` и `readinessProbe`
- Общие параметры для любого типа проверок (Exec/HTTP/TCP):
  - `initialDelaySeconds` - время от старта контейнера до первой проверки (особенно актуально для `livenessProbe`)
  - `periodSeconds` - частота проверки
  - `timeoutSeconds` - сколько ждать выполнения проверки (дефолт - 1 секунда, что не всегда хорошо)
  - `failureThreshold` - сколько раз подряд должны получить фейл, чтобы отреагировать (дефолт - 3)
  - `successThreshold` - аналогично, для восстановления статуса *Ready* (не очень актуально для `livenessProbe`)

Проверки Pod | Конфигурация

- HTTP-проверки определяются через `httpGet`:
  - `scheme`, `host`, `port`, `path` - задают куда подключаться,
  - `httpHeaders` - список HTTP-заголовков для запроса

```
httpGet:
  path: /somewhere
  httpHeaders:
    - name: X-Secret-Token
      value: plain-text-secret-value
```

При заданном `scheme: HTTPS` сертификаты не проверяются на валидность.

Проверки Pod | Проблема 🐔 и 🥚

Если в качестве адреса для проверки (ключ `host`) указать имя сервиса, то так не заработает - пока контейнер не пройдет проверку, в Service Discovery ничего не появится.

Если в приложении настроено несколько виртуальных хостов, то правильно задавать их через `httpHeaders`.

Менять ключ `host` рекомендуется в тех редких случаях, когда под подключен к сети хоста (`hostNetwork: true`) и слушает на 127.0.0.1

Проверки Pod | Конфигурация

- TCP-проверки определяются через ключ `tcpSocket`
  - Из доступных параметров - только порт

```
tcpSocket: { port: 179 }
```

- Для Exec-проверок определяются через ключ `exec`:
  - Передается только список аргументов в поле `command`

```
exec: { command: ['sh', '-c', 'kill -2 1'] }
```

Проверки Pod

Если ваше приложение использует gRPC, то ни один из способов проверки не подходит в полной мере.

Условно рекомендованный способ - использовать `grpc-health-probe`

Это CLI-утилита, которую придется скачать/положить внутрь контейнера с приложением и запускать через Exec-проверку.

Pod Lifecycle Hooks

Hooks - это действия, которые выполняет `kubelet` в начале и конце жизненного цикла контейнера:

- `postStart` - выполняется сразу после старта контейнера, параллельно с запуском *entrypoint* (соответственно нет гарантий того, что в контейнере что-то будет запущено в момент старта хука)
- `preStop` - выполняется, когда `kubelet` планирует остановить контейнер. Если контейнер сам завершил работу - хук не вызывается.

> `kubelet` вызывает хук только один раз и не делает повторов (как правило).

Pod Lifecycle Hooks

Хуки блокируют состояние контейнера (и пода) с точки зрения Kubernetes:

- висящий `postStart` не даст контейнеру перейти в состояние **Running**
- висящий `preStop` блокирует отправку SIGTERM процессам контейнера. Pod будет болтаться в статусе **Terminating**, пока не кончится `terminationGracePeriodSeconds`
- если Grace-период закончился, то SIGTERM будет отправлен в контейнер... И у него будет 2 секунды до SIGKILL

> Если при удалении Poda указан `--grace-period=0`, то `preStop` хук не будет вызван

Pod Lifecycle Hooks | Конфигурация

- Настройка хуков делается через блок `lifecycle` в конфигурации контейнера
- Конфигурация аналогична `livenessProbe`/`readinessProbe` (ну, почти)
- Можно указывать `httpGet` и `exec`, и те же параметры что и в проверках
- `tcpSocket` **не** поддерживается, но в спецификации API он есть
- `httpGet` всегда подставляет `http://` в URL (пруф):
  - параметр `scheme` - игнорируется
  - параметр `path` должен быть без `/` в начале

Pod Lifecycle Hooks | Конфигурация

```
apiVersion: v1
kind: Pod
...
spec:
  containers:
  - name: example
    ...
    lifecycle:
      preStop:
        httpGet:
          path: /v1/api/termination-endpoint
      postStart:
        exec:
          command:
            - "sh"
            - "-c"
            - "curl https://register.int/v1/checkin/self"
```

Завершение работы Pod

- При завершении работы Pod обновляется его состояние в хранилище объектов:
  - добавляется временная метка, когда объект должен быть удален (`deletionTimestamp`)
  - добавляется информация о grace-периоде (`deletionGracePeriodSeconds`)
- С этого момента Pod будет виден в UI как **Terminating** (хотя такой фазы у него нет)
- `kubelet` подхватывает измененный объект Pod с `kube-apiserver` и приступает к остановке пода

Завершение работы Pod

- При завершении работы Pod, он удаляется из списка `endpoints`
- Также за ним перестают следить контроллеры (`ReplicaSet` или `ReplicationController`)
- Когда заканчивается grace-период, процессы в Pode получают свой SIGKILL
- В финале, `kubelet` обновляет объект на `kube-apiserver`:
  - `deletionTimestamp` возвращается уменьшается на значение grace-period
  - `deletionGracePeriodSeconds` выставляется равным `0`
  - клиенты API перестают видеть объект, он помечается на удаление

Запуск нескольких Pod

Контроллеры

- Нужны для управления группами однотипных подов
- Такие группы подов описываются как отдельный виды объектов
- Каждый контроллер работает со своим видом объектов
- Объект = шаблон PodSpec + селектор + настройки контроллера
- Контроллер = Немножко логики + подписка на события + создание объектов

Есть контроллеры, которые контролируют контроллеры, но о них дальше

ReplicationController

- Первый заход в сторону управления подами
  - до сих пор в API `core/v1`
- Следит за тем, чтобы число подов соответствовало заданному
  - Умеет пересоздавать Pods при отказе узла (обычные "голые" поды умирают вместе нодой)
  - Умеет добавлять/удалять Pods не пересоздавая всю группу
- **НЕ** проверяет соответствие запущенных Podов шаблону

> RC еще встречаются в дикой природе, например, в OKD/OpenShift

ReplicaSet

- Почти как `ReplicationController`, такой же ограниченный
- Добавлена поддержка *set-based* селекторов:
  - было:

```
selector: { app: nginx, tier: front, environment: stage }
```

  - стало:

```
selector:
  matchLabels: { app: nginx }
  matchExpressions:
    - { key: tier, operator: In, values: [front] }
    - { key: environment, operator: NotIn, values: [prod] }
```

- Убрали авто-генерацию селектора

Лирическое отступление: Set-based selector

- Очевидно, синтаксис стал более гибким
- Для `ReplicaSet` и `ReplicationController` выполняется проверка селектора и меток в шаблоне Pod:

```
  Invalid value: map[string]string{"app":"web"}: `selector` does not match
template `labels`
```

- Но теперь можно "населектить" лишнего и пройти проверку

На слайде выше - вполне валидный сниппет. Но "голые" поды в окружениях `dev` или `test` не смогут запуститься - `ReplicaSetController` пристрелит их почти сразу

ReplicaSet || Пример манифеста

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: rset_web_1-0
spec:
  replicas: 2
  selector: { matchLabels: { app: web }}
  template:
    metadata: { labels: { app: web }}
    spec:
      containers:
      - image: avtandilko/web:1.0
        name: web
```

⭐ Бета-версии API `ReplicaSet` позволяли менять селектор в созданном объекте (не пробуйте это в Production)

Job

- Запуск одного или нескольких pod для выполнения разовых задач
- Контроллер следит за общим временем выполнения задачи, останавливая все Pods при превышении
- Контроллер следит за числом параллельно запущенных подов, запуская новые, по мере необходимости
- Контроллер следит за общим числом запусков подов
- Метки и селекторы контроллер задает сам, но можно указать вручную (лучше не стоит)
- Увы, планировщик не делает отличий между batch-подами и обычными

Job || Пример манифеста

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: very_good_job
spec:
  # Число попыток, перед Failed с нарастающим интервалом
  backoffLimit: 4
  # Максимальная продолжительность Job
  activeDeadlineSeconds: 60
  # Количество одновременно запущенных Podов
  parallelism: 3
  # Разрешить TTLController-у прибрать остатки, 0 - чистить сразу
  # На данный момент требует включить Feature Gate TTLAfterFinished
  ttlSecondsAfterFinished: 600
  # Суммарное число запусков Podов
  completions: 9
  template:
    spec:
      containers:
      ...
      restartPolicy: OnFailure  # или Never
```

CronJob

Это первый пример контроллера контроллеров :)

- Генерация и запуск Job по расписанию:
  - Однократно в заданное время
  - Периодически
- Регулярные задачи останавливаются, если пропустили последние 100 запусков
- Также он отслеживает "наложение" задач по времени и может управлять такими ситуациями

CronJob || Пример манифеста

```yaml
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: periodic_job_runner
spec:
  #  Стандартный формат расписания для Cron
  schedule: "@hourly"
  # Сколько секунд есть для запуска задачи,
  # если пропустили запланированное время
  startingDeadlineSeconds: 200
  # Что делать с наложением задач во времени:
  #   forbid – не запускать новую
  #   allow – запускать параллельно
  #   replace – удалять старые задачи
  concurrencyPolicy: replace
  # Притормозить планировщик
  suspend: false
  # Хранить ли историю запущенных задач
  successfulJobsHistoryLimit: 0
  failedJobsHistoryLimit: 0
  jobTemplate:
    ...
```

Пора деплоить...

Итак:

- `Replica**Controller` не умеют рестартовать запущенные поды при обновлении шаблона
- Мы можем попробовать обновить `template` и менять `replicas` вверх и вниз

1. `replicas: 2, labels: { app: web, version: v1 }`

```
$ kubectl get pods -L version
NAME        READY   STATUS    RESTARTS   AGE   VERSION
web-ksdh9   1/1     Running   0          24s   v1
web-nvqf5   1/1     Running   0          24s   v1
```

Пора деплоить...

2. `replicas: 4, labels: { app: web, version: v2 }`

```
$ vkubectl get pods -L version
NAME        READY   STATUS     RESTARTS   AGE   VERSION
web-bp6xb   0/1     Init:0/1   0          2s    v2
web-ksdh9   1/1     Running    0          30s   v1
web-nvqf5   1/1     Running    0          30s   v1
web-sbm76   0/1     Init:0/1   0          2s    v2
```

3. `replicas: 2, labels: { app: web, version: v2 }`

```
kubectl get pods -L version
NAME        READY   STATUS        RESTARTS   AGE   VERSION
web-bp6xb   1/1     Terminating   0          27s   v2
web-ksdh9   1/1     Running       0          55s   v1
web-nvqf5   1/1     Running       0          55s   v1
```

Oops, `ReplicationController` удалил **новые** поды 🤔

Пора деплоить...

Нужен другой алгоритм:

- Создать второй `ReplicaSet` с новой конфигураций
- Одновременно:
  - уменьшаем `replicas` в старом `ReplicaSet`
  - увеличиваем `replicas` в новом `ReplicaSet`
- Получаем возможность запускать канареек, делать Blue/Green, rolling update и rip-and-replace
- Старожилы припомнят, что так работал `kubectl rolling-update` в связке с `ReplicationController`
- ГлавНО - приходится это делать руками

Deployment

- Это контроллер контроллеров, управляющий `ReplicaSets`
- Делает то, что описано на слайде выше, но без ручного вмешательства, внутри кластера K8s
- Соответственно, это "декларативный" способ развертывания
- И рекомендованный способ запуска Podов (даже если нужна только одна реплика)

> OKD пошел своей дорогой, создавая специальный под, управляющий развертыванием

Deployment

Для управления развертываниями через `kubectl` используется подкоманда `rollout`

- `rollout status` - посмотреть текущее состояние `Deployment`
- `rollout history` - посмотреть на историю развертываний (и причины изменений, если есть)
- `rollout undo` - вернуться к предыдущей версии `Deployment`
- `rollout pause` / `resume` - приостановить обновление или продолжить

Deployment || Состояния и условия

`Deployment` может быть в трех состояниях:

- `Progressing` - новые поды создаются и все идет хорошо
- `Complete` - мы все выкатили, больше ничего не делаем
- `Failed` - что-то пошло не так

На уровне API указанных состояний как таковых нет, но есть метрики и условия (`conditions`) в объекте `DeploymentStatus`.

Как и в случае с Podами, условия состоят из названия, статуса и текстового описания причины статуса.

Deployment || Состояния и условия

Всего определено три условия:

- `Available` - если минимально необходимое количество реплик запущено и готово к работе
- `Progressing` - все реплики были созданы в срок
- `ReplicaFailure` - добавляется к статусу, если не удалось удалить или добавить поды

Метрики в `DeploymentStatus`:

- **availableReplicas** - реплики, готовые к работе
- **unavailableReplicas** - число недостающих реплик
- **updatedReplicas** - число реплик, соответствующих шаблону
- **replicas** - число подов, попадающих в селектор

Параметры Deployment

- `minReadySeconds` - время, которое под должен быть стабилен и Ready, для продолжения деплоя (*0*)
- `paused` - развертывание приостановлено (*false*)
- `progressDeadlineSeconds` - фейлим деплой, если не было прогресса (*600s*)
- `revisionHistoryLimit` - количество хранимых версий `Deployment` (*10*)
- `strategy` - стратегия для деплоя. Может быть **Recreate** и **RollingUpdate** (дефолт)

Стратегии Deployment

- Стратегия **Recreate** не имеет настроек. Фактически, это удаление старого `ReplicaSet` и создание нового.
- **RollingUpdate** можно покрутить в разные стороны:

```
strategy:
  type: RollingUpdate
  rollingUpdate:
    # Количество реплик, которое можно создать с превышением значения replicas
    # Можно задавать как абсолютное число, так и процент. Default: 25%
    maxSurge: 0
    # Количество реплик от общего числа, которое можно "уронить"
    # Аналогично, задается в процентах или числом. Default: 25%
    maxUnavailable: 1
```

Deployment

Задавая параметры `maxSurge` и `maxUnavailable`, можно регулировать скорость деплоя и масштаб трагедии:

- `maxSurge: 100%` и `maxUnavailable: 0` - имитация blue-green, если вовремя поставить на паузу
- `maxUnavailable: 100%` - аналогично **Recreate**
- `maxSurge > 0`, `maxUnavailable: 0` - имитация канареечных релизов

> Рекомендованный способ для blue-green и canary, это не ставить деплой на паузу, а делать два разных `Deployment`

StatefulSet

- Управляет выкаткой и масштабированием pod также как и `Deployment`
- Гарантирует очередность запуска и уникальность запущенных pod
- Гарантирует, что pod не сменит свои характеристики (name, PVC) при пересоздании
- Необходимо вручную обрабатывать отказы хостов с pod, управляемыми `StatefulSet`

Применяется, когда приложению нужно:

- Постоянные уникальные сетевые идентификаторы
- Отличное от Deployment поведение при работе с томами

DaemonSet

- Обеспечивают приоритетный запуск Poda на каждой ноде кластера
- `DaemonSet` могли быть запущены до старта `kube-scheduler`
  - Начиная с версии 1.12 они используют штатный планировщик
  - Но добавляют ряд `tolerations`, чтобы почти гарантированно запуститься
  - Вместо `nodeName` использует механизм `NodeAffinity` для привязке к ноде
- Поддерживают последовательное обновление узлов

DaemonSet || Пример манифеста

```yaml
kind: DaemonSet
apiVersion: apps/v1
metadata:
  name: prometheus-node-exporter
  namespace: Service
  labels: { k8s-app: prometheus, component: node-exporter }
spec:
  # Стратегия обновления Podов
  updateStrategy: onDelete
  # Обязательный атрибут с версии 1.8
  selector:
    matchLabels: { k8s-app: prometheus, component: node-exporter }
  template:
    metadata:
      name: prometheus-node-exporter
      labels: { k8s-app: prometheus, component: node-exporter }
    spec:
      containers:
        ...
```

Ссылки

- What happens when про K8s
  - слегка устарел, но в целом **must-read**
- Репортаж Из жизни подов в блоге OKD
- Репозиторий с документацией SIGов по изменениям в Kubernetes, в этих документах есть ссылки на обсуждения и PRы, из которых становится понятно те только "как?", но и "почему?". Линк
- Инструмент Kubespy и короткое демо в блоге Pulumi

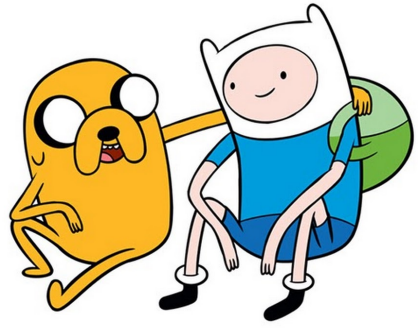
Что еще?

К этой лекции не предусмотрено домашнего задания, но вы можете поэкспериментировать с демо-манифестами (инструкции написаны в комментариях внутри каждого файлика).

А также можно починить `web-deployment.yaml`. Задача не столько в том, чтобы написать правильный манифест для деплоя, а в том, чтобы поиграться с инструментами для дебага и насмотреться на разные сообщения об ошибках.

Just relax and have fun!

Файлики лежат тут



Спасибо за внимание!

Время для ваших вопросов!