

# **Механика запуска и взаимодействия контейнеров**

# Не забудь включить запись!



# План

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

# 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  
далше

# ReplicationController

- первый, но до сих пор в API
- следит за тем чтобы число подов соответствовало заданному
- Note: A Deployment that configures a ReplicaSet is now the recommended way to set up replication.

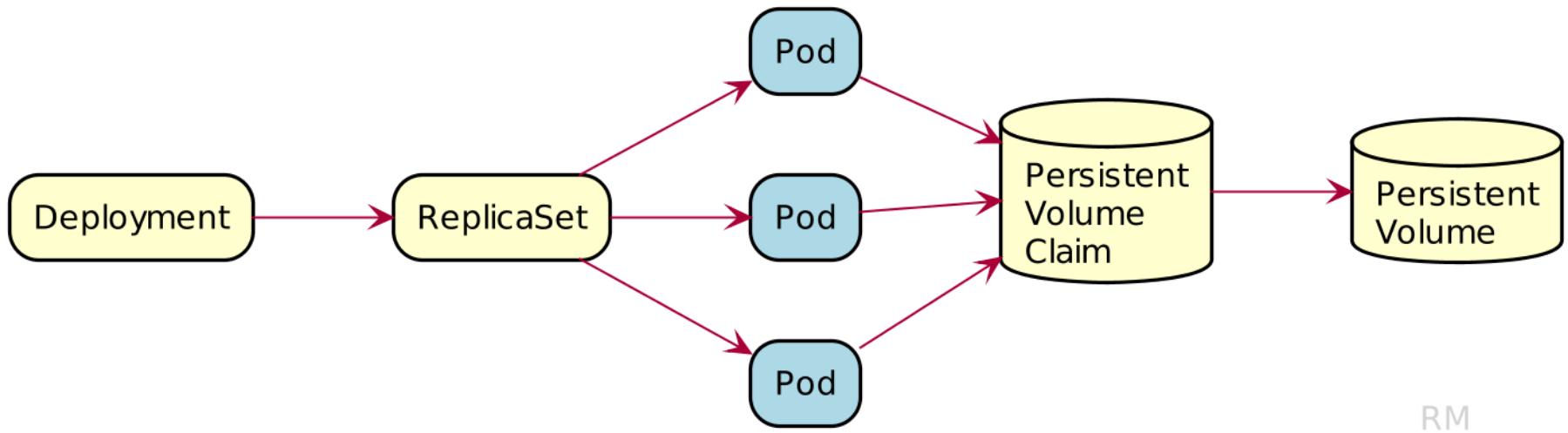
# Deployment and ReplicaSet, DaemonSet, StatefulSet

controller

manage Pods

update strategy

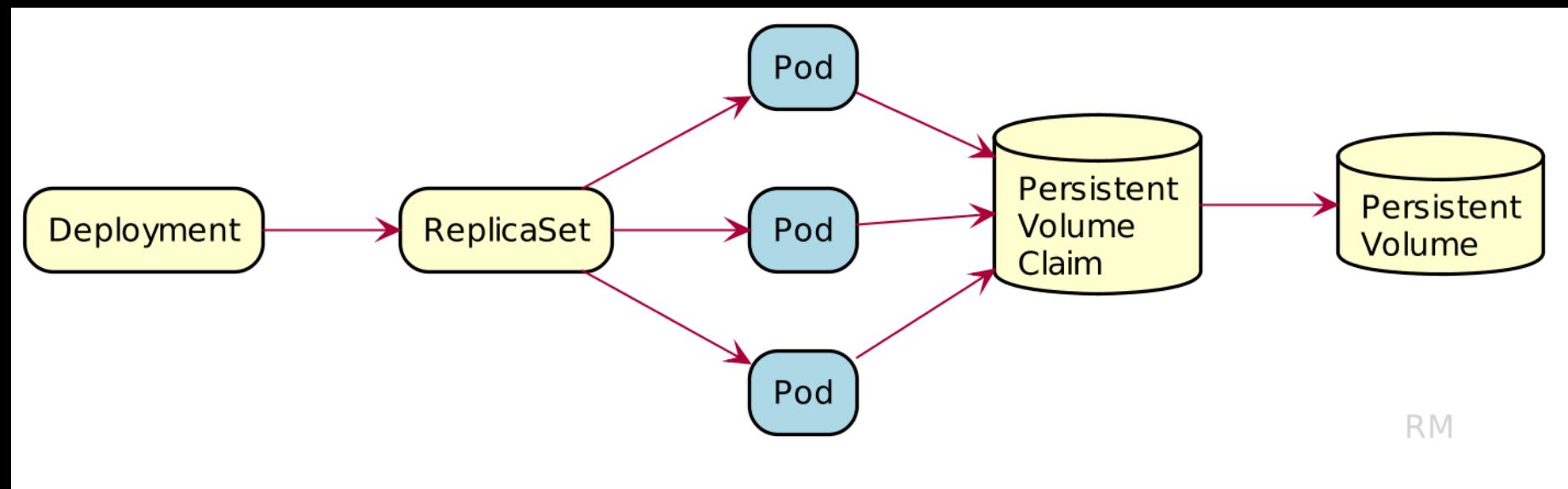
# Deployment and ReplicaSet



no unique identities

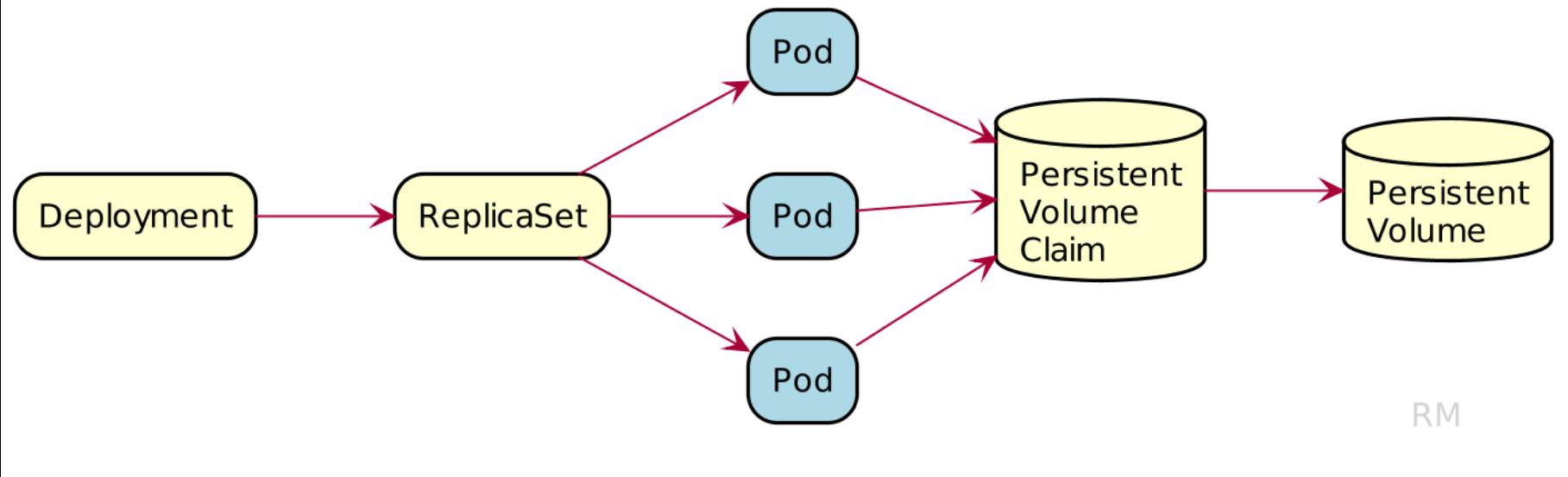
stateless apps

# ReplicaSet



specified number of identical Pods

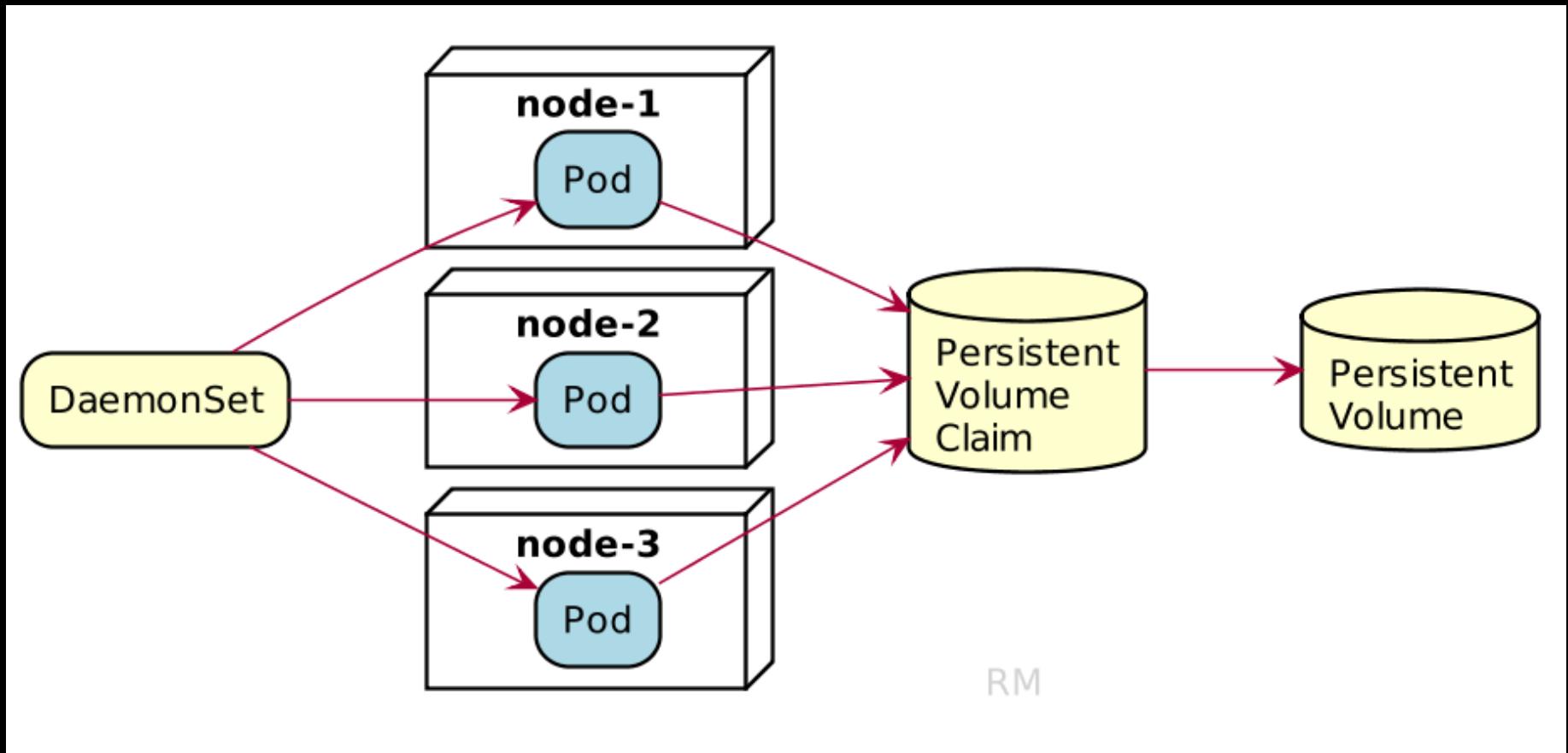
# Deployment



контроллер контроллеров

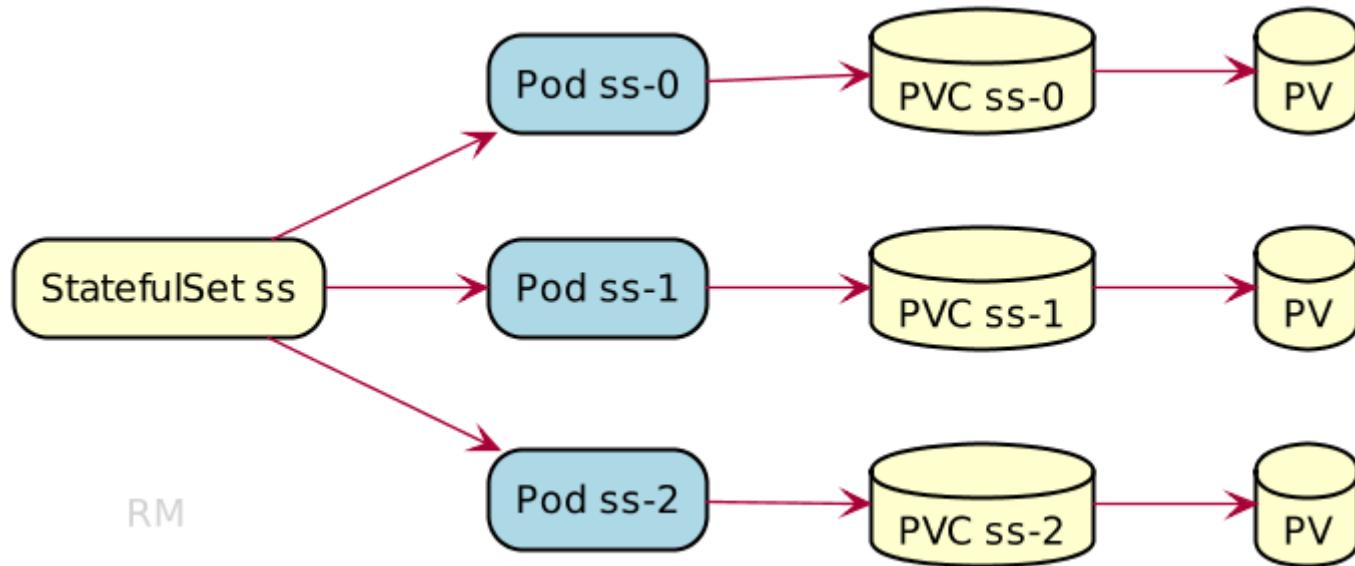
best practice: Naked pod vs ReplicaSet and  
Deployment

# DaemonSet



all Nodes have a Pod

# StatefulSet



unique, persistent identities and stable hostnames

ordered deployment

stateful apps

# StatefulSet

- Управляет выкаткой и масштабированием pod также как и **Deployment**
- Гарантирует очередь 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

# Job

- Запуск одного или нескольких pod для выполнения разовых задач
- Контроллер следит за общим временем выполнения задачи, останавливая все Podы 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

```
1 apiVersion: batch/v1
2 kind: Job
3 metadata:
4   name: very_good_job
5 spec:
6   # Число попыток, перед Failed с нарастающим интервалом
7   backoffLimit: 4
8   # Максимальная продолжительность Job
9   activeDeadlineSeconds: 60
10  # Количество одновременно запущенных Podов
11  parallelism: 3
12  # Разрешить TTLController-у прибрать остатки, 0 – чистить сразу
13  # На данный момент требует включить Feature Gate TTLAfterFinished
14  ttlSecondsAfterFinished: 600
15  # Суммарное число запусков Podов
16  completions: 9
17  template:
18    spec:
19      containers:
20      ...
21      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

```
1 apiVersion: batch/v1beta1
2 kind: CronJob
3 metadata:
4   name: periodic_job_runner
5 spec:
6   # Стандартный формат расписания для Cron
7   schedule: "@hourly"
8   # Сколько секунд есть для запуска задачи,
9   # если пропустили запланированное время
10  startingDeadlineSeconds: 200
11  # Что делать с наложением задач во времени:
12  #   forbid – не запускать новую
13  #   allow – запускать параллельно
14  #   replace – удалять старые задачи
15  concurrencyPolicy: replace
16  # Притормозить планировщик
17  suspend: false
18  # Хранить ли историю запущенных задач
19  successfulJobsHistoryLimit: 0
20  failedJobsHistoryLimit: 0
21  jobTemplate:
22    ...
```

# Запуск Pod

# 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

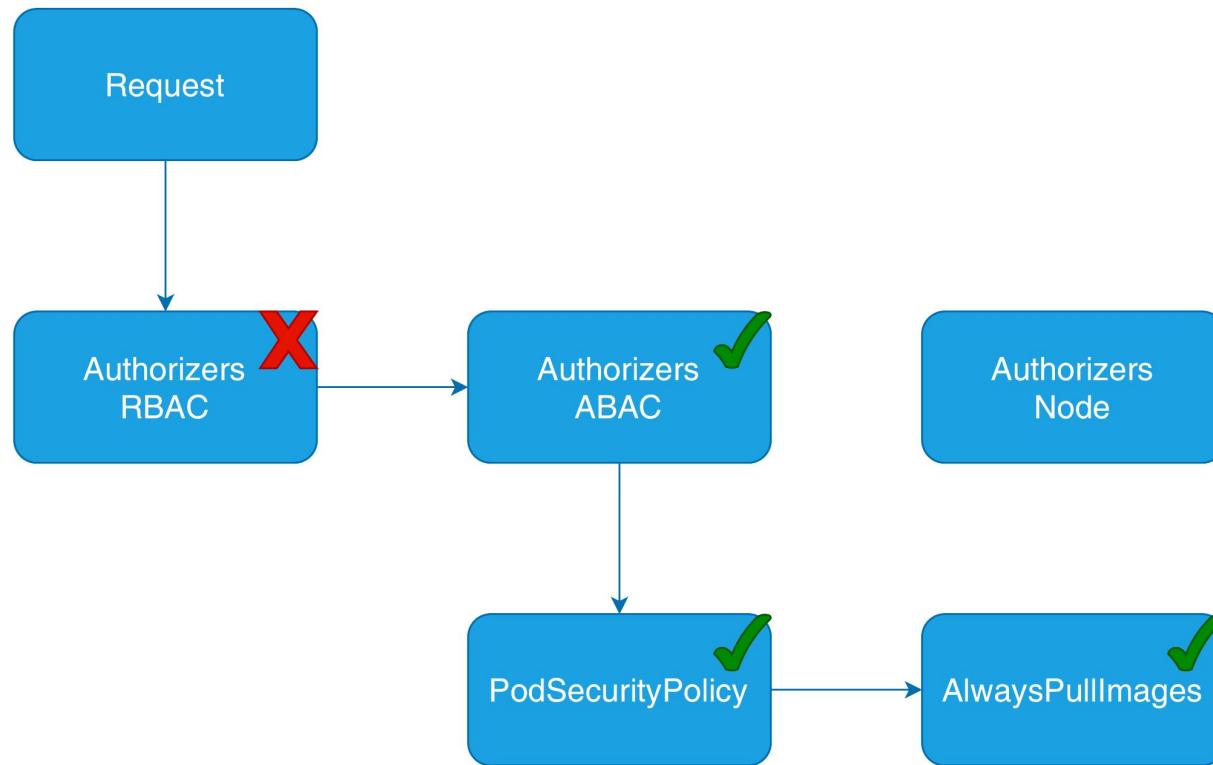
6. Основной цикл обработки ресурсов (e.g. Deployment Controller)
7. Запуск планировщика
8. Магия `kubelet` 
9. Магия runtime. Подготовка изолированного окружения
10. Магия runtime. Инициализация сети
11. Магия runtime. Запуск контейнеров
12. И снова `kubelet` ... Запуск хуков

# 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`)

# Тот самый AAA

## Схема прохождения запроса



# 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 2

7. Проверки на совместимость с Runtime конфигурацией
  8. Доступные ресурсы, Pod priority
  9. Периодические проверки запущенных Pod, например ActiveDeadline для Job
- Итог: Pod прошедший все предыдущие проверки включая планировщик не обязательно будет запущен

# 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`)

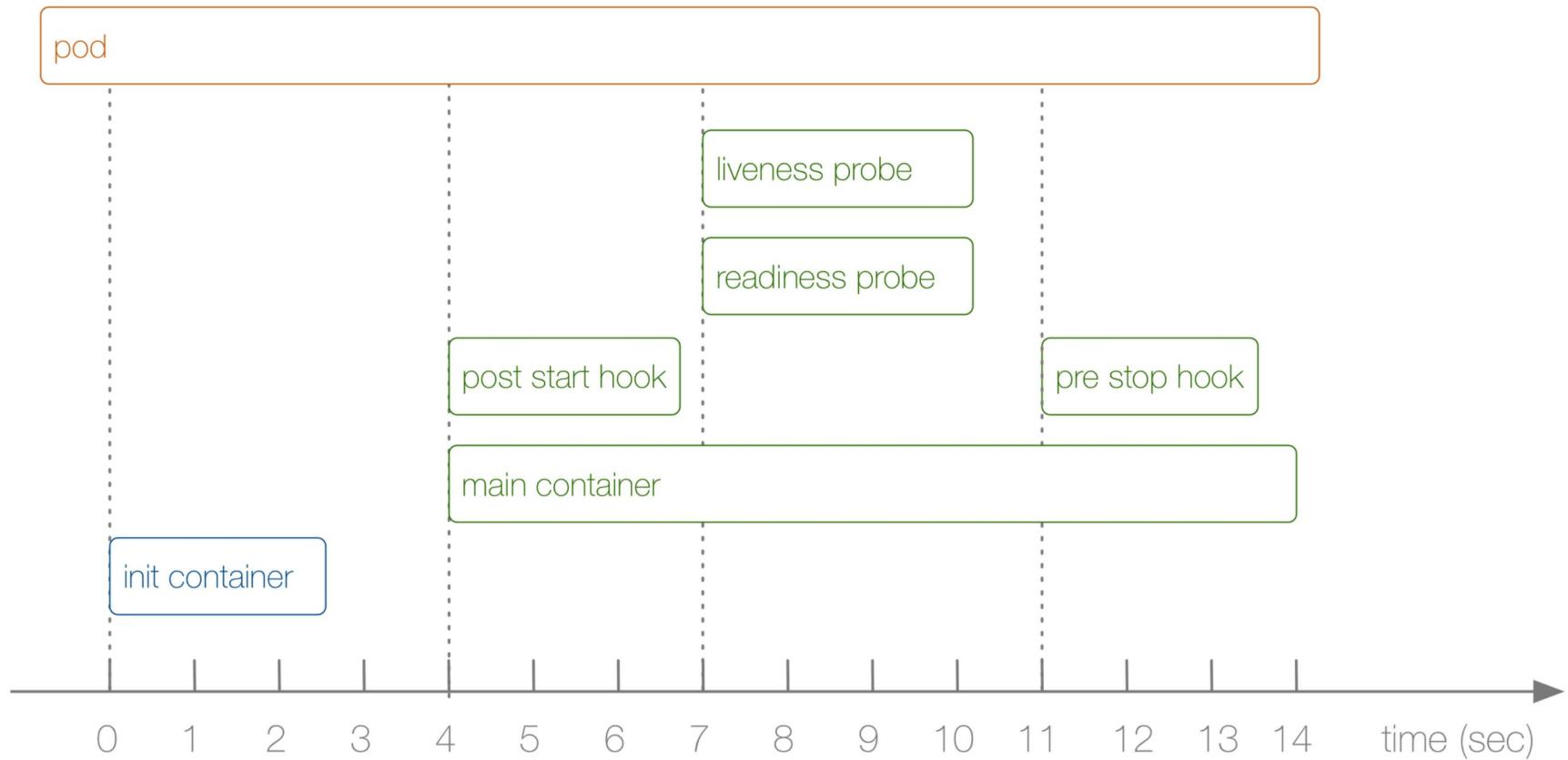
# PodStatus || PodConditions

- **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

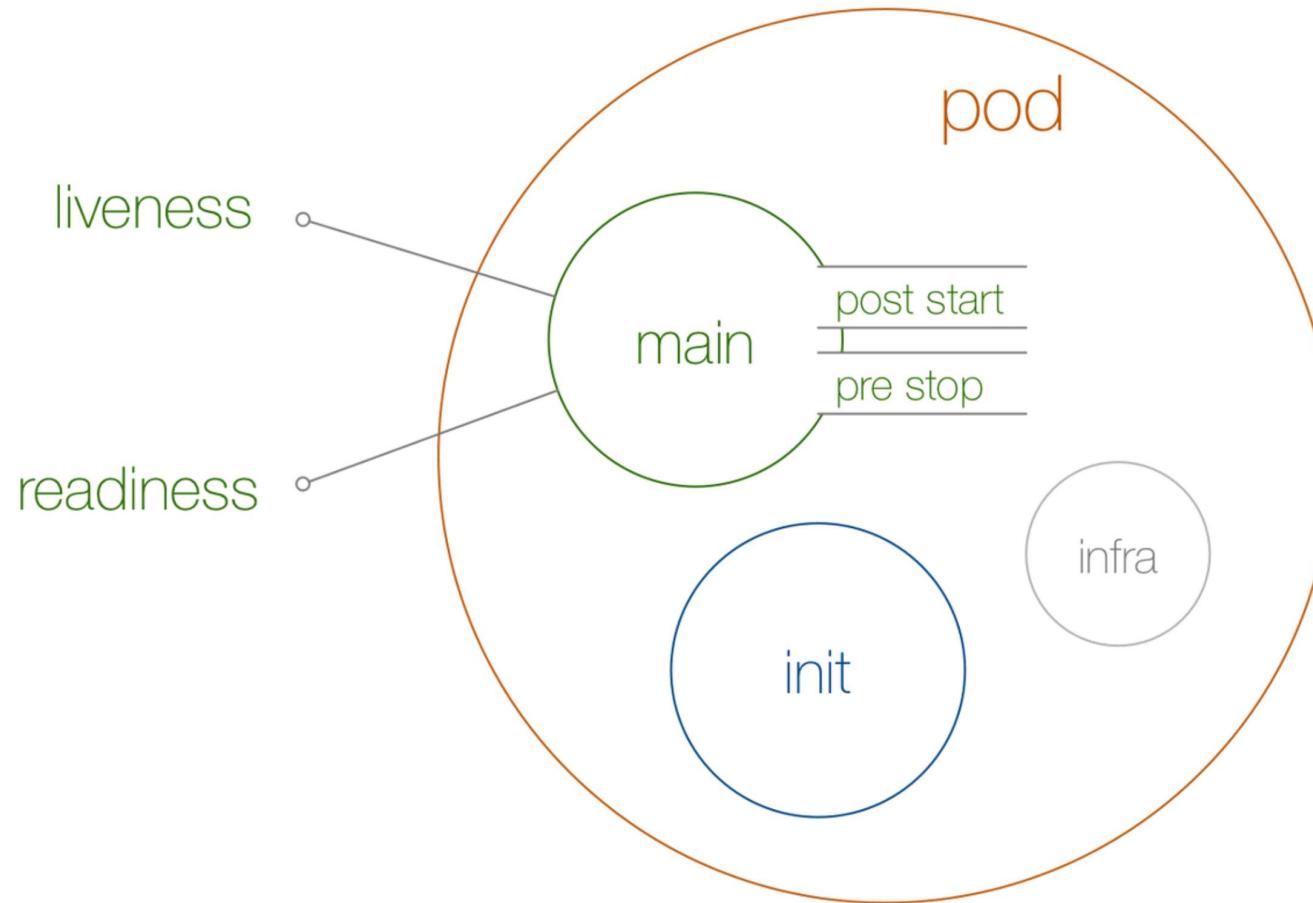
- **Unschedulable** - планировщик не смог найти подходящую ноду

# Запуск Pod



Источник: блог OKD, ссылка в конце презентации

# Структура Pod



Источник: блог OKD, ссылка в конце презентации

# PodPhase II 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 контейнеров...

# **PodPhase II 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 II What, still pending???

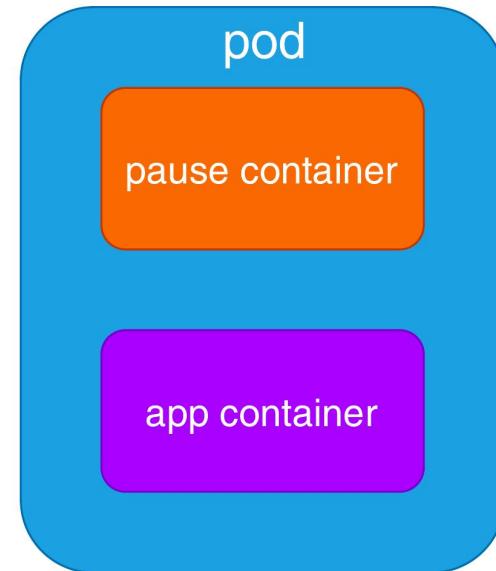
Теперь **kubelet** может запустить свой первый контейнер:

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

- Нужен, чтобы зарезервировать Kernel Namespace и сетевую конфигурацию для Pod
- 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)
- Потребление памяти - примерно 180 байт.



Подробный рассказ есть [тут](#)

# Проверки Pod

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

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

В версии 1.16 добавился параметр `startupProbe` - он позволяет отложить выполнение других проверок, если первый запуск приложения занимает значительное время. После первого успешного выполнения `startupProbe` начинают выполняться `livenessProbe`.

# Проверки Pod

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-адреса Poda и указанный `containerPort`).

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

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

# Проверки Pod | Конфигурация

Проверки определяются непосредственно в спецификации контейнера:

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: web
5   labels:
6     app: web
7 spec:
8   containers:
9     - name: web
10    image: example/web:1.0
11    readinessProbe:
12      httpGet:
13        path: /health
14        port: 80
15        periodSeconds: 10
16    livenessProbe:
17      exec: ['sh', '-c', 'pgrep app-worker-thread']
18      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 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 Ехес-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 Ехес-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:

- висячий `postStart` не даст контейнеру перейти в состояние **Running**
- висячий `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

```
1 apiVersion: v1
2 kind: Pod
3 ...
4 spec:
5   containers:
6     - name: example
7       ...
8     lifecycle:
9       preStop:
10         httpGet:
11           path: /v1/api/termination-endpoint
12       postStart:
13         exec:
14           command:
15             - "sh"
16             - "-c"
17             - "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 Podе 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

# Ссылки

- [What the hell is a pod anyways](#)
- 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 Kubespuy и короткое демо в [блоге Pulumi](#)

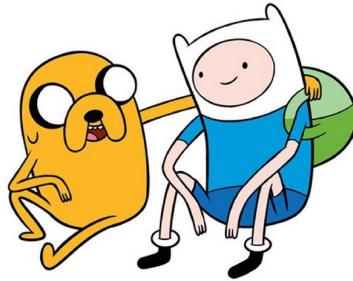
# Что еще?

Помимо домашнего задания, вы можете поэкспериментировать с демоманифестами (инструкции написаны в комментариях внутри каждого файлака).

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

**Just relax and have fun!**

Файлики лежат [тут](#)



# Спасибо за внимание!

**Время для ваших вопросов!**