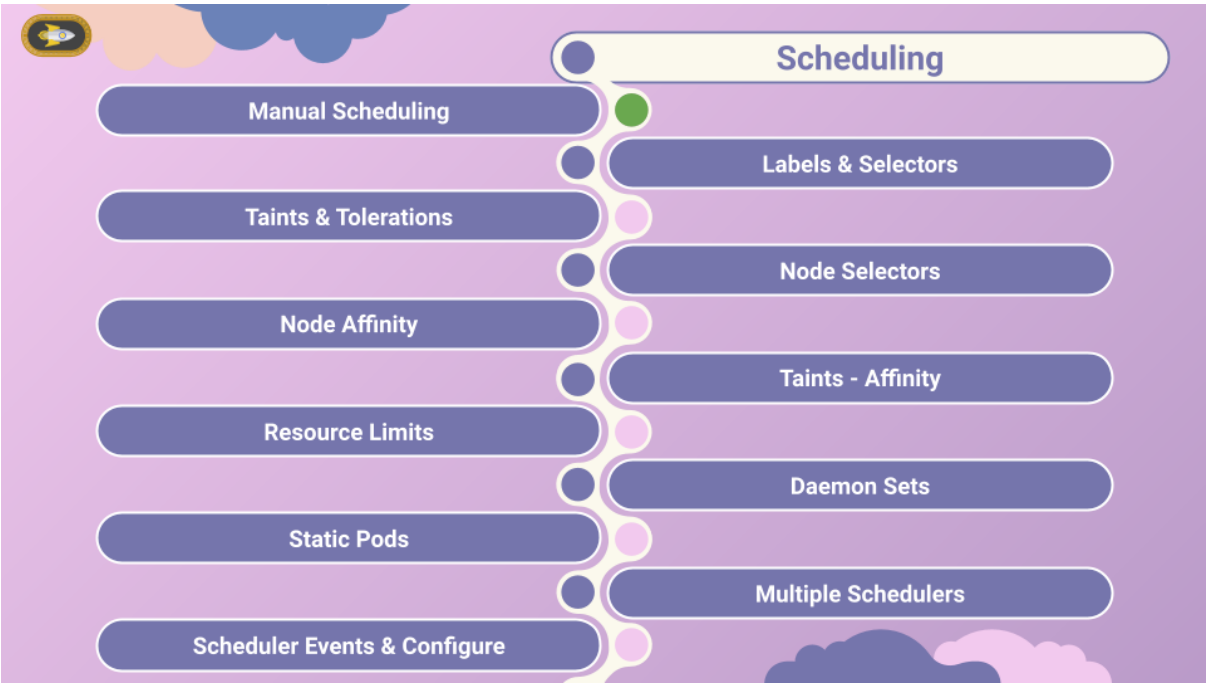


Привет и добро пожаловать на эту лекцию. В этой лекции мы рассмотрим различные способы ручного планирования POD на узлы.

Что нам делать, если у нас в кластере нет планировщика?

Например, мы не хотим полагаться на встроенный планировщик, а вместо этого желаем самостоятельно назначать PODs.

Итак, как именно scheduler работает в глубине?

Начнем с простого файла определения POD.

У каждого POD есть поле под названием `nodeName`, которое по умолчанию не установлено.

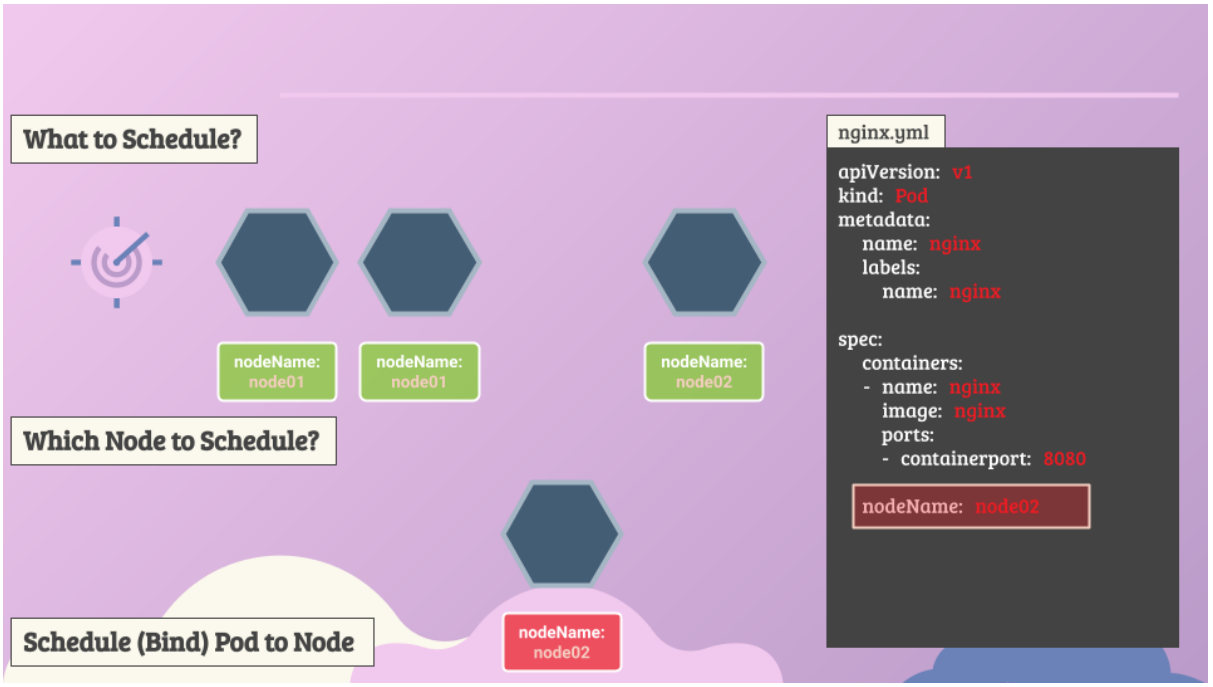
Обычно мы не указываем это поле при создании файла манифеста, Kubernetes добавляет его автоматически. Планировщик просматривает все PODs и ищет те, для которых это свойство не задано.

Это кандидаты на назначение. Затем он определяет правильный узел для POD, запуская алгоритм шедулинга. После определения ноды, он назначает POD на этот узел, устанавливая свойство nodeName в значение имени выбранного узла.

Итак, если у нас нет scheduler, то некому мониторить и планировать нагрузки на ноды. Что же тогда будет происходить?

PODs продолжат находиться в состоянии ожидания.

И что мы можем с этим поделать?



Ну, например, можно вручную назначить PODs на ноды. Самый простой способ назначить POD без планировщика - это при создании установить в его поле nodeName название той ноды, которая примет этот POD.

После создания POD назначается указанному узлу. Ты можешь указать имя узла только во время создания.

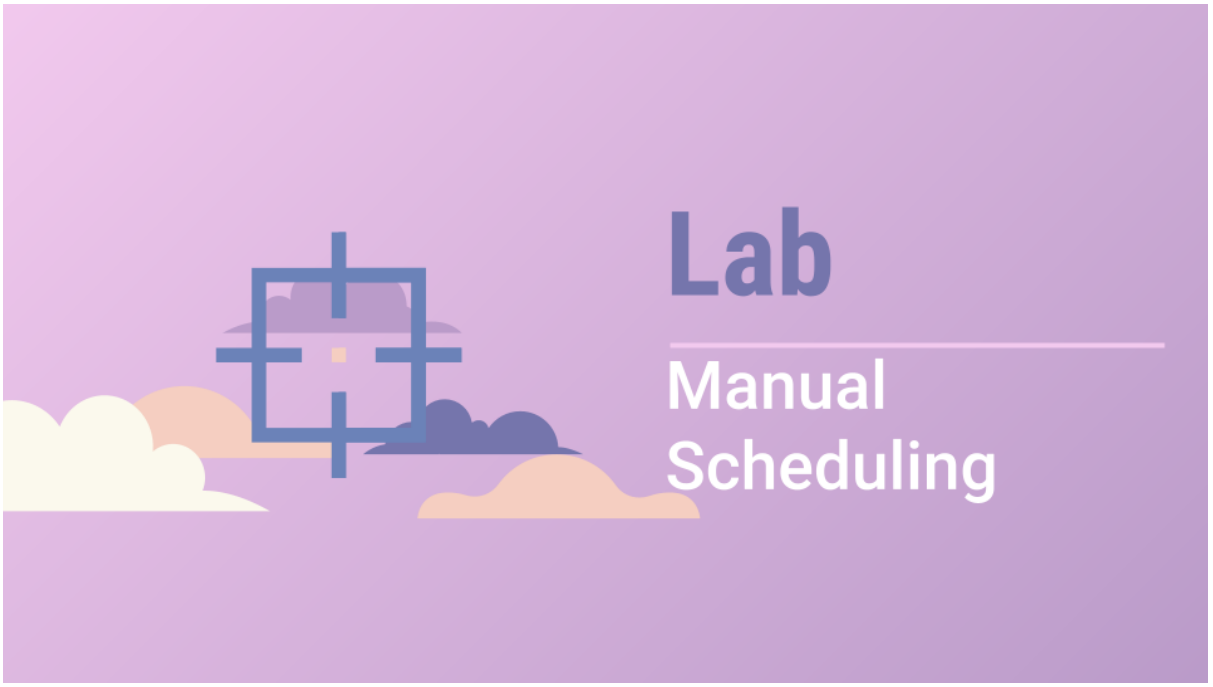
Но что делать, если POD уже создан, а мы хотим переназначить его на другой узел? Kubernetes не позволит нам изменить свойство nodeName на работающем POD.

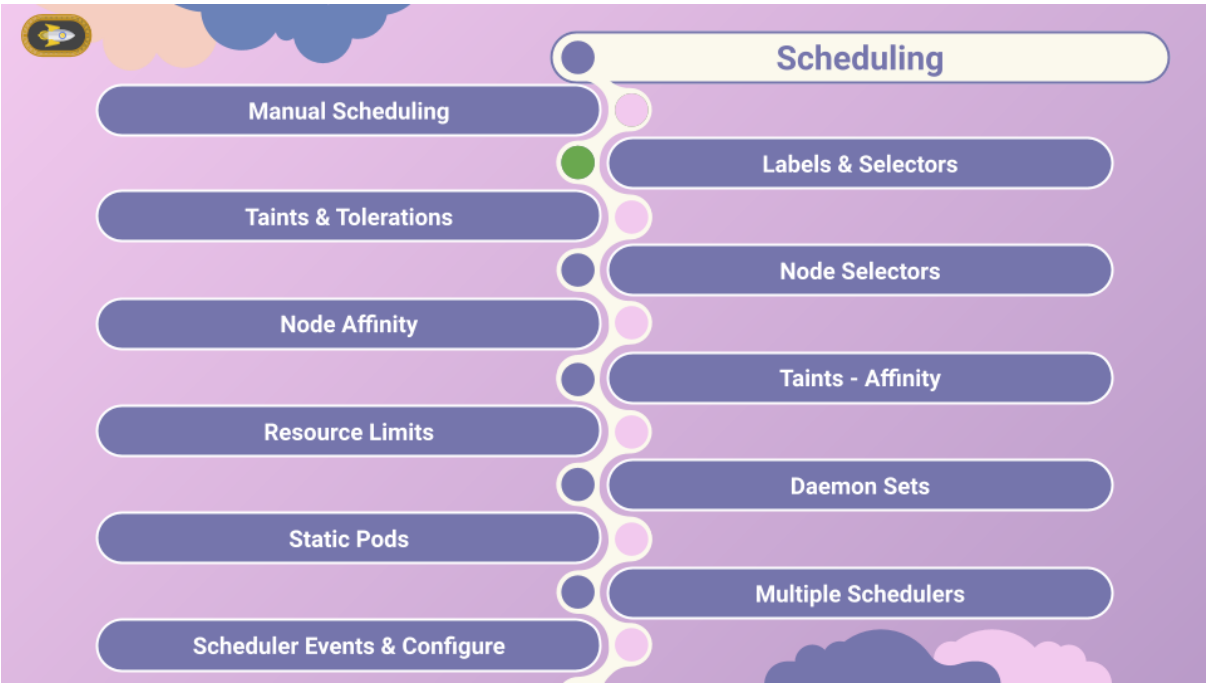
Ок, еще один способ назначить POD - создать объект привязки, объект с kind: `Binding` и отправить его в виде JSON с помощью POST-запроса в binding-api. Этим самым мы как бы притворяемся scheduler и делаем то, что он сам делает в обычной ситуации.

В объекте привязки нам нужно указать в разделе `target` имя целевой ноды, а затем отправить его в API этого POD, как ты видишь.
В связи с этим нужно быть уверенным, что твой преобразованный YAML эквивалентен данным в JSON-формате.

Ну вот и все о ручном планировании.

Переходи к лабораторной и потренируй ручное назначение PODs.


Lab
Manual
Scheduling



Привет и добро пожаловать на лекцию о labels и selectors.

Что мы знаем о метках и селекторах?

Метки и селекторы являются стандартным методом группировки элементов.

Допустим, у нас есть набор разных видов существ из самых разных категорий.

Я хочу иметь возможность фильтровать их на основе различных критериев, например, на основе их класса или вида, а не только по общему признаку, что все они живые.

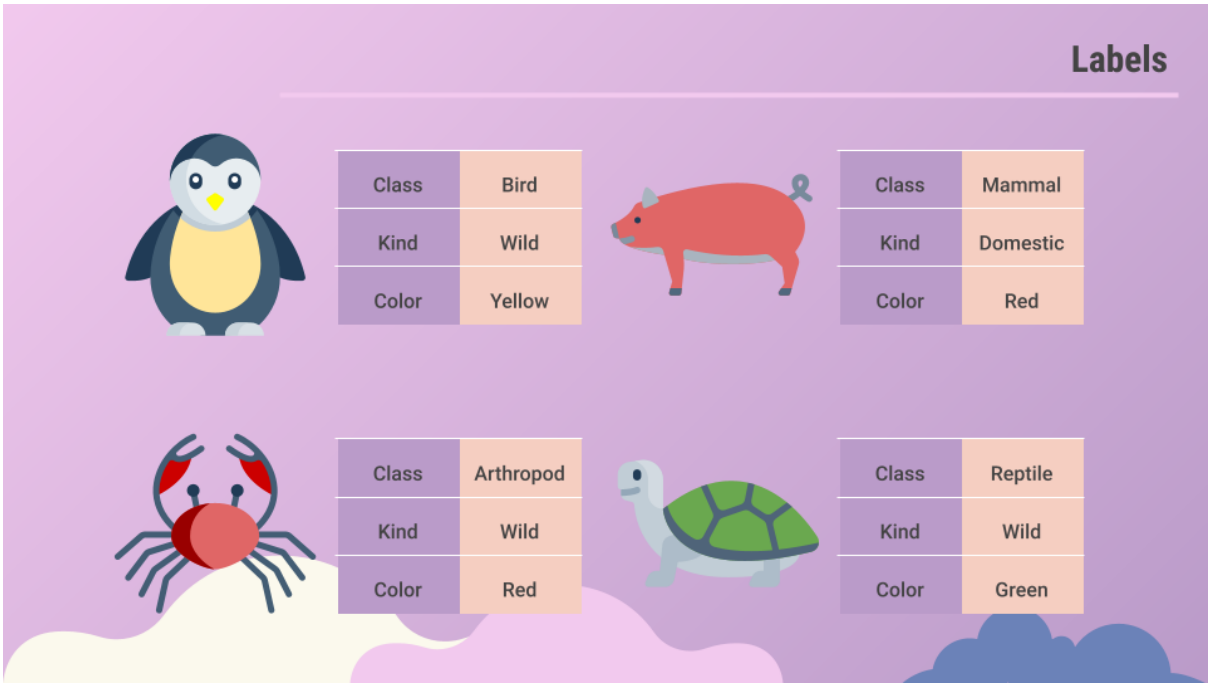
Например, домашние они или дикие, того или иного цвета.

Я желаю иметь возможность фильтровать их по нескольким критериям, таким как все желтые и при этом животные, а дальше выбрать из этого только птиц.

Какой бы ни была эта классификация, нам понадобится возможность группировать вещи и фильтровать их в зависимости от своих потребностей.

Лучший способ сделать это - с помощью меток. Метки это описание свойств, прикрепленные к каждому элементу.

Таким образом, мы добавляем эти описания-метки - пары `название свойства` - `значение свойства` к каждому элементу в соответствии с его типом, а селекторы свойства `цвет` помогают фильтровать эти элементы.



Например, когда мы говорим, что тип цвет равен `красный`, мы получаем список всех красных существ, а когда говорим, что тип равен `дикий`, мы получаем красных и диких существ.

Мы видим метки и селекторы постоянно, они используется всюду, например, ключевые слова, в тегах к видео на YouTube, которые помогают пользователям фильтровать и находить нужный контент.
Или категории, которые представляют собой ярлыки для группировки товаров в интернет-магазине, которые реализуют функции различных фильтров для покупателей.

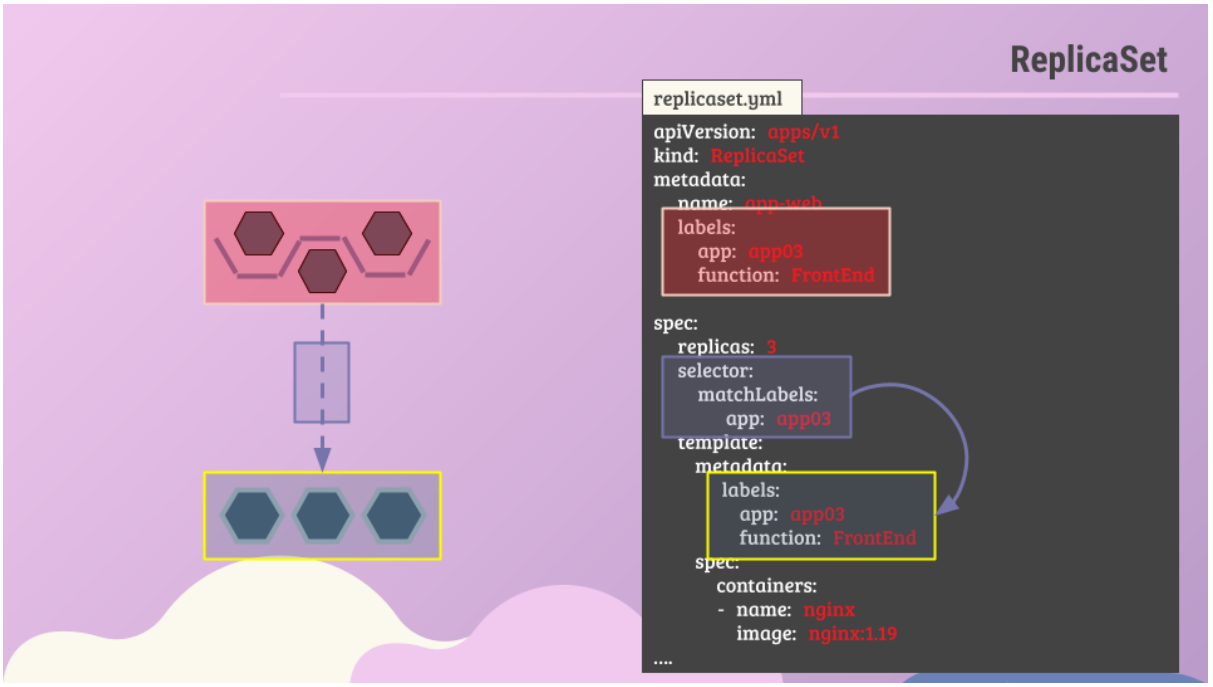
Так как же используются метки и селекторы в Kubernetes?

В Kubernetes мы создаем множество различных типов объектов: PODs, Services, Replicasets, Deployments т. д. Для Kubernetes все это разные объекты.

Со временем в твоем кластере могут оказаться сотни или тысячи таких объектов.
И тебе необходим способ фильтровать и просматривать различные объекты по разным категориям. Например, группировать объекты по их kind или просматривать объекты по приложениям или по их функциональности, какими бы они ни были.

Мы можем группировать и выбирать объекты, используя labels and selectors для каждого объекта, прикрепляя эти метки в соответствии со своими потребностями. Например описав, что это за приложение, его функционал, его уровень в ярусах архитектуры и т. д.

Далее это условие будет использоваться для фильтрации нужных объектов при выборе. Например, `app` равно `app07`.



Итак, как именно указать метки в Kubernetes?

В файле определения POD в секции metadata создай раздел с именем label. В него добавляй метки в формате ключ-значение, подобном тому, как видишь.
Ты можешь добавить сколько угодно меток.

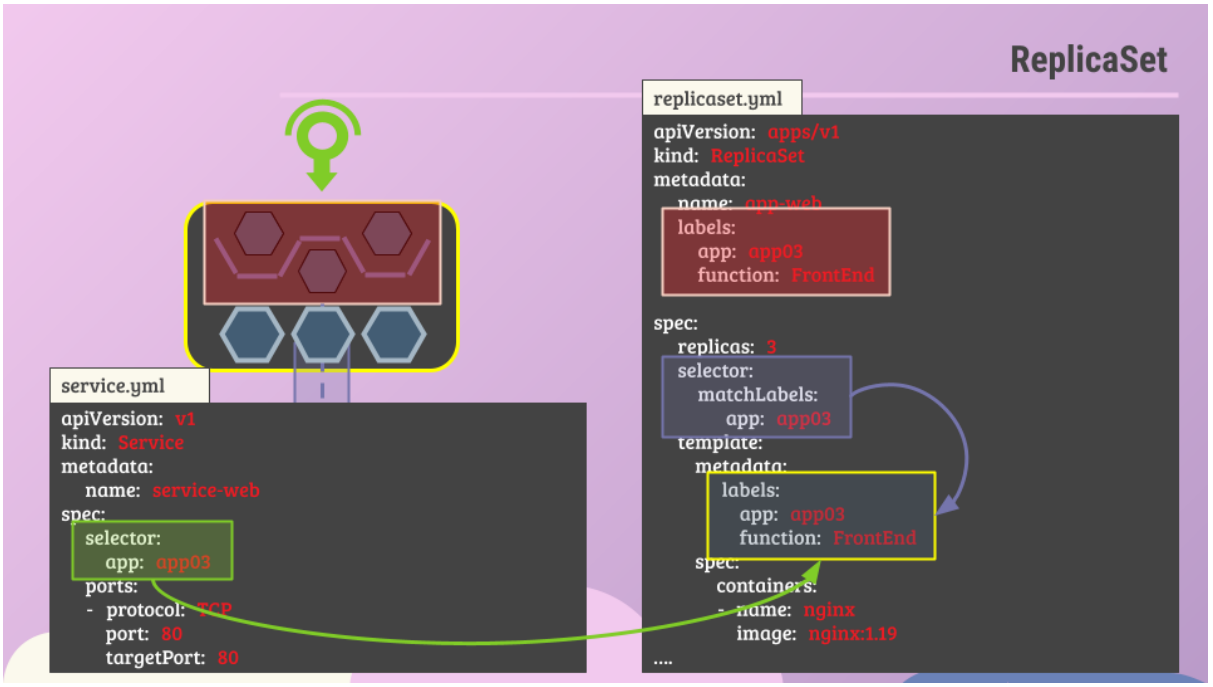
После создания POD для его выбора по меткам используй команду kubectl get pods вместе с параметром `--selector` и укажи в нем условие, например app=app07.

Это один из вариантов использования меток и селекторов. Объекты Kubernetes используют метки и селекторы для внутренней механики, чтобы соединить различные объекты.
Например, чтобы создать replicaset, состоящий из 3 разных PODs, мы сначала ставим labels в шаблоне определения PODs и самого replicaset, а потом используем selector в спецификации replicasert, чтобы знать, с какой группой порожденных PODs работать далее.

Здесь ты видишь метки, определенные в двух местах.
Здесь обычно новички совершают ошибку.

Метки, определенные в разделе шаблона (`template`), являются метками, настроенными на PODs. Метки, которые ты видишь вверху, являются метками самого replicaset.

На данный момент нас не особо беспокоят метки replicaset, потому что мы пытаемся обнаружить те PODs, которые создал replicaset.



Эти верхние метки, принадлежащие replicaset будут использоваться, если ты настроишь какой-либо другой объект, и ему придется искать сам этот replicaset в кластере.

Т.е. верхние метки это когда кто-то ищет этот replicaset, а нижние - когда replicaset ищет свои PODs.

Итак, чтобы подключить replicaset к его PODs мы настраиваем поле selector в соответствии с теми метками, что мы указами в шаблоне POD. Нужно чтобы хотя бы одно поле селектора соответствовало одной метке, определенной в POD.

Однако, если ты сомневаешься и чувствуешь, что могут быть другие PODs с такой же меткой, но посторонние, не имеющие отношение к этому replicaset, укажи несколько меток, чтобы гарантировать, что только правильные PODs будут обнаружены replicaset.

Если метки совпадают, то replicaset будет создан успешно.
Это работает одинаково для других объектов, например, для служб.

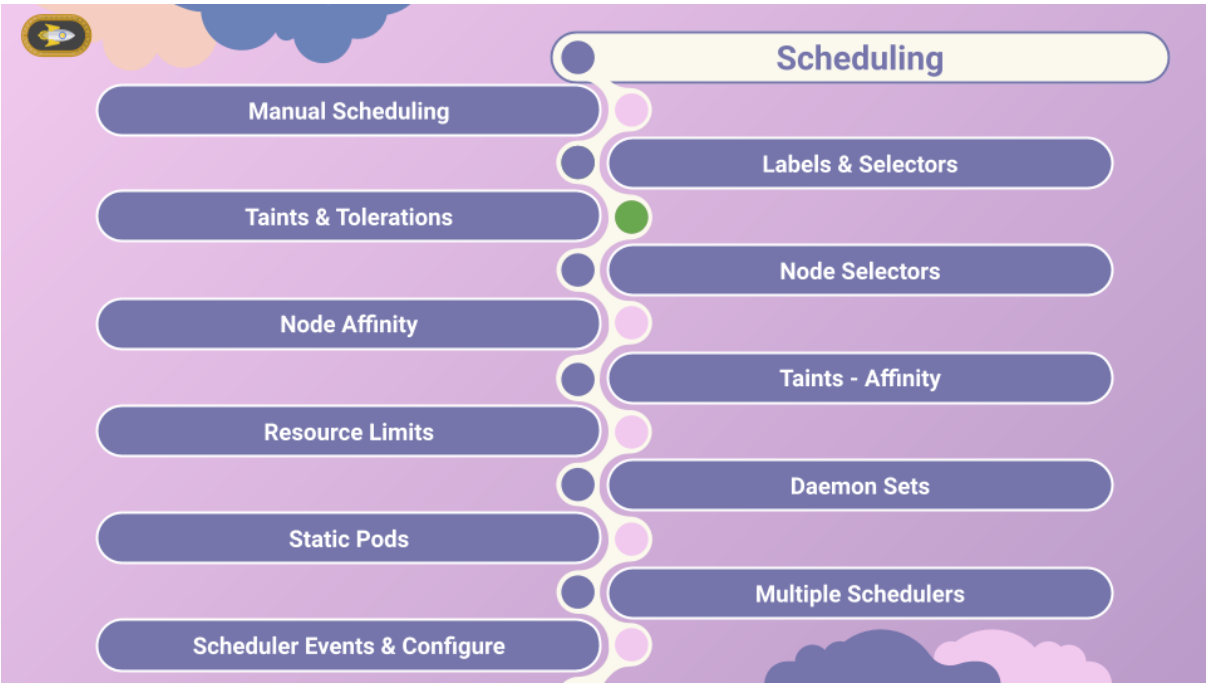
Когда служба создается, она использует селектор, указанный в файле определения службы, чтобы найти те свои PODs, куда она должна вести.
Служба ориентируется в этом поиске по меткам, установленным на PODs в файле определения replicaset.

И, наконец, давай посмотрим на аннотации. В то время как метки и селекторы используются для группировки и выбора объектов, аннотации используются для записи других деталей в информационных целях.

Например, технические сведения, такие как внутренне имя, информация о версии и т. д. Или это могут быть контактные данные, номера телефонов, идентификаторы электронной почты и т. д.
Еще это может быть использовано для каких-то целей интеграции, например средств мониторинга.

Ок, это все, что нужно для начала знать о labels, selectors и annotations.

Переходи в раздел упражнений и тренируй работу с метками и селекторами.



Привет и добро пожаловать.
В этой лекции мы обсудим отношения между PODs и Nodes в части того, как мы можем ограничить размещение определенных нагрузок на не предназначенных для этого узлах кластера.

По началу концепция `taints and tolerations` всех немного сбивает с толку. Это нормально для новичка.

Давай попробуем разобраться о чем все это используя простую аналогию. Пусть это будет комар, который хочет укусить китайского пчеловода. Прошу прощения, если обидел китайцев, если ты из этой прекрасной страны, считай что это вьетнамский пчеловод. Назовем его `Фей`.

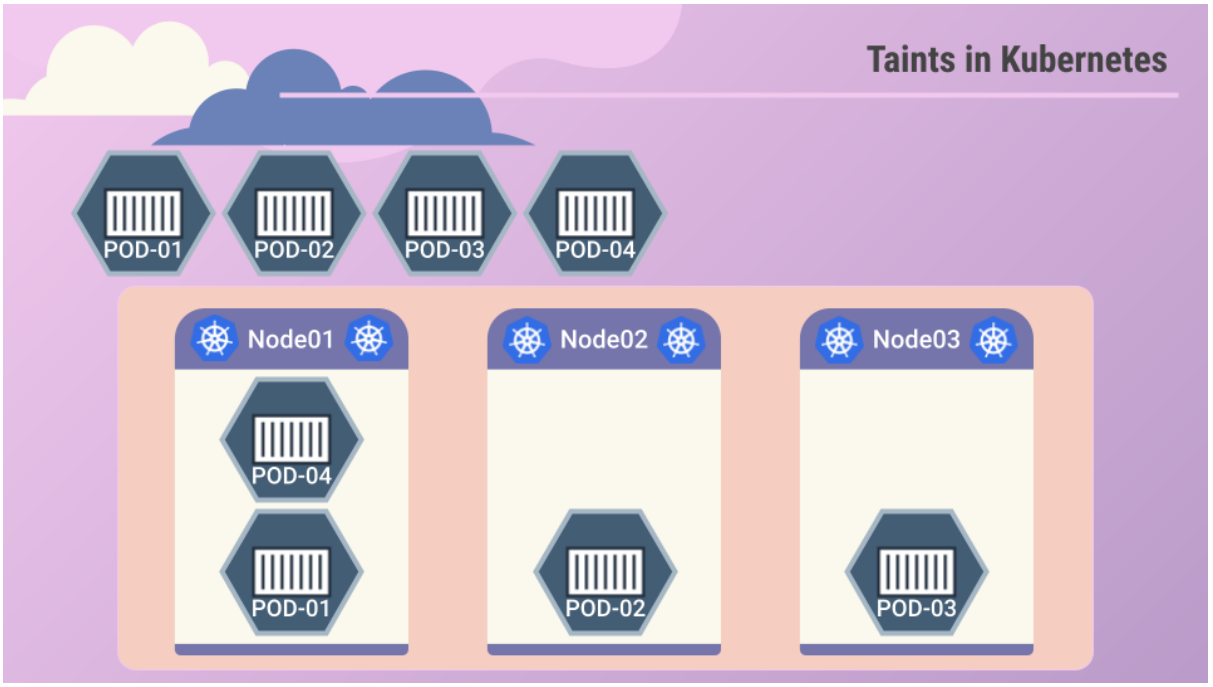
Итак, комары хотят укусить Фея. Но если мы опрыскаем его спреем-репеллентом, вокруг Фея образуется ароматное облако. Теперь окружен этой зловонной заразой. Можно сказать, что мы заразили (`taint`) его нашим спреем.

И это запах, которого этот вид комаров не может вынести. Иными словами комары нетолерантны (`intolerant`) к нашему зеленому спрею.

Однако у Фея есть пчелы, которые не боятся этой taint. Поэтому запросто летят к Фею поздороваться. Этот вид насекомого tolerant к нашей зеленой taint.

Итак, есть две вещи, которые решают, может ли букашка попасть на человека:

- taint на человеке.
- насколько насекомое tolerant к этой taint



В терминах Kubernetes node - это человек, а PODs - это букашки.

Ок, еще момент, taints and tolerations это внутренний механизм распределения PODs по нодам, он не имеет ничего общего с безопасностью или вторжением в кластер. Это используется, чтобы установить ограничения на назначение PODs.

Рассмотрим простой кластер с тремя рабочими нодами, которые называются node01, 02, 03.

У нас также есть несколько PODs, которые нужно развернуть в этих нодах.

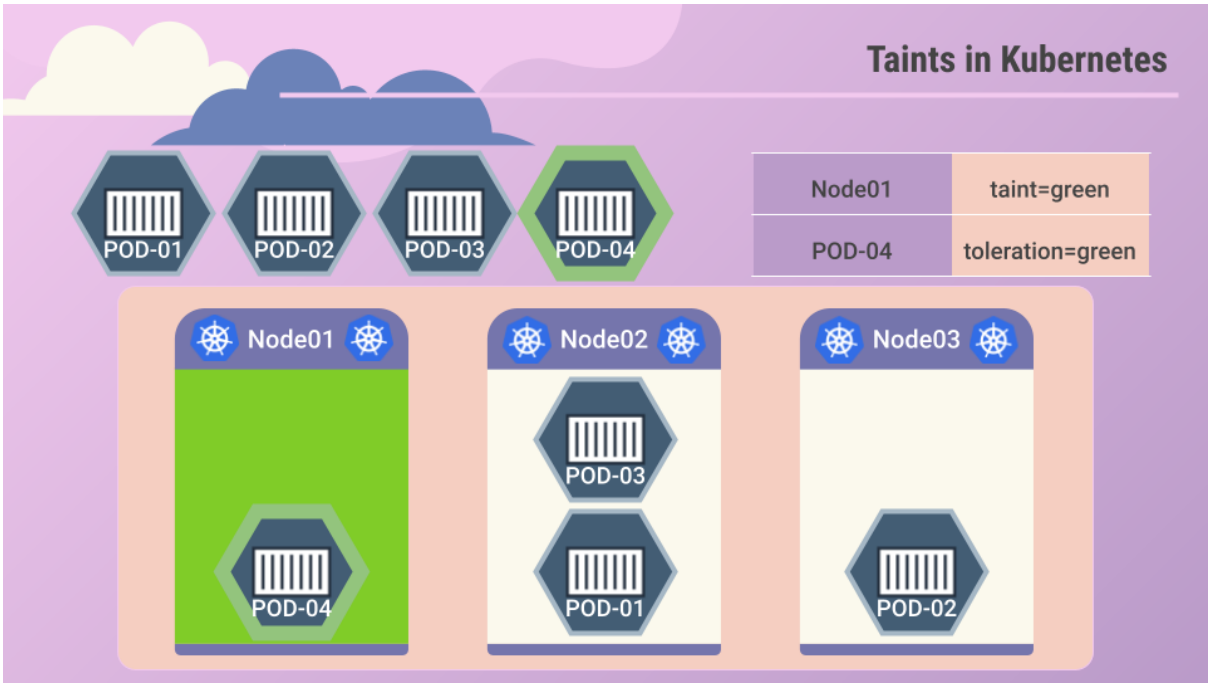
Назовем их POD-01, 02, 03, 04.

Когда мы просто создаем PODs, scheduler пытается разместить всех их на доступных рабочих узлах. На данный момент ограничений нет, и поэтому планировщик размещает эти части по всем узлам, чтобы сбалансировать их одинаково.

Теперь предположим, что у нас есть выделенные ресурсы на узле 1 для какого-то особого варианта использования или отдельного приложения.
Мы хотели бы, чтобы на узле 1 были размещены только те PODs, которые принадлежат этому приложению.

Сначала мы предотвращаем размещение всех PODs на узле, помещая на узел как зараженый, ставя на него зеленый taint. Назовем его green. Ты видишь это в таблице.

Как видишь не один из PODs будет допущен на зараженную ноду, поскольку они не могут терпеть такую заразу.



Таким образом, все эти PODs в данный момент будут размещены планировщиком на остальных нодах, но не на первой. Итак, taint=green решает половину нашей задачи, а именно нежелательные PODs не будут помещены на эту ноду.

Другая половина задачи - дать возможность нужным PODs разместиться на нужных нодах.

Для этого мы должны указать, какие PODs будут tolerant этой зеленой taint.

В нашем случае мы хотели бы разрешить размещение на этой node01 только POD-04, поэтому мы добавляем toleration=green к этому POD.

Теперь POD-04 терпим к зеленой заразе, и, когда планировщик попытается разместить этот POD на ноде 01, он будет успешно назначен на эту ноду.

При этом другие PODs не могут быть назначены на узел 01, поскольку на нем присутствует green taint.

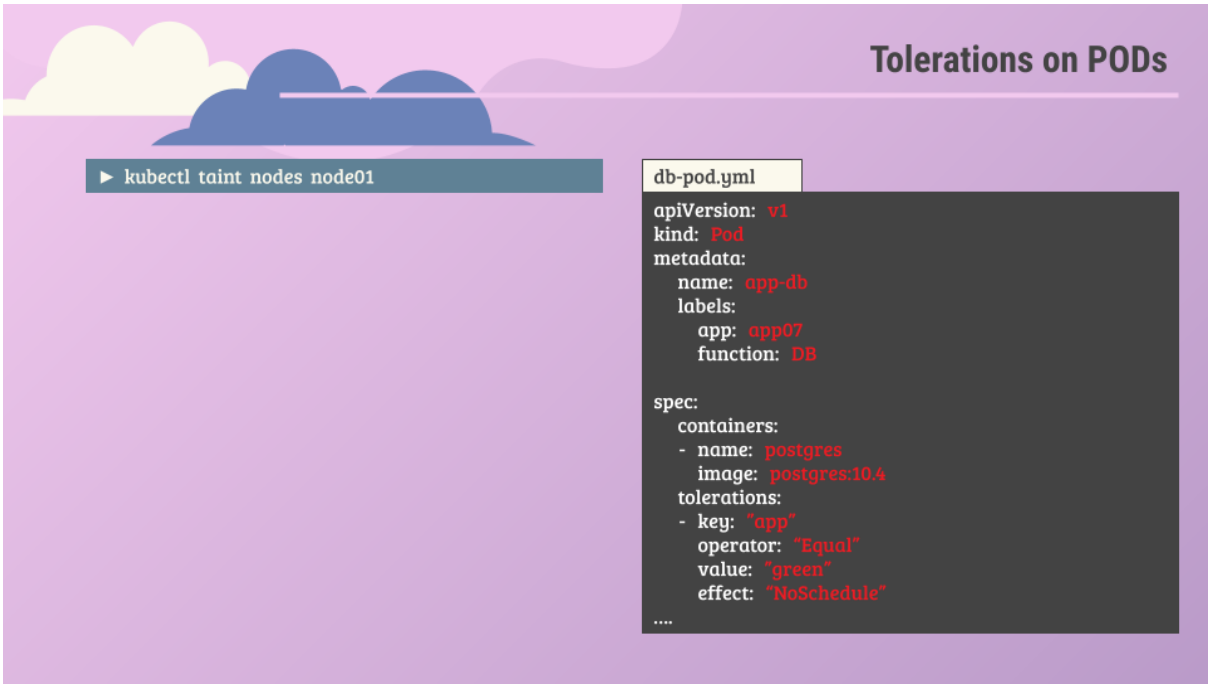
Давай посмотрим, как будут спланированы PODs.

Планировщик пытается разместить POD-01 на первой ноде, но видит там taint, а у POD-01 не устойчивости против этой заразы, тогда он подыскивает подходящую ноду, это будет 02.

Далее планировщик берет POD-02, проверяет ноду 01, она не подходит, и планировщик оценивает оставшиеся ноды. Очевидно, что это будет нода 03, т.к. она оказывается следующим свободным узлом. Тоже самое происходит и с POD-03, с той

разницей, что scheduler может назначить под с той же вероятностью что на 02, что на 03 ноду.

Наконец, планировщик пытается разместить POD-04 на узле 01, и, поскольку этот POD толерантен к заразе узла 01, он успешно назначается туда на выполнение.



Итак, как нам это сделать.

Используй команду kubectl taint nodes для установки заражения на этот узел.

Укажи имя узла, за которым следует сам taint. Он представляет собой пару ключ-значение. Далее, после двоеточия идет эффект заразы. Этот эффект заразы действует на те PODs, у которых не сопротивления к этому taint. В данный момент эти эффекты:

- NoSchedule
- PreferNoSchedule
- NoExecute

Таким образом, если ты хочешь выделить ноду 01 для зеленого приложения, тогда нужная тебе пара ключ-значений будет `app=green`, а эффект будет `NoSchedule`.

Это значит, что POD, который проходит процедуру назначения и не имеет toleration=green никогда не будет назначен на эту ноду. Это достигается эффектом NoSchedule.

Если бы там был эффект PreferNoSchedule, то Kubernetes попытается в процессе назначения подыскать этому POD другую, не эту ноду. Но если совсем не останется вариантов, то назначит на нее.

Третий эффект, NoExecute - самый жесткий. Он не только не дает создавать PODs без нужной толерантности, но и принудительно удалит все такие PODs, если они в данный момент присутствуют на ноде. Т.е. после того, как нода будет заражена, kubelet принудительно выселит все эти неподходящие PODs.

Давай взглянем на tolerations, как это все выглядит в файлах определений.

Сначала откроем файл определения POD и в разделе спецификации, создадим секцию tolerations, как ты здесь видишь.
В нем будут те же значения, которые использовались при создании taint для ноды, а именно:

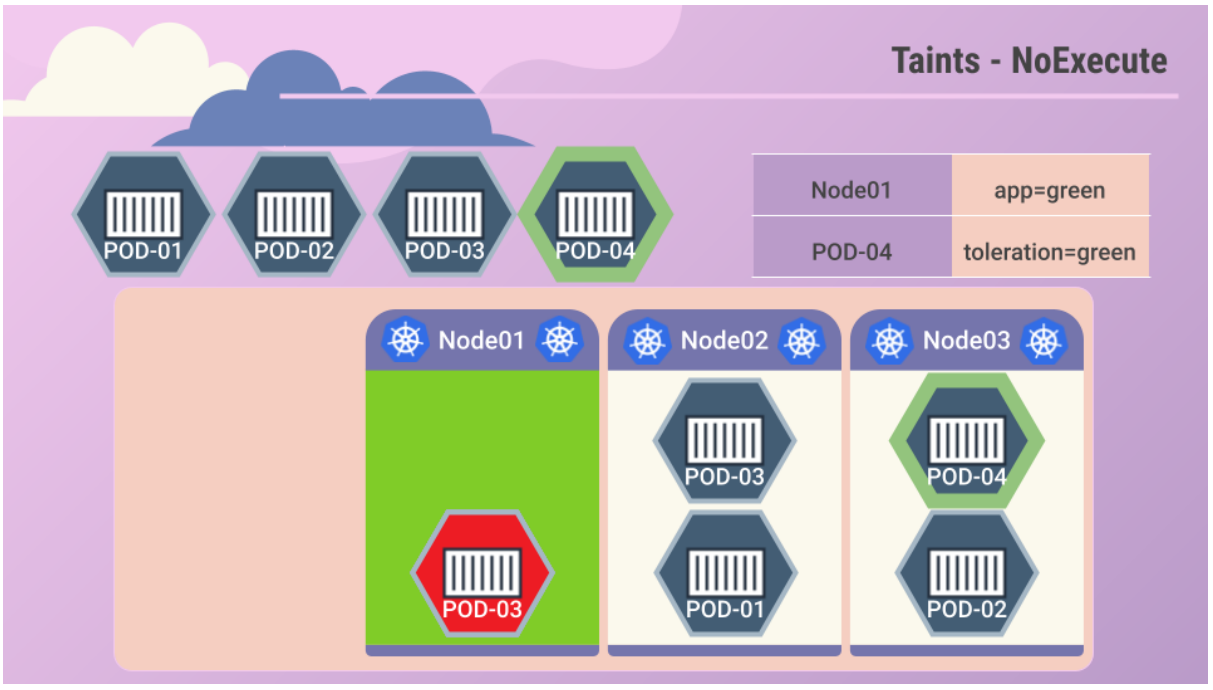
- `key` у нас выбран как `app`
- `operator` будет `Equal`
- `value` - `green`
- `effect` - `NoSchedule`
- .

Отметь для себя, что все этих поля необходимо оборачивать в двойные кавычки.

Ок. Когда PODs создаются или обновляются с новыми tolerations, они либо не планируются на какие-то узлы, либо выселяются из существующих узлов, в зависимости от установленного эффекта. Давай заглянем глубже в эффект NoExecute.

В этом примере у нас есть три узла, на которых выполняется некоторая рабочая нагрузка.

На данный момент у нас никаких taints and tolerations, поэтому PODs располагаются таким образом.



Далее мы решили выделить узел 01 для специального приложения, и поэтому добавили этой ноде taint, также добавили toleration для этого особого приложения, которое сейчас бежит в POD-04.

В этом случае, мы устанавливаем эффект заражения в NoExecute.
После того, как taint будет применет к узлу, POD-01 будет удален. Это значит, что его рабочая нагрузка на этом узле будет уничтожена, а сам POD останется лишь в виде записи в ETCD, ожидая размещения на какую-либо ноду.

POD-04 благополучно останется и продолжит свою работу на ноде, поскольку POD толерантен к эффекту этой заразы.

Теперь вернемся сценарию, когда у нас уже настроены taints and tolerations, но PODs еще не назначены на ноды.

Помни, что правила taints and tolerations предназначены только для ограничения узлов от принятия определенных PODs. Но этот механизм не гарантирует, что POD с нужной устойчивостью попадет на зараженную ноду. Он может попасть на ноду, которая по мнению scheduler наиболее подходящая в данный момент.

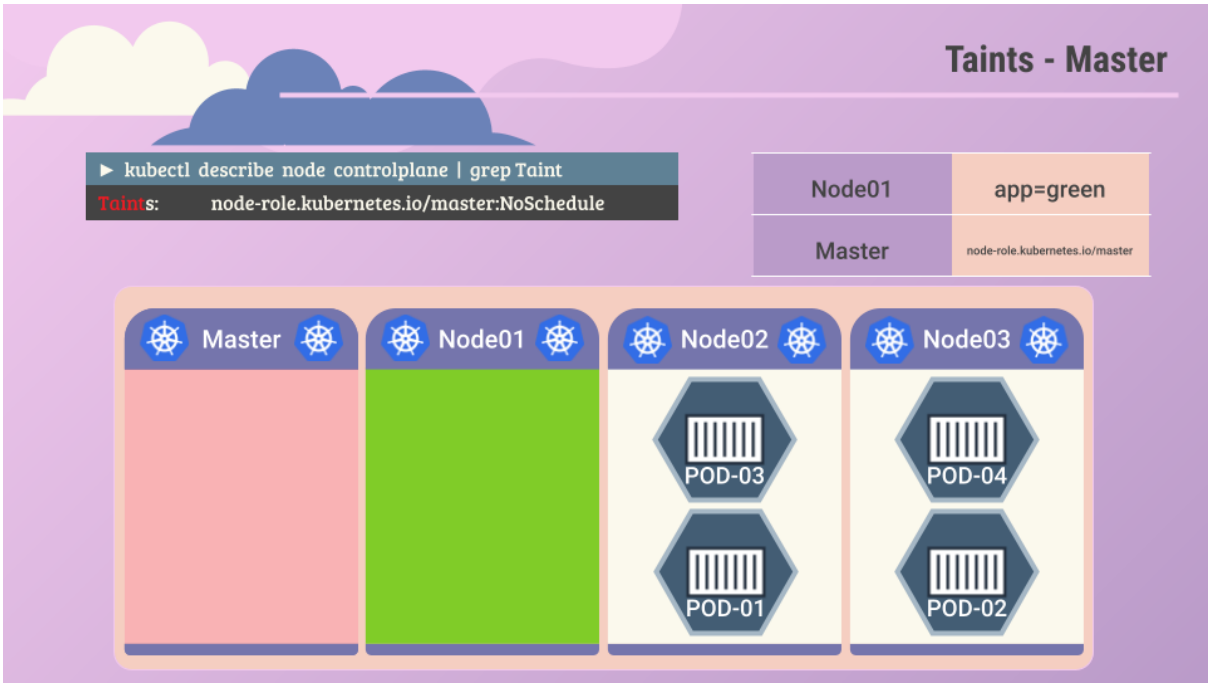
Поскольку к двум другим узлам не применяются taints and tolerations, POD-04 вполне может быть размещен на любой из этих нод.

Новички часто путаются, поэтому еще раз. Taints and tolerations не предписывают POD размещаться на этом конкретном узле. Вместо этого они указывает узлу принимать только PODs с определенными критериями.

Если ваше требование состоит в том, чтобы ограничить пространство PODs определенными нодами кластера, это достигается с помощью другого механизма. Эта концепция называется `node affinity`, и о ней мы поговорим уже скоро, в следующей лекции.

Но раз уж мы затронули эту тему, давай также взглянем на интересный факт.

До сих пор мы имели в виду только рабочие узлы.



Но у нас также есть главные узлы в кластере, мастера. Но технически они являются просто еще одной нодой, в которой размещена рабочая нагрузка controlplane. В остальном мастер имеет все возможности для размещения рабочих нагрузок на себе.

Я не уверен, но заметил ли ты, что планировщик не назначает никаких обычных рабочих нагрузок на главные узлы? Но это действительно так. Почему это происходит?

Когда мы только устанавливаем свой кластер Kubernetes, этот taint - master устанавливается на мастер-ноду автоматически. Это предотвращает планирование каких-либо PODs на нем, ну, разумеется, за исключением компонентов controlplane.

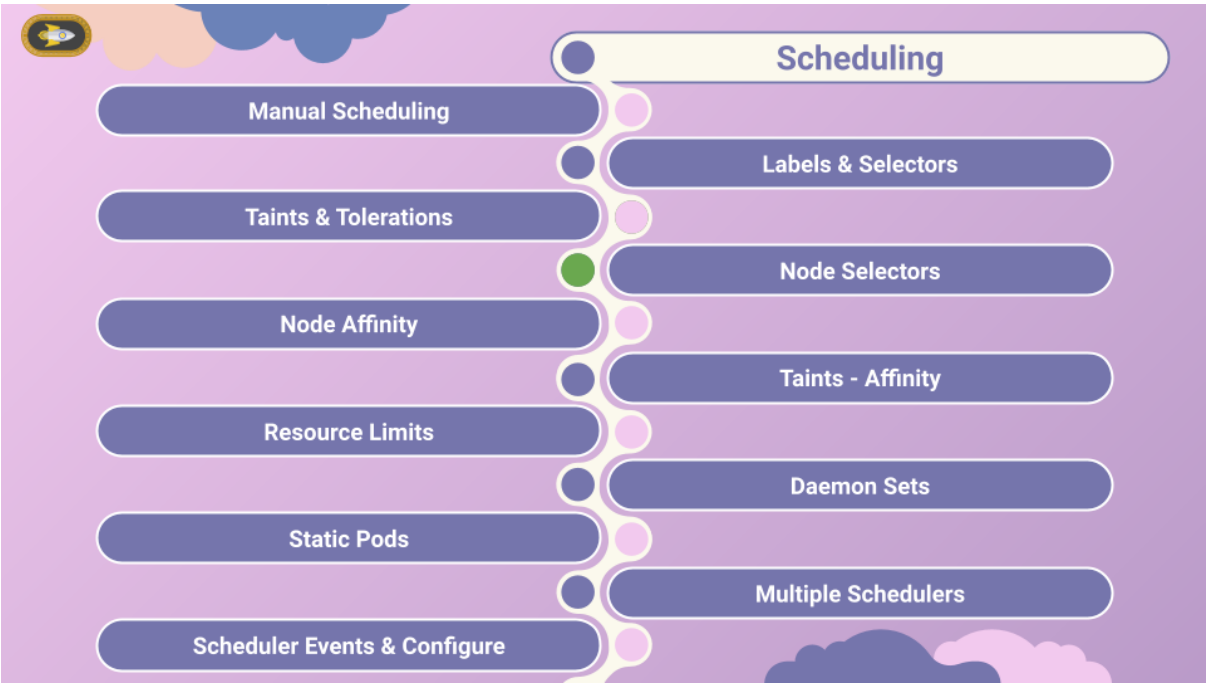
Ты можешь просмотреть этот taint, а также при необходимости изменить его поведение.

Однако лучшая практика - это не развертывать рабочие нагрузки приложений на главном сервере. Чтобы увидеть этот taint на мастере, сделай команду kubectl describe и имя мастер-ноды.

Там в разделе Taints ты увидишь этот taint с эффектом NoSchedule, который и запрещает назначать PODs на мастер.

Ну вот и все в этой лекции. Перейди в раздел упражнений и потренируй работу с taints and tolerations. Я жду тебя в следующей.


Lab
Taints and
Tolerations



Привет и добро пожаловать на лекцию. В этой лекции мы поговорим о node selectors в Kubernetes.

Начнем с простого примера.

У тебя есть кластер с тремя рабочими нодами, две из которых представляют собой меньшие узлы с меньшими аппаратными ресурсами, а одна - более мощный и вместительный узел.

В этом кластере выполняются различные виды рабочих нагрузок. Ты хотел бы выделить рабочие нагрузки по обработке данных, которые требуют более высокой мощности, на этот крупный узел, поскольку это единственная нода, у которого не будут исчерпаны ресурсы в случае, если задание потребует дополнительных мощностей.

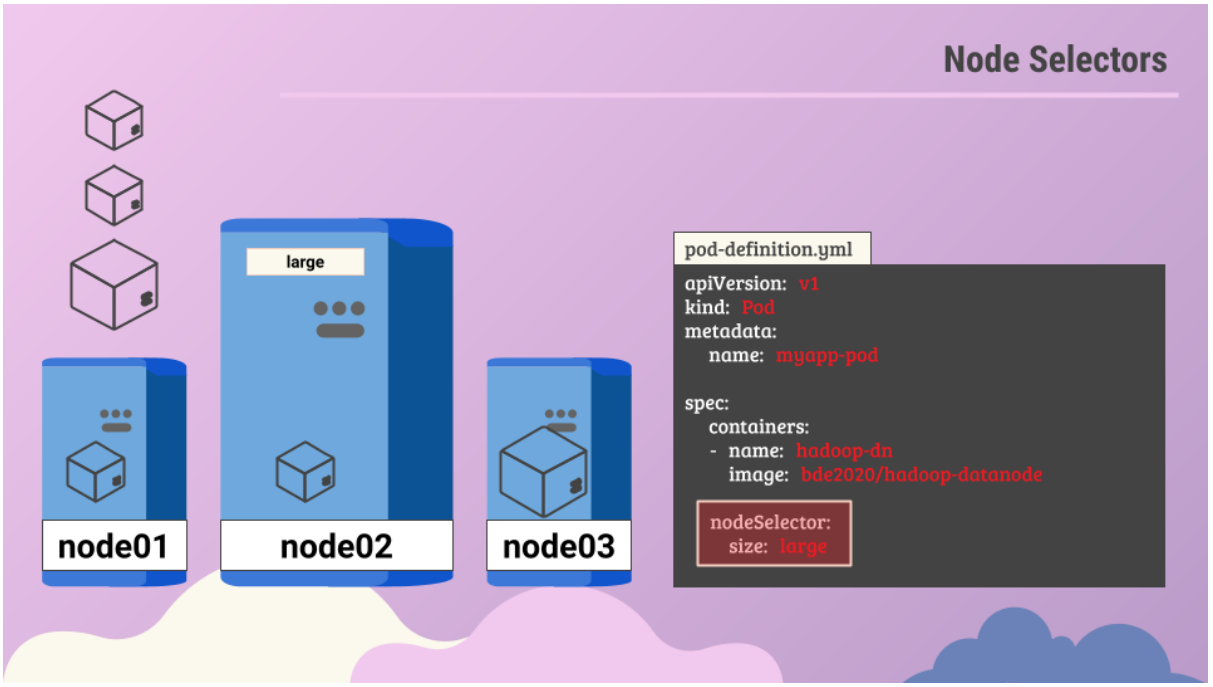
Однако в текущих настройках по умолчанию любые PODs будут назначаться на любые ноды.

Таким образом, этот толстый POD вполне может оказаться на первой или третьей ноде, что нежелательно. Чтобы решить эту проблему, мы можем установить ограничение для PODs, чтобы они появлялись только на определенных узлах.

Есть два способа сделать это.

Первый - это использование node selectors, что является более простым и легким методом.

Для этого мы смотрим на файл определения POD, который создали ранее.



В этом файле мы указали прожорливый образ для создания среды обработки больших данных, и нам понятно, что в кластере эта нагрузка сможет работать только на нашей самой мощной ноде, другие ее просто не потянут.

Мы добавляем новое свойство под названием nodeSelector в раздел спецификации и указываем размер ноды (size) как большой (large).

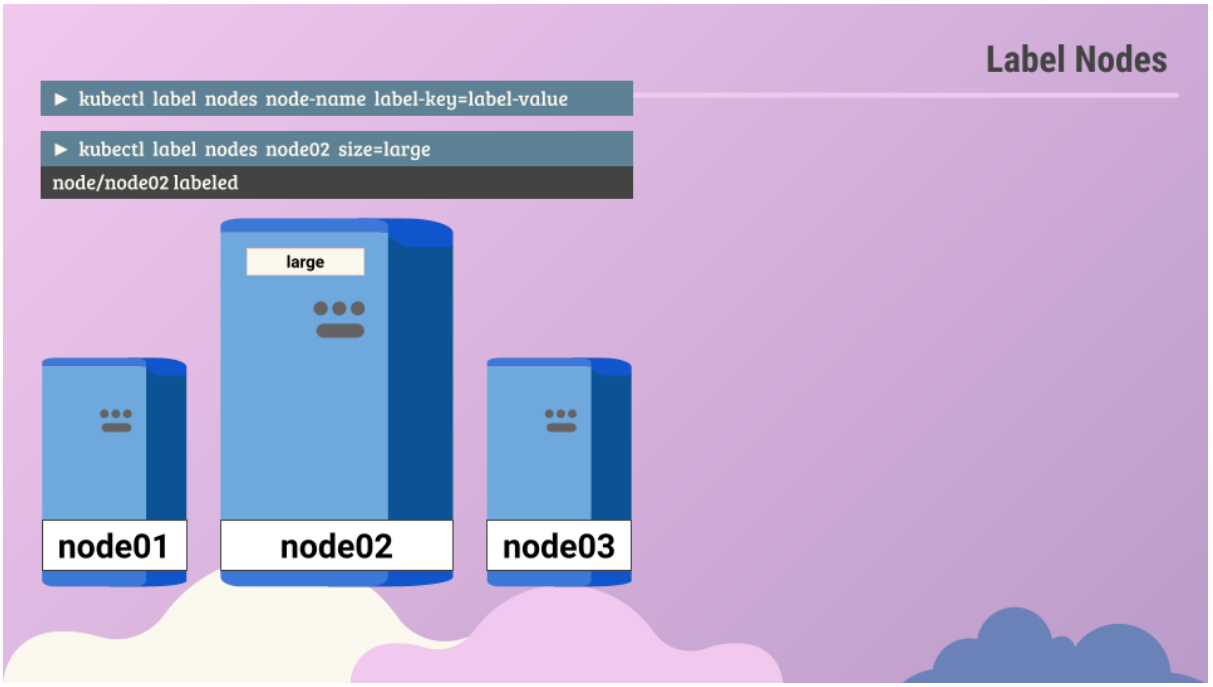
Но подожди минутку.
Откуда взялся этот large size и как Kubernetes узнает, какой узел является большим?

Пара ключ-значение `size-large` на самом деле являются меткой, добавленной к узлам. Эти метки будет использовать планировщик, чтобы сопоставить и определить правильную ноду для размещения PODs.

Ты уже познакомился с метками и селекторами для PODs, это та тема, которой мы не раз касались на протяжении этого курса Kubernetes. Например, обсуждая механику replicasets и deployments.

Итак, перед созданием этого POD мы должны сначала пометить свои узлы.

Давай посмотрим, как мы можем установить labels на узлы.



Чтобы пометить ноду используй команду kubectl label nodes, за которой следует имя узла и метка в формате пары ключ-значение.

В этом случае я хочу поставить метку на node02, пишу kubectl label node node02 size=large. Т.о. Kubernetes ничего не знает о размере ноды, но он сможет проанализировать значение метки size.

Теперь, когда мы пометили узел, мы можем вернуться к созданию POD, но на этот раз уже с установленным nodeSelector `size` в значение `large`.

Когда POD создается, он помещается на узел 02, как мы и задумали.

Ок, эти селекторы нод прекрасно служат нашим целям, но у него есть ограничения.

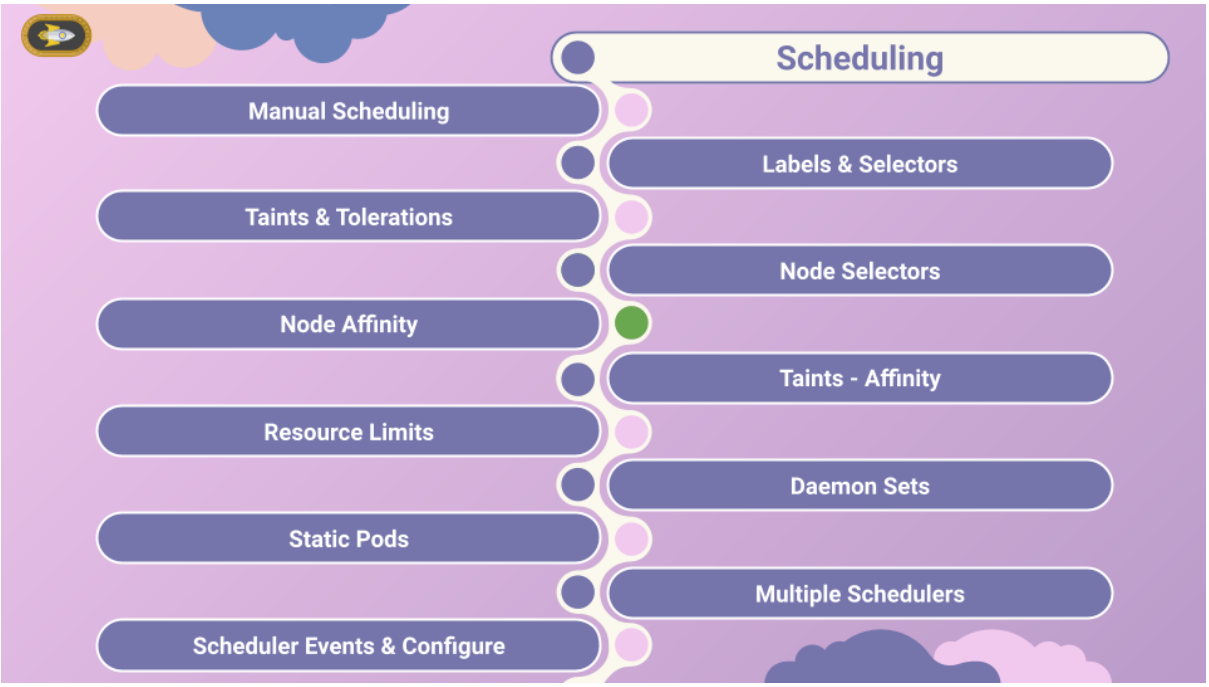
Мы использовали одну метку и селектор для достижения нашей цели.

Но что, если наше требование будет намного сложнее?

Например, мы хотели бы сказать что-то вроде: `размести POD на большом или среднем узле` или что-то типа: `направь эти PODs на любые, но не малые узлы`.

Мы не сможем этого добиться, используя nodeSelectors. Для этого в Kubernetes есть механизмы nodeAffinity и anti-affinity, и мы поговорим об этом дальше.

А с этой темой у нас все, жду тебя в следующей лекции.



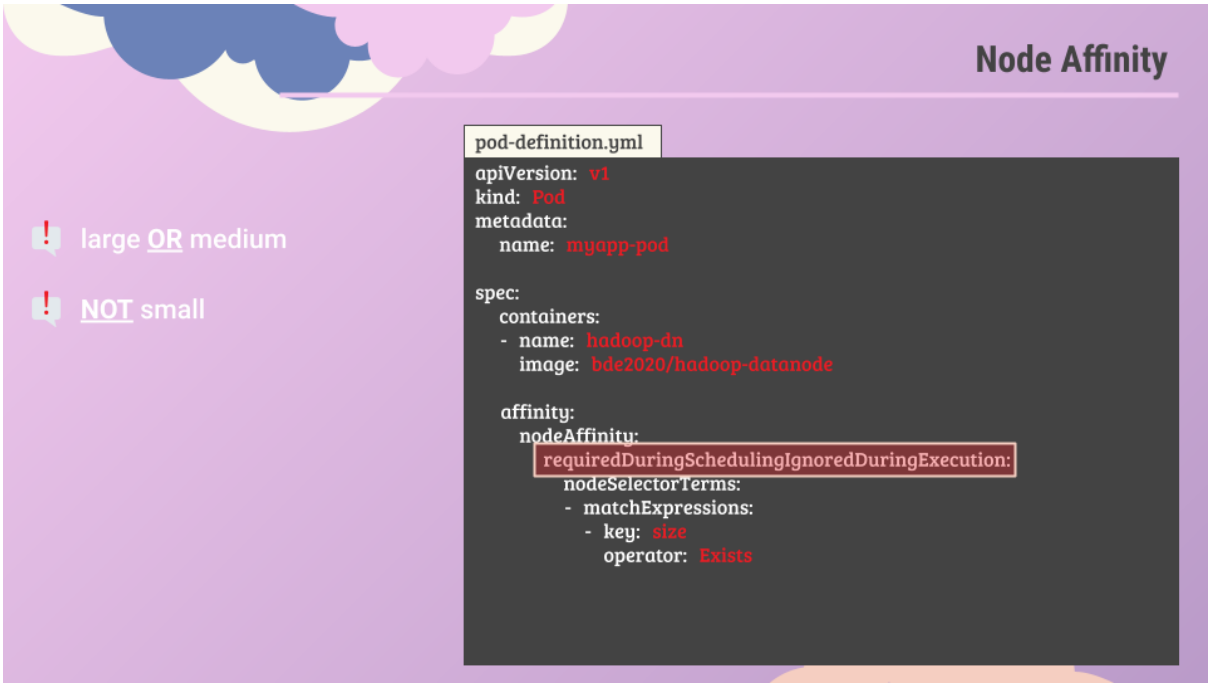
Привет и добро пожаловать на лекцию. В этой лекции мы поговорим об особенностях node affinity в Kubernetes.

Основная цель node affinity - обеспечить размещение PODs на определенных нодах.

В этом случае нам нужна гарантия, что этот ресурсно затратный POD с приложением обработки big data попадет на выделенную ему ноду. В предыдущей лекции мы сделали это легко, используя node selectors. Еще мы упомянули об ограничениях этого механизма, а именно, что мы не можем реализовать расширенные выражения, такие как `OR` и `NOT` с помощью node selectors.

Функция node affinity предоставляет нам расширенные возможности для лимитирования размещений PODs на определенных узлах. Но этот механизм обладает как большой мощью так и высокой сложностью.

Итак, спецификация обычного node selector, реализованная с помощью node affinity теперь будет выглядеть так. Жутковато, не так ли? При этом оба варианта кода делают одно и тоже - заставляют назначать POD на ноду с меткой large.



Давай посмотрим на это внимательнее. В секции со спецификацией у нас появилось свойство affinity, а в ней определено свойство nodeAffinity. Далее идет длинное предложение, которое гласит что-то вроде `требуетсяВовремяНазначенияИгнорируетсяВовремяИсполнения`. Ну тут вобщем понятно, оно само себя описывает. А затем у нас идут условия для node selector, которые представляют собой массив, в котором будут наши пары ключ-значение.

Кроме пары ключ-значение здесь еще находится поле, которое обеспечивает гибкую логику. Это поле `operator`. В данный момент значение оператора `In`. Т.о. выстраивается логика, по которой будет искаться метка size на нодах-кандидатах и проверяться на вхождение в массив значений values.
Сейчас там только large. Но если ты решишь, что этот POD сможет разместиться на средней ноде, то можешь добавить туда значение medium. После этого ноды с такой меткой смогут тоже участвовать в планировании.

Как видишь оператор in дает нам логику `или`, т.е. POD будет размещен на узле, метка size которого имеет любое значение из списка, а это или large или medium.
Если ты решишь зайти с другой стороны и определить ноды, на которые не стоит размещать определенные PODs, nodeAffinity тоже может тебе с этим помочь.

Например, ты не хочешь размещать этот POD ноде с тегом small, а на остальных можно. В этом случае тебе поможет оператор `notin`, который реализует логику `не`. Таким образом POD будет назначаться на те узлы, которые не помечены как size=small.

Мы установили метки размера только на большую и среднюю ноды, а на всех остальных метки просто не установлены. Это дает нам возможность применить еще один оператор - `exists`.

Ведь по сути, нам действительно не нужно проверять значение метки, если мы уверены, что не устанавливали ее для неподходящих машин. Мы проверим лишь ее наличие, как видишь здесь. Оператор exists даст нам тот же результат.
Еще этому оператору не требуется значение, он ничего не сравнивает, поэтому поле values здесь отсутствует.

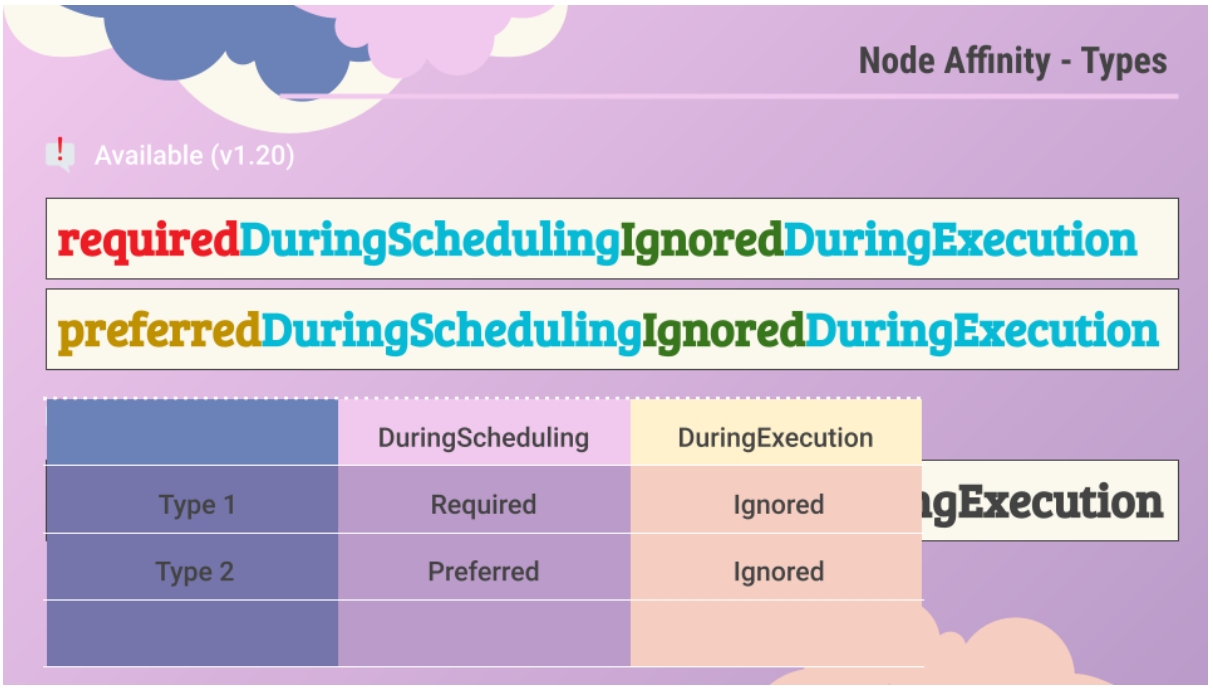
Есть также ряд других операторов, это самые ходовые. Подробнее ищи на страницах документации.

Ок, теперь мы все это понимаем, что нам станет легче жить, определяя в кластере некоторые правила affinity.
Эти правила будут работать при создании PODs, и PODs будут назначаться на нужные узлы.

Но что, если выражение nodeAffinity не будет соответствовать ни одному узлу в кластере?
Например, если в нем нет узлов с меткой size. Или вдруг случится, если кто-то изменит метки на нодах? Остануться ли PODs с этой affinity на нодах после этого?

На эти вопросы нам отвечает то длинное предложение, находящееся сразу после свойства nodeAffinity. Это предложение описывает поведение scheduler, а условия, которые определены в этом предложении триггером.

Т.е. триггер - метка size существует на ноде, а поведение - принимать выражение только во время назначения POD.



Согласно этому руководству и действует планировщик на протяжении всего жизненного цикла POD.
Давай разберем подробнее эту длинную запись.

В настоящее время доступны два типа nodeAffinity:
- requiredDuringSchedulingIgnoredDuringExecution
- preferredDuringSchedulingIgnoredDuringExecution

также планируется тип:
- requiredDuringSchedulingRequiredDuringExecution

Эти типы описывают состояния в жизненном цикле POD, относящиеся к node affinity, а именно промежутки во время планирования и во время выполнения.

Во время планирования (DuringScheduling) - состояние, при котором POD не существует и создается впервые.

Когда POD был создан, наши правила node affinity помещают его на нужную ноду.
Что делать, если узлы с совпадающими метками недоступны?
Например, мы забыли пометить узел как большой.

Именно здесь начинает работать тип используемой affinity. Если мы выберем тип required, который является первым, планировщик потребует, чтобы POD был размещен на узле, указанными в условиях nodAffinity у POD.
Если он не может найти его, POD не будет запланирован. Этот тип используется в тех случаях, когда размещение POD имеет решающее значение. Если соответствующий узел не существует, POD не будет назначен.

Но предположим, что размещение POD менее важно, чем выполнение самой рабочей нагрузки. В этом случае мы можем установить тип как предпочтительный (preferred). В тех случаях, когда соответствующий узел не найден, планировщик просто проигнорирует правила nodeAffinity и назначит POD любую доступную ноду.

Т.е. это способ сказать планировщику, что он изо всех сил старается разместить приложение на соответствующем узле, но если действительно не может сделать это, просто помести его куда-нибудь.

Вторая часть свойства или другое состояние или дугой промежуток в жизни POD - во время выполнения.

DuringExecution - это состояние, в котором POD был запущен, но в среду были внесены изменения, влияющие на node affinity, например изменение метки узла.

Скажем, администратор удалил с узла метку, о которой мы говорили ранее, что размер равен большому. Что будет с PODs, работающими на ноде?

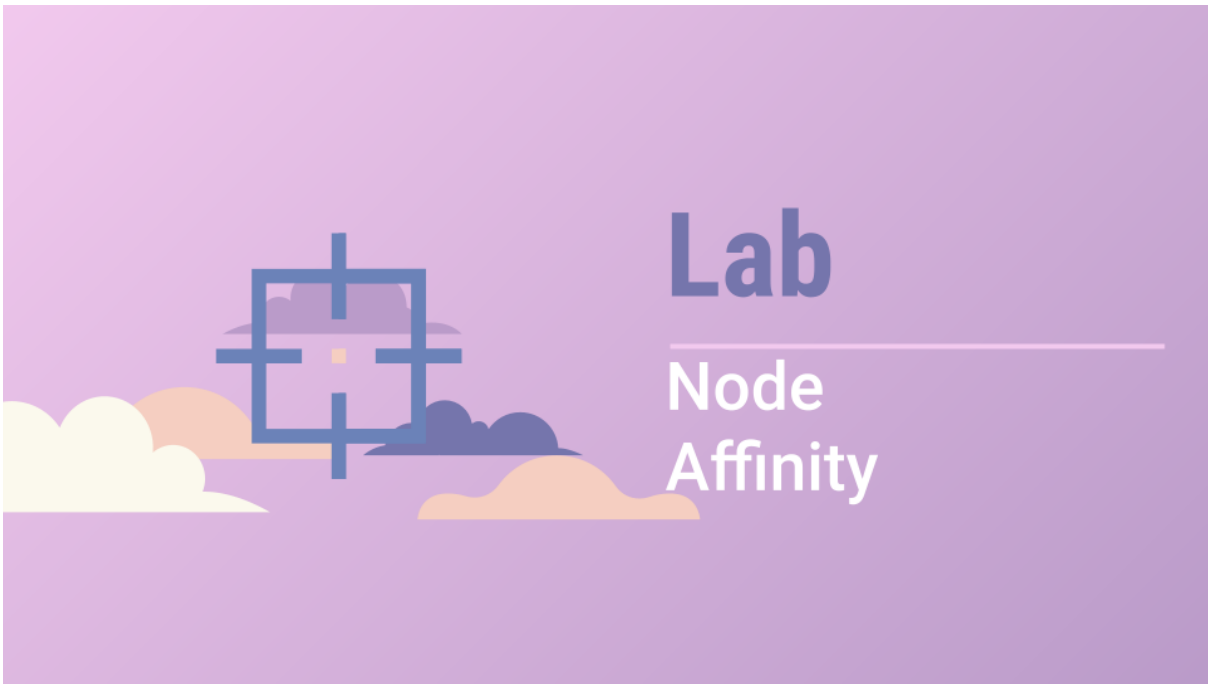
Как ты можешь видеть, для двух доступных сегодня типов привязки узлов это значение также игнорируется. Это значит, что PODs будут продолжать работать, и любые изменения node affinity не повлияют на них после того, как они были запланированы.

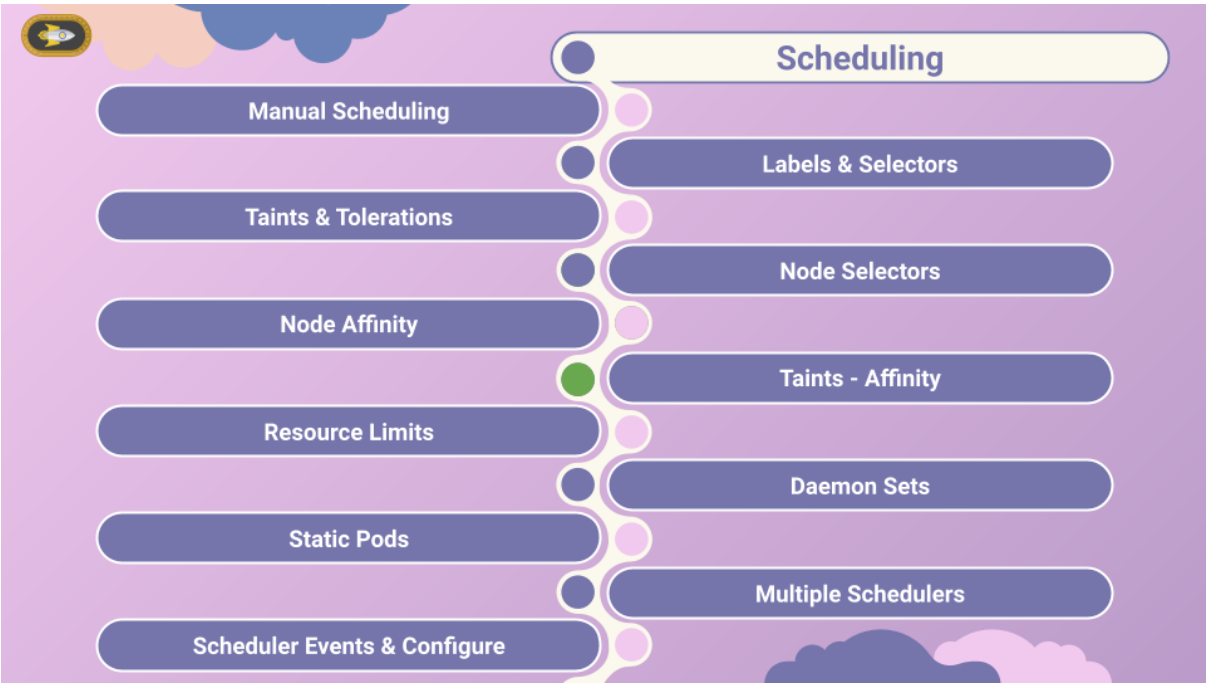
Новый тип, ожидаемый в будущем, имеет различие только во время фазы выполнения, введена новая опция, называемая required во время выполнения, которая вытесняет любые PODs, работающие на нодах, которые не соответствуют правилам node affinity.

В предыдущем примере POD, работающий на большом узле, будет исключен или завершен, если с узла будет удалена метка size=large.

Как ты видишь, механизмы taints and toleration и node affinity похожи. Мы сравним их в следующей лекции.

А в этой уже все! Перейди в упражнения и попрактикуй работу с правилами node affinity. Увидимся в следующей.





Привет и добро пожаловать на эту лекцию.
Теперь, когда мы узнали taints and tolerations и node affinity,
давай свяжем вместе эти две концепции с помощью мысленного упражнения.

У нас есть три узла и три PODs, каждый в трех цветах: синий, зеленый и красный.
Конечная цель - разместить синий в синем узле, красный в красном узле, а зеленый в зеленом.

Мы используем один и тот же кластер Kubernetes с другими группами пользователей. В кластере есть другие PODs, а также другие узлы.

Мы не хотим, чтобы на наших узлах размещались какие-то другие PODs.
Еще мы хотим, чтобы наши поды размещались на своих узлах.

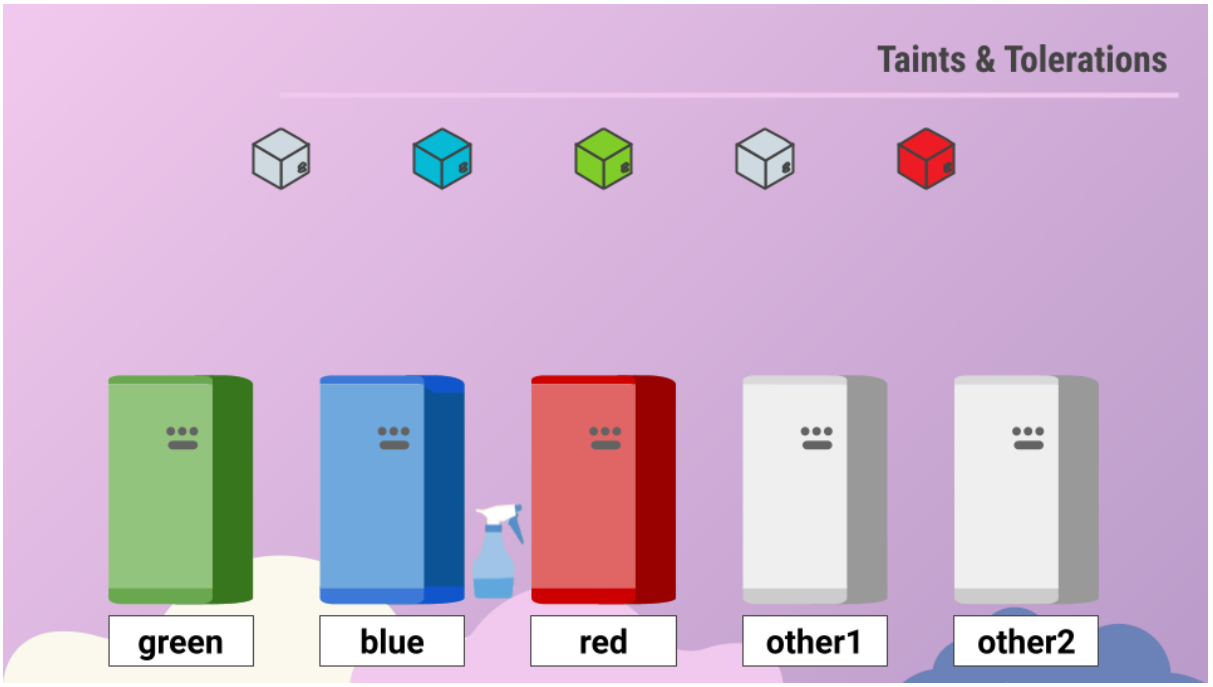
Давай сначала попробуем решить эту проблему, используя taints and tolerations.

Мы применяем taint к узлам, отмечая их их цветами: зеленым, синим и красным.

Затем мы устанавливаем tolerations на PODs, чтобы они могли быть терпимыми к соответствующим цветам.

Когда PODs созданы, узлы гарантируют, что они принимают только PODs с правильным допуском.

Итак зеленый POD назначился на зеленую ноду, а синий - на синюю. А вот с красным все вышло не очень.

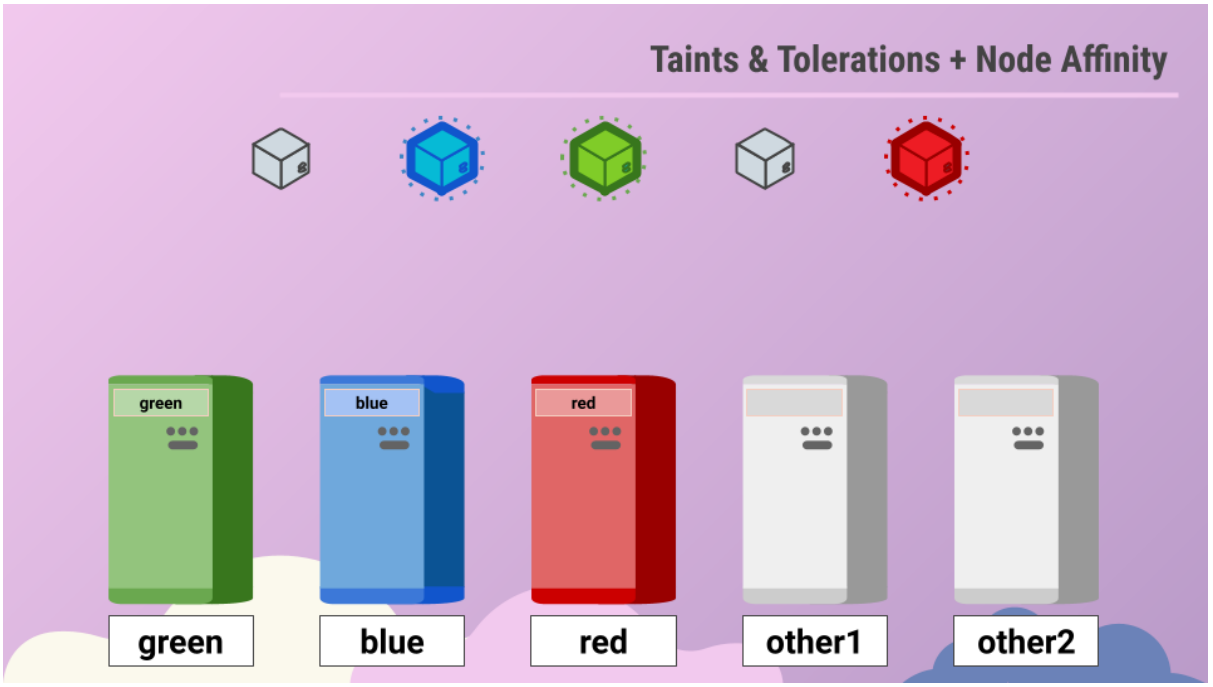


Как ты помнишь, taints and tolerations не гарантируют, что PODs выберут именно эти узлы, поэтому красный POD оказывается на одном из других узлов, для которых не задано никаких taints.

Нас это не устраивает.

Попробуем решить ту же проблему, используя node affinity.
Используя node affinity, мы сначала помечаем узлы их соответствующими цветами: зеленым, синим и красным. Затем устанавливаем селекторы узлов на PODs, чтобы связать их с нодами.
Таким образом, PODs попадают в нужные nodes.

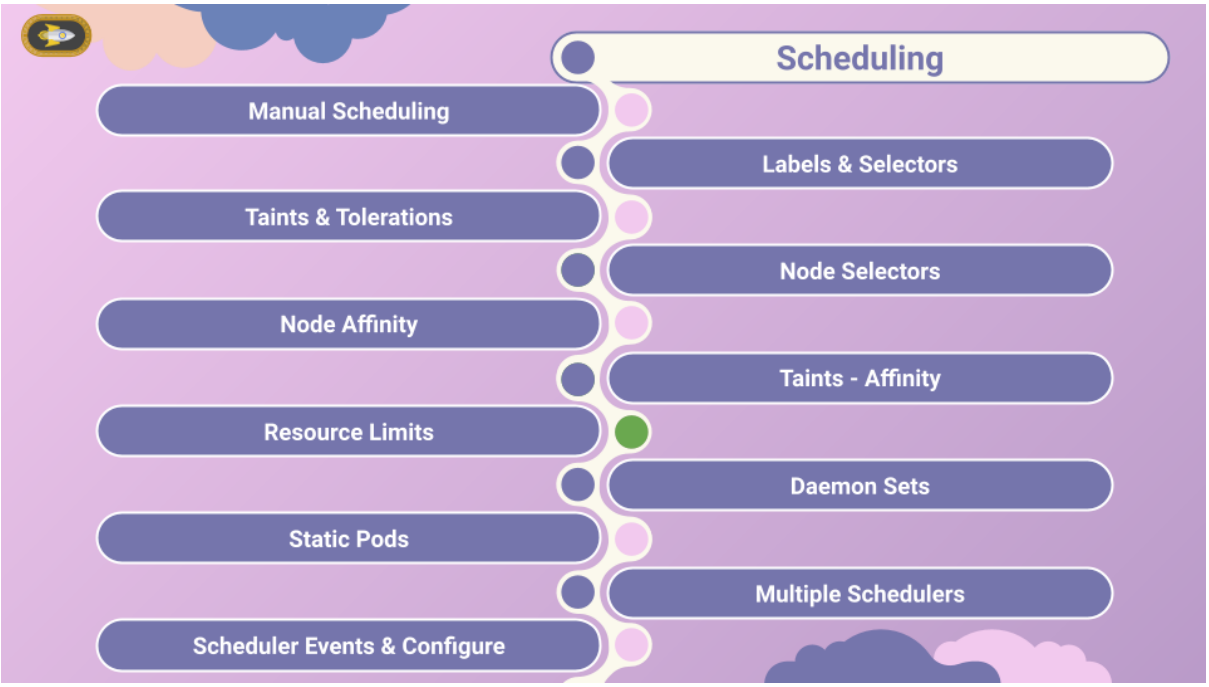
Однако это не гарантирует, что на этих узлах не будут размещены другие нежелательные PODs.
В этом случае есть вероятность, что один из других PODs может оказаться на наших нодах.



Это тоже нас не устраивает. Таким образом, комбинация taints and tolerations и node affinity может использоваться вместе, чтобы полностью выделить узлы для определенных нагрузок.

Сначала мы используем taints and tolerations, чтобы предотвратить размещение других PODs на наших узлах.
Затем мы используем node affinity, чтобы наши PODs размещались на своих узлах.
Получилось быстро и эффективно.

Ок, в этой лекции все. Увидимся в следующей!



Привет и добро пожаловать на лекцию. Здесь мы разберем выделение ресурсов в Kubernetes.

Давай посмотрим на кластер Kubernetes с тремя узлами. Каждый узел имеет набор доступных ресурсов центрального процессора, памяти и диска. Каждый POD потребляет набор ресурсов.

Чтобы нам было проще, думай об этом, как об игре в тетрис.

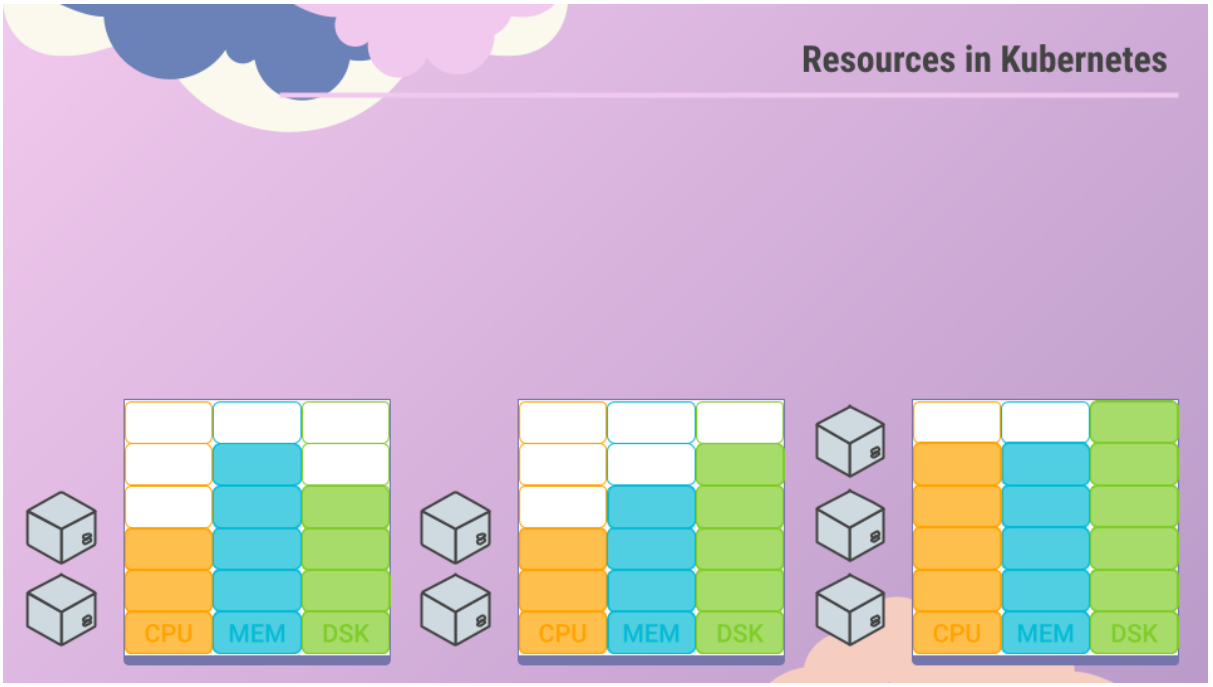
В этом случае одну единицу CPU, две памяти и немного места на диске. Каждый раз, когда POD размещается на узле, он потребляет ресурсы, доступные этому узлу.

Как мы уже обсуждали ранее, Kubernetes scheduler решает, к какому узлу переходит POD. Планировщик учитывает количество ресурсов, необходимых POD и их количество, доступное на узлах.
В этом случае планировщик назначает новый POD на узел 2.

Если у узла недостаточно ресурсов, планировщик избегает размещения POD на этом узле. Вместо этого помещает POD на тот, где в данный момент есть достаточно ресурсов.

В случае, если у всех нод кластера недостаточно ресурсов, Kubernetes сдерживает планирование POD, и мы увидим, что POD перешел в состоянии ожидания.
Если ты посмотришь на события, то увидишь причину - недостаточно процессоров.

Таким образом, задача scheduler также состоит в том, чтобы PODs не было тесно на нодах. Он рационально подходит к назначению, учитывая требования приложений и возможности нод.



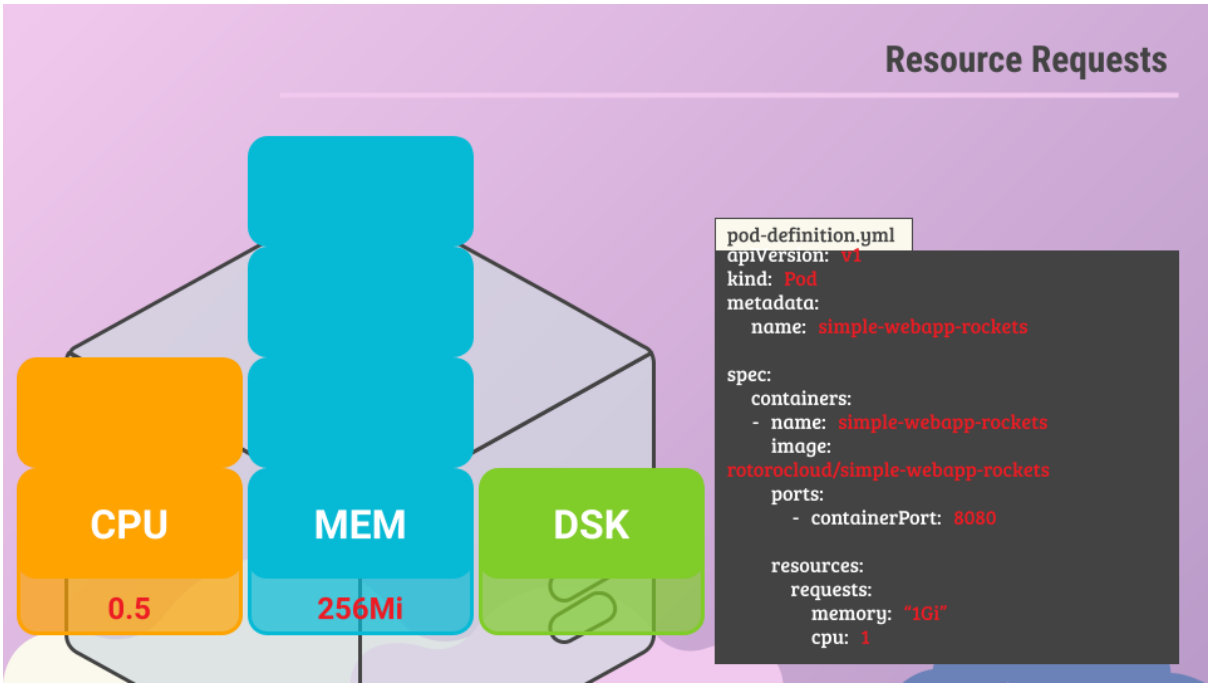
Ок, давайте теперь сосредоточимся на том, что это за требования к ресурсам для каждого POD. И что это за блоки, из которых складывается наш тетрис и каковы их значения?

По умолчанию Kubernetes предполагает, что POD или контейнер внутри POD требует .5 CPU и 256 мебибайтов памяти.
Это известно как запрос ресурса для контейнера - минимальный объем ЦП и памяти, которые запрашивает контейнер для работы.

Когда планировщик пытается разместить POD на узле он использует эти числа для идентификации узла, имеющего достаточное количество доступных ресурсов.

Теперь, если мы знаем что нашему приложению потребуется больше, чем по умолчанию, мы можем изменить эти значения, указав их в своем файле определения POD.

Добавим раздел с названием resources в секцию описания контейнера, в котором в поле запросы (requests) укажем новые значения для использования памяти и процессора.
В этом случае я установил 1 Гибибайт памяти и 1 количество виртуальных CPU. Так что же на самом деле означает выражение 1 CPU?

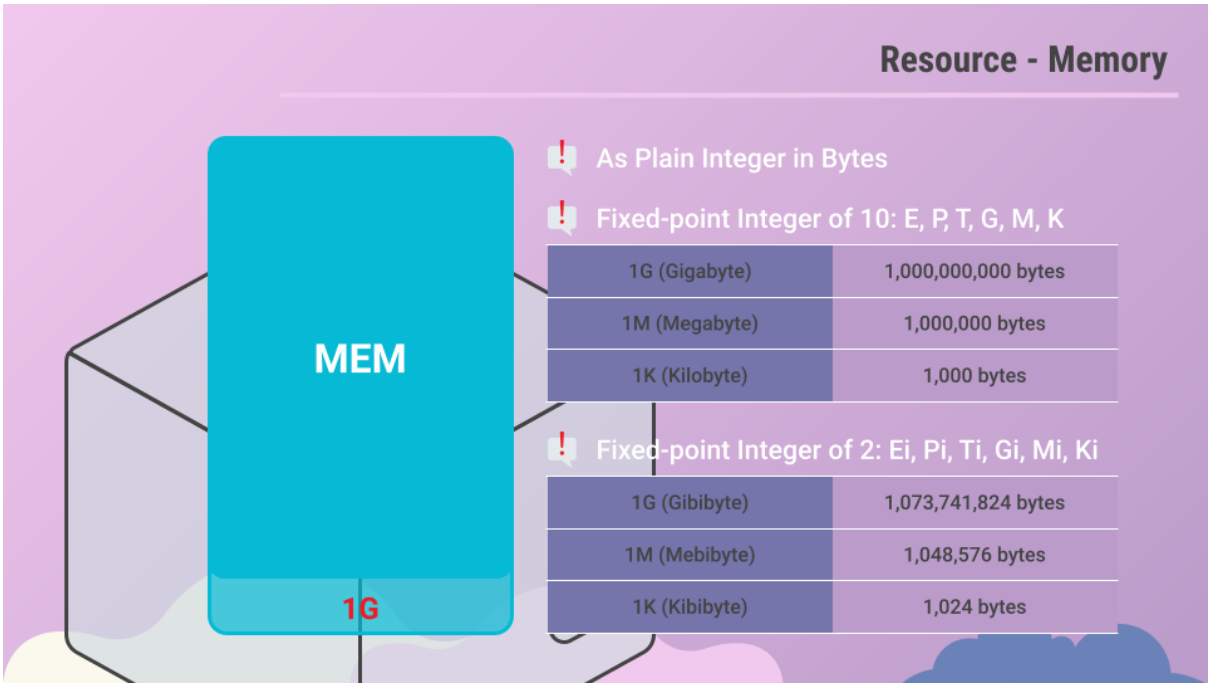


Эти блоки используются для наглядной иллюстрации. Значения CPU не обязательно должны меняться с шагом 0.5 пять баллов. Ты можешь указать любое значение от 0.1. Я указал здесь 0.3 CPU, оно также может быть выражено как 300m, где m означает милли. Если мы укажем это в милли, то сможем спускаться ниже 100mCPU, но не ниже 1m.

1 счетчик ЦП эквивалентен 1 виртуальному ЦП. Это 1 виртуальный CPU в AWS, 1 ядро в GCP или Azure, или 1 Hyperthread.

Ты можешь запросить большее количество процессоров для контейнера при условии, что твои узлы могут это предоставить.

Точно так же с памятью. Ты можешь указать 256 мебибайтов, используя суффикс Mi. Или укажи такое значение в памяти вот так. Это будет в байтах. Или используй M для мегабайтов.



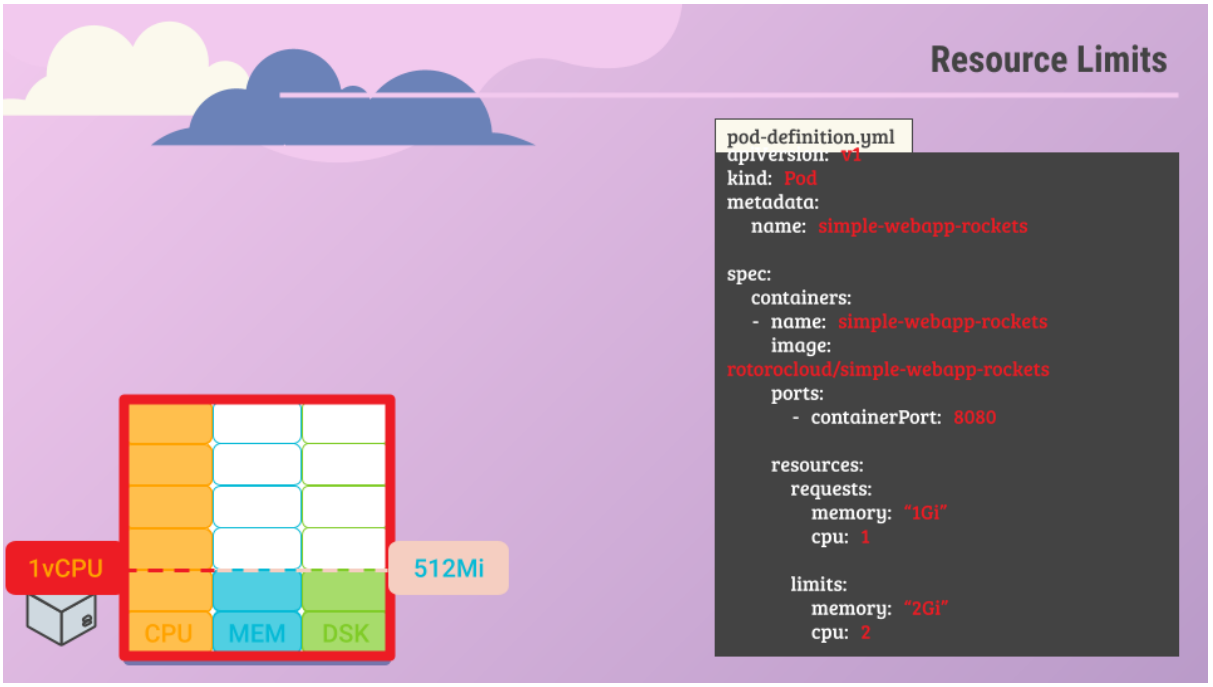
Для гигабайта используй суффикс G. Обрати внимание на разницу между G и Gi. G - это гигабайт, и это означает 1000 мегабайт, тогда как Gi относится к гибибайту, что равно 1024 мебибайтам.
То же самое и с мегабайтами, и с килобайтами.

Эти таблицы есть в документации, если ты вдруг запутаешься.

Давай теперь посмотрим на контейнер, работающий на узле в мире Docker.

По умолчанию у докер-контейнера нет ограничений на ресурсы, которые он может потреблять на узле. Допустим, контейнер начинается с 1 виртуального ЦП на узле, он может повышать загрузку CPU, потребляя столько ресурсов, сколько ему требуется. При этом он начнет душить нативные процессы ОС, а также другие контейнеры.

Однако мы можем установить ограничение на использование ресурсов для этих PODs. По умолчанию Kubernetes устанавливает для контейнеров ограничение в 1 виртуальный ЦП. Поэтому, если ты не укажешь явно, контейнер будет ограничен потреблением только 1 виртуального CPU ноды. То же самое и с памятью. По умолчанию Kubernetes устанавливает ограничение в 512MiB для контейнеров.



Если тебя не устраивает ограничение по умолчанию, можно изменить его, добавив параметр `limits` в раздел `resources` в файл определения POD. Укажи здесь новые ограничения для памяти и ЦП, которые тебе нравятся.

При создании POD, Kubernetes установит новые ограничения для контейнера. Помни, что limits и requests устанавливаются для каждого контейнера внутри POD.

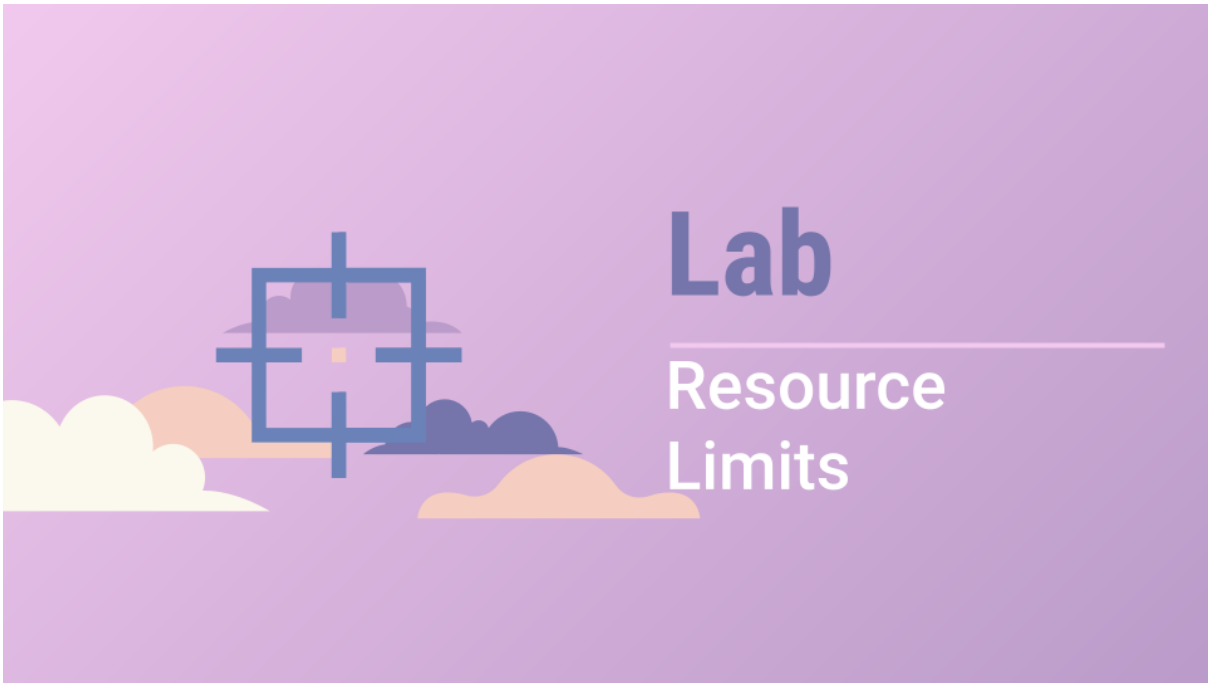
Но что если тебе не подходят ограничения по умолчанию, а ты не хочешь каждый раз их явно задавать? Ты можешь использовать механизм переопределения лимитов. Для этого создай новый объект типа LimitRange. Обрати внимание, что это LimitRange будет действовать для определенного namespace.

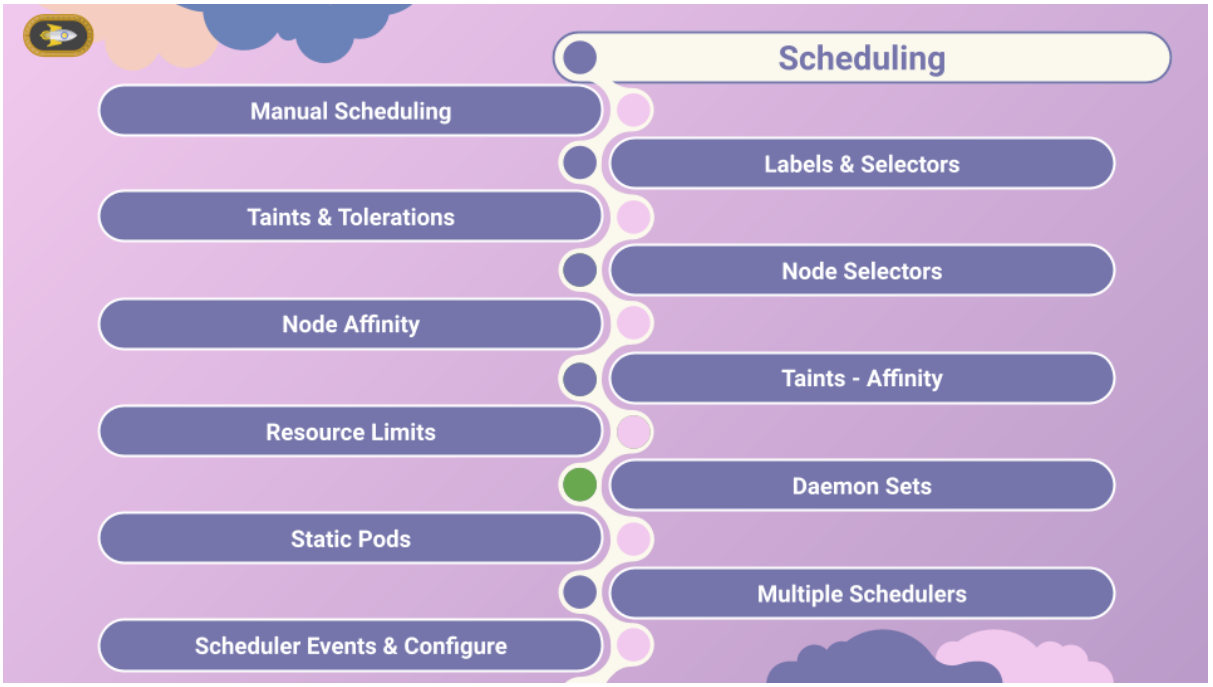
Итак, что происходит, когда POD пытается превысить ресурсы сверх указанного лимита?

В случае с CPU Kubernetes регулирует его так, чтобы время использования центрального процессора не превышало указанный предел.
И контейнер не сможет использовать больше ресурсов ЦП, чем установленный в параметре limit.

Однако с памятью дело обстоит иначе. Контейнер МОЖЕТ использовать больше ресурсов памяти, чем он ограничен. Таким образом, если POD постоянно пытается использовать больше памяти, чем установленный предел, POD будет остановлен.

Что ж, это все о требованиях к ресурсам в Kubernetes. Практикуйся в заданиях и обрати внимание на мои замечания о работе команды kubectl edit, они есть в презентации. Увидимся на следующей лекции.


Lab
Resource
Limits



Привет и добро пожаловать на лекцию. В этой лекции мы рассмотрим daemonsets в Kubernetes.

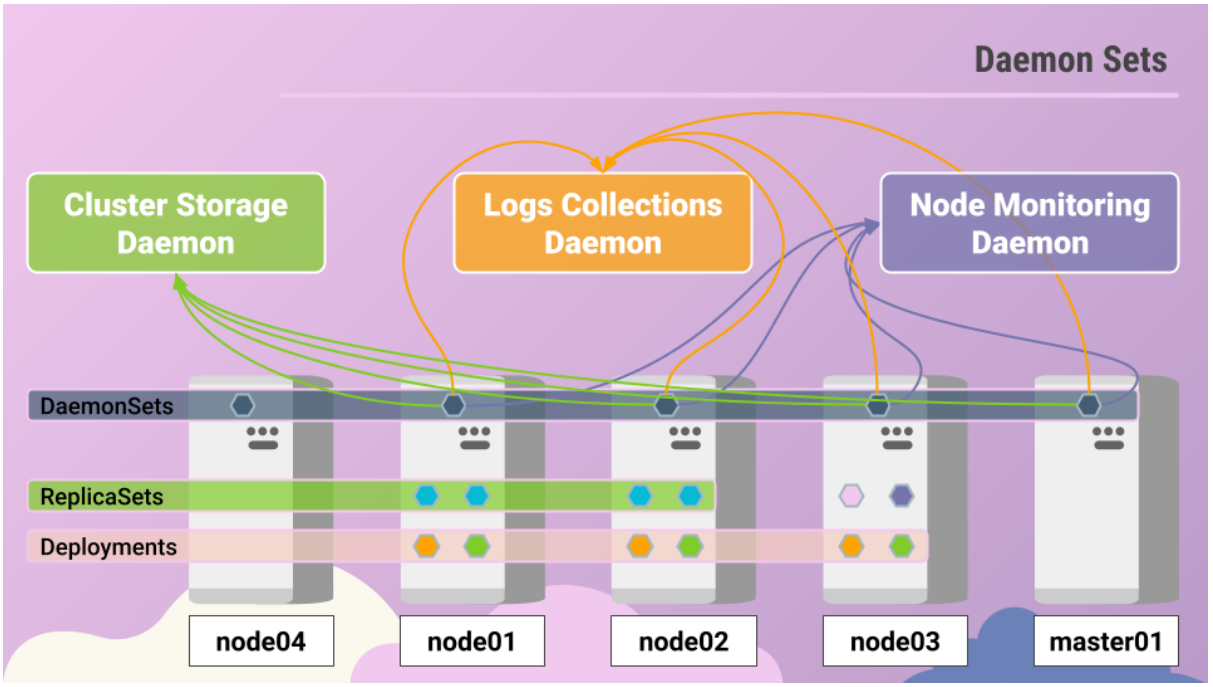
Пока мы разворачивали различные PODs на разных узлах нашего кластера с помощью replicaset и deployments. Два последних примитива помогли нам создать несколько реплик нашего приложения, распространив его по нескольким рабочим нодам.

Daemonset похож на replicaset, поскольку он помогает нам развернуть несколько экземпляров POD. Но есть отличия. Он запускает по одной копии своего POD на каждом узле кластера. Каждый раз, когда в кластер добавляется новая нода, реплика POD автоматически добавляется на этот узел, а когда узел удаляется, POD автоматически удаляется.
Daemonset гарантирует, что одна копия POD всегда будет присутствовать на всех узлах кластера.

Итак, в каких случаях показано применение daemonsets, какие у них use cases?

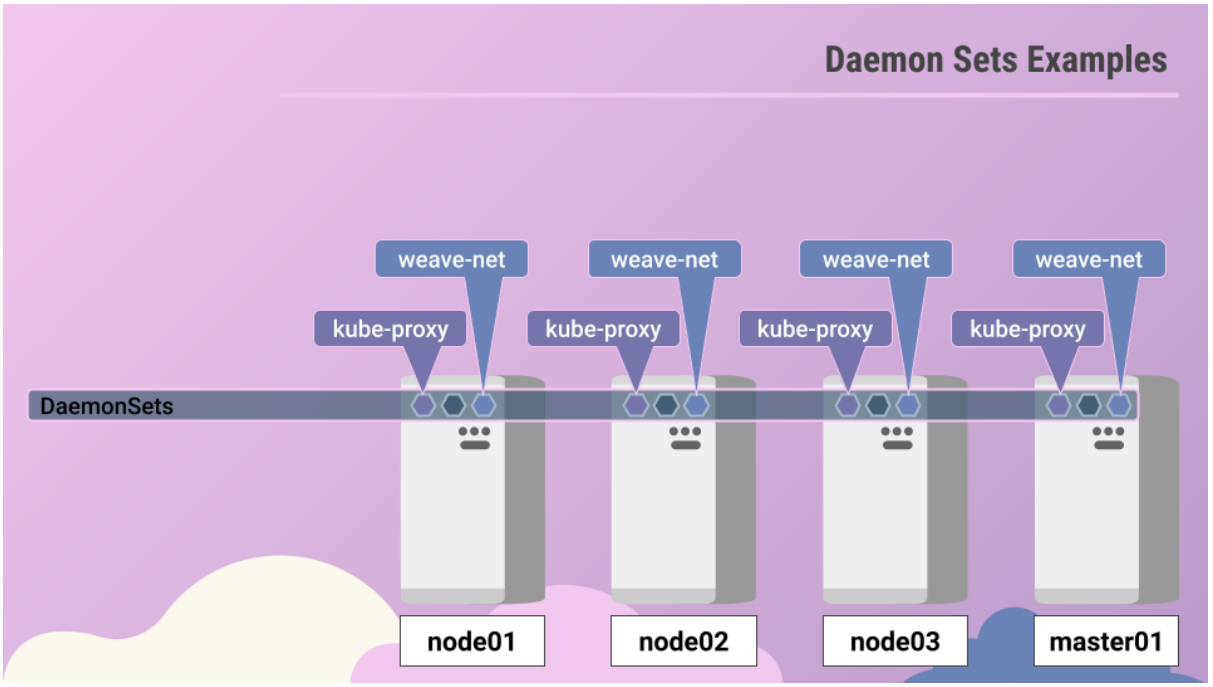
Допустим, чтобы перестать угадывать, что происходит в кластере, мы решили развернуть агент мониторинга и сборщик логов на каждой машине в кластере.

Daemonset в этом случае идеально подходит, поскольку он может развернуть наш агент мониторинга в форме POD на всех узлах кластера. Тогда нам не нужно беспокоиться о добавлении / удалении агентов мониторинга с этих узлов, когда в кластере что-то поменяется, поскольку daemonset позаботится об этом сам.



Обсуждая архитектуру Kubernetes, мы узнали, что одним из необходимых компонентов каждого узла в кластере является kube-proxy. Это один из подходящих вариантов, чтобы воспользоваться daemonset. Компонент kube-proxy можно развернуть как daemonset. Утилита kubeadm именно так и делает.

Другой вариант использования - работа в сети. Сетевые решения, такие как Weave-net, требуют развертывания агента на каждом узле кластера.



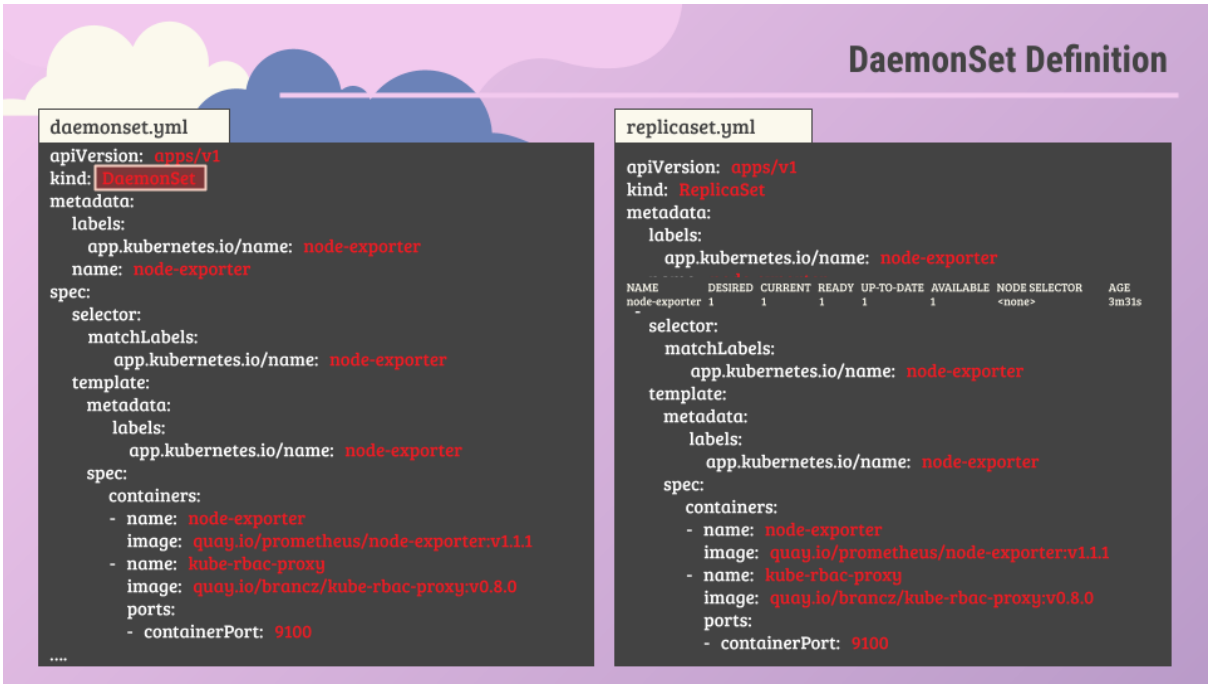
Мы обсудим сетевые концепции более подробно позже в ходе этого курса, но пока я просто хотел указать на это здесь.

Создание DaemonSet аналогично процессу создания ReplicaSet. У него есть спецификация вложенных POD в разделе шаблонов и селекторы для связывания daemonsets со своими PODs.

Файл определения daemonSet имеет аналогичную структуру.

Начнем как обычно с четырех: apiVersion, kind, metadata и spec.

apiVersion - apps/v1. Kind - будет DaemonSet вместо ReplicaSet.



Мы установим имя под metadata, это будет node-exporter. В соответствии со спецификацией у нас есть селектор и шаблон спецификации POD.
Это почти точно так же, как определение ReplicaSet, за исключением того, что это DaemonSet.

Убедись, что метки в селекторе совпадают с метками в шаблоне POD. Когда все будет готово, создай daemonset с помощью команды kubectl create daemonset. Для просмотра созданного daemonset выполни команду kubectl get daemonset.
И, конечно же, чтобы просмотреть более подробную информацию, запусти команду kubectl describe demonSet.

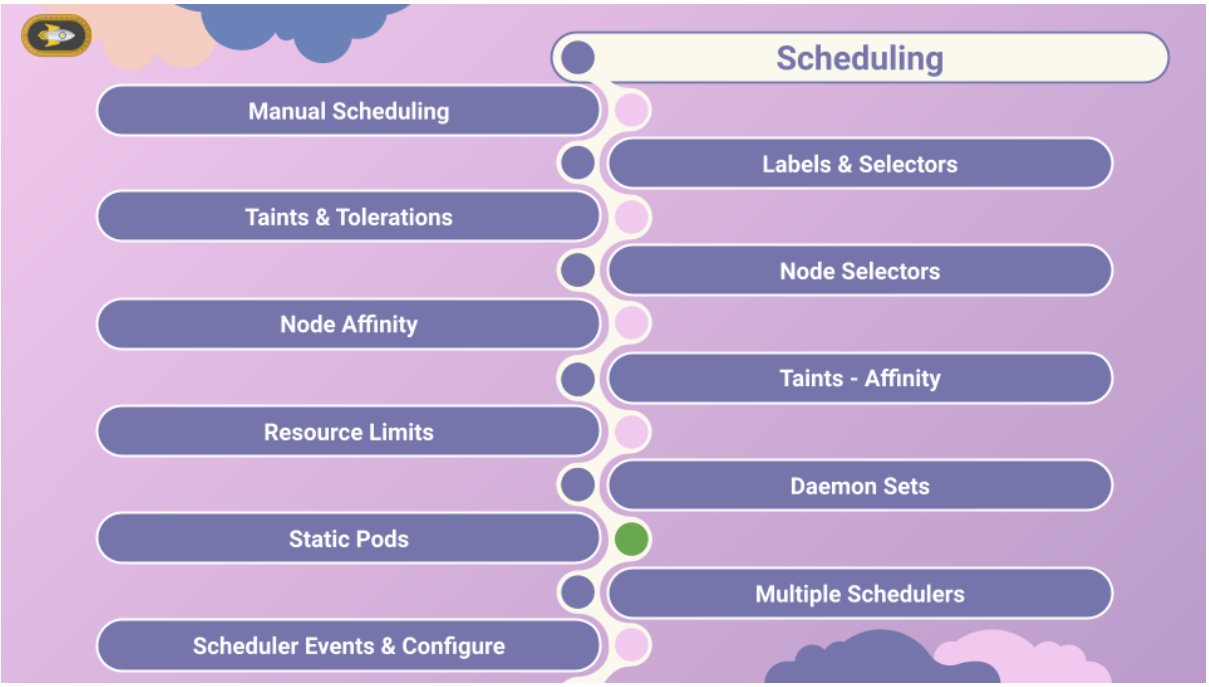
Итак, а как устроен daemonset? Как он назначает свои PODs на каждую ноду? И как он убеждается, что на каждой ноде есть именно один POD, ни больше, ни меньше?

Если тебя попросили назначать установку POD на каждом узле кластера, как бы ты это сделал?

В одной из предыдущих лекций этого раздела мы обсуждали, что можем вручную установить свойство nodeName в определении POD, чтобы обойти планировщик и поместить POD непосредственно на узел. Итак, это один из подходов.

Для каждого POD установим свойство nodeName в его спецификации до его создания, и когда они будут созданы, они автоматически попадут на соответствующие узлы. Так было до версии Kubernetes v1.12. Начиная с версии 1.12 и далее daemonset использует дефолтный scheduler и правила node affinity, о которых мы узнали в одной из предыдущих лекций. Вот так daemonset планирует PODs на ноды.

Ну вот и все в этой лекции. Переходи к практическому тесту и попрактикуйся в работе с DaemonSets.



Привет и добро пожаловать на эту лекцию. В этой лекции мы обсуждаем static PODs в Kubernetes, почему это важно и какие у них особенности.

Ранее, в этом курсе мы говорили об архитектуре и о том, что kubelet функционирует как исполнительная часть многокомпонентного аппарата controlplane.

Kubelet полагается в работе на kube-apiserver, он получает оттуда инструкции о том, какие PODs загружать на свой узел. Основанием для загрузки было решение, принятое kube-scheduler. Вдобавок все это было сохранено в надежном хранилище данных ETCD.

Но что если:

- не было kube-apiserver, kube-scheduler, контроллеров и кластера ETCD?
- не было бы мастера?
- не было других узлов.

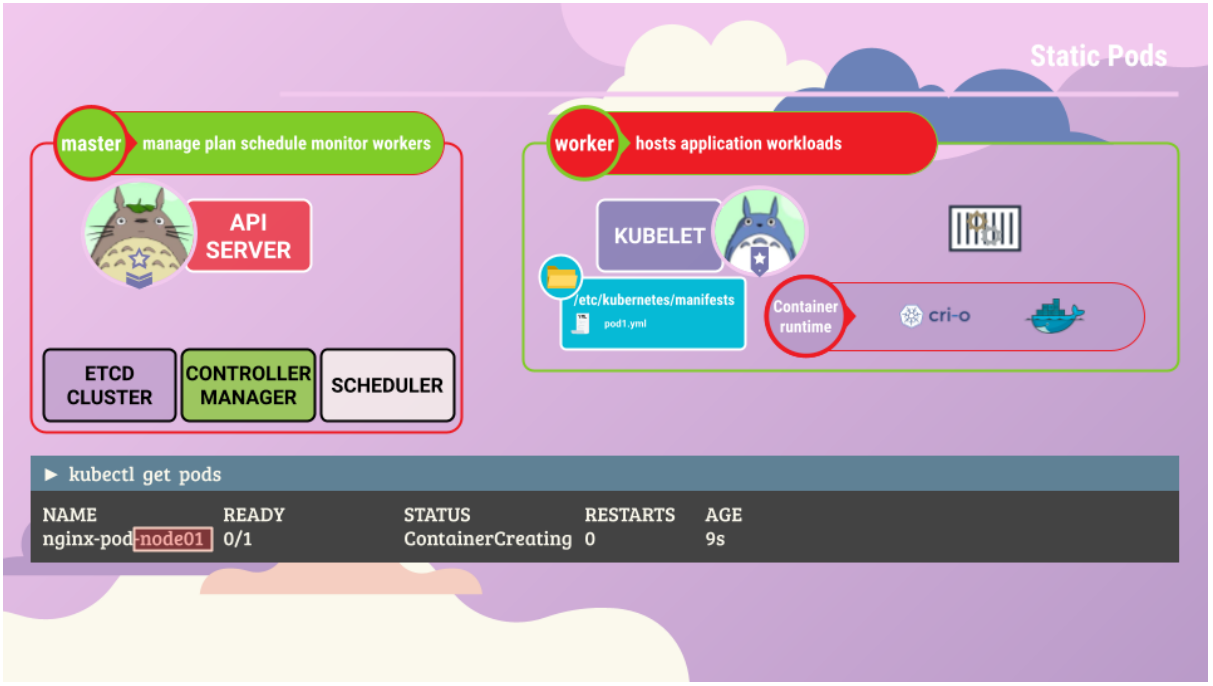
Что делать, если ты совсем один в дальнем космосе?
Т.е., в терминах Kubernetes, не часть кластера.

Может ли kubelet что-нибудь делать в роли капитана корабля?
Может ли он работать как независимый узел?
А если да, то кто предоставит инструкции, необходимые для создания POD?

Что ж, kubelet может управлять узлом самостоятельно.

На этом хосте у нас установлен kubelet. И, конечно же, у нас есть Docker для запуска контейнеров.

Но здесь нет кластера Kubernetes. Так что нет сервера API или чего-то подобного.



Kubelet умеет создавать POD. Но у нас нет kube-apiserver для того, чтобы рассказать kubelet подробности о POD. К настоящему времени мы знаем, что для создания POD нам понадобятся детали POD в файле определения. Но как нам предоставить файл определения POD для kubelet без API-сервера?

Ну например, мы можем настроить kubelet для чтения файлов определения PODs из специального каталога на сервере. Этот каталог будет использоваться для хранения информации о PODs. Файлы определений из этого каталога, kubelet периодически проверяет, и если что-то находит, то читает их и создает контейнеры на своем хосте. Он не только создает контейнер, но и гарантирует, что тот останется в живых. Если приложение вылетает, kubelet пытается перезапустить его. А если ты внесешь изменения в любой файл в этом каталоге, то kubelet воссоздаст POD, чтобы эти изменения вступили в силу.
Также, если ты удалишь файл из этого каталога, POD будет автоматически удален.

Итак, эти POD, созданные kubelet без вмешательства сервера API и остальных компонентов кластера Kuberentes известны как static PODs.

Важно, что мы можем создавать только PODs таким образом. У нас не выйдет создать replicaset, deployment или service, если мы положим в этот каталог подобный манифест.

Все эти объекты являются концептуальной частью архитектуры Kubernetes, для которой требуются другие компоненты controlplane, с которыми они взаимодействуют. Например, контроллеры репликации и деплойментов и т. д.
Kubelet работает на уровне POD и может понимать только их. Вот почему таким образом можно создавать static PODs.

Итак, что это за такая особенная папка и как ее настраивают.

Это может быть любой каталог на хосте. И расположение этого каталога передается в kubelet в качестве опции при запуске службы. Параметр называется `pod-manifest-path`, и здесь он установлен в `/etc/kubernetes/manifests`.

Существует также другой способ настроить это: вместо того, чтобы указывать параметр непосредственно в файле kubelet.service, можно указать путь к другому файлу конфигурации с помощью параметра `--config` и определить путь к каталогу как `staticPodPath` в этом файле.

Этот подход используется при настройке кластеров с помощью инструмента kubeadm. Если ты разбираешься с настройками уже существующего кластера, тебе следует проверить эту опцию kubelet, чтобы определить путь к каталогу.

После этого ты узнаешь, где можно разместить файлы определений для своих static PODs. Так что имей это в виду, когда станешь проходить лабораторные работы. Тебе нужно знать, как просмотреть и настроить этот параметр. Независимо от метода, используемого для настройки кластера, сначала проверь параметр pod-manifest-path в файле службы kubelet. Если его там нет, найди параметр --config и установи, какой файл используется в качестве файла конфигурации. Затем в этом файле конфигурации найди параметр staticPodPath.

Один из этих подходов должен дать указать тебе правильный путь в директорию статических PODs.

После создания static POD, стоит на них посмотреть запустив команду docker ps.

А почему бы не использовать команду kubectl, как мы это делали до сих пор?

Помнишь, что мы одни в космосе, у нас нет контролькой станции, т.е. у нас нет остальной части кластера Kubernetes.
Утилита kubectl работает с kube-apiserver. А поскольку сейчас у нас нет сервера API, утилита kubectl бесполезна.
Поэтому мы используем команду docker.

ОК, когда узел является частью кластера, это работает так. Сервер API передает kubelet запрос, чтобы он создавал POD.

А может ли kubelet создавать оба типа PODs одновременно? Kubelet работает так, что он может принимать запросы на создание PODs из разных точек входа.

Первая - через файлы определения POD из папки статических определений, как мы только что видели.
Второй - через endpoint HTTP API. Именно так kube-apiserver предоставляет данные для работы kubelet.
Kubelet может создавать оба типа PODs - статические и те, что с API-сервера и делать это одновременно.

Но тогда, знает ли сервер API о static PODs ноды, созданных kubelet?

Да, знает.

Если мы запустим команду kubectl get pods на мастере, static PODs будут перечислены как любые другие PODs.

А как это происходит?
Когда kubelet создает static POD, если он является частью кластера, он также создает зеркальный объект в kube-apiserver.

То, что мы видим в kube-apiserver - это просто тень того статического POD и она доступна только для чтения.
Ты можешь просматривать сведения о POD, но не можешь редактировать или удалять его, как это делали с обычными.

Их можно удалить, только изменив файлы из папки манифеста на узле, где выполняется static POD.

Обрати внимание, что к имени автоматически добавляется еще и имя узла.
В этом случае node01.

Давай поговорим, зачем нам использовать статические PODs?

Поскольку static PODs не зависят от controlplane Kubernetes, мы можем использовать static PODs для развертывания непосредственно самих компонентов плоскости управления в качестве PODs на мастер-узлах.

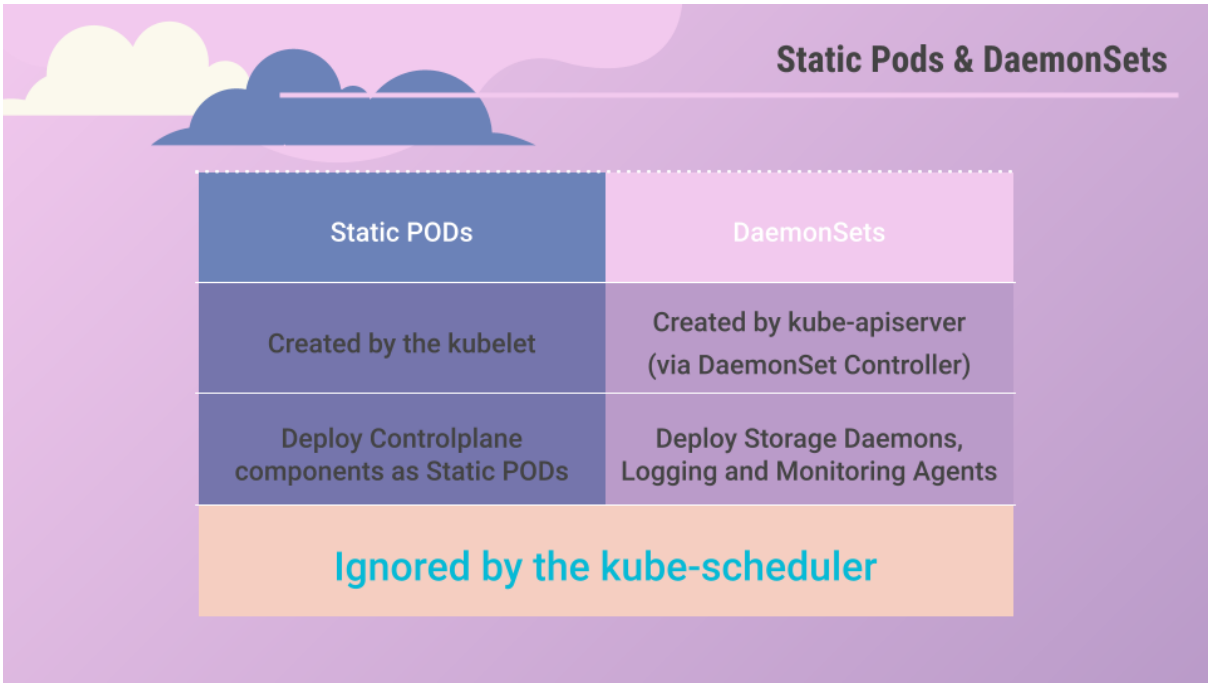
Для этого начнем с установки kubelet на все главные узлы. Затем создадим файлы определения PODs, которые используют докер-образы различных компонентов controlplane, таких API-server, controller-manager, etcd и т. д.

Поместим файлы определений в указанную папку манифестов. Теперь kubelet сам позаботится о развертывании остальных компонентов Kubernetes в качестве PODs в кластере.

Таким образом, нам не нужно загружать бинарники, настраивать службы или беспокоиться о том, что какая-та из служб упадет.
Если какой-либо из этих компонентов выйдет из строя, поскольку это static POD, он будет автоматически перезапущен kubelet. Аккуратно и просто.

Вот так инструмент kubeadm и настраивает кластер Kubernetes.

Вот почему в кластере, настроенном с помощью инструмента kubeadm, когда ты перечисляешь PODs в пространстве имен kube-system, то видишь компоненты controlplane как PODs.



Далее в курсе будет лабораторная, где мы будем ставить кластер с помощью kubeadm.

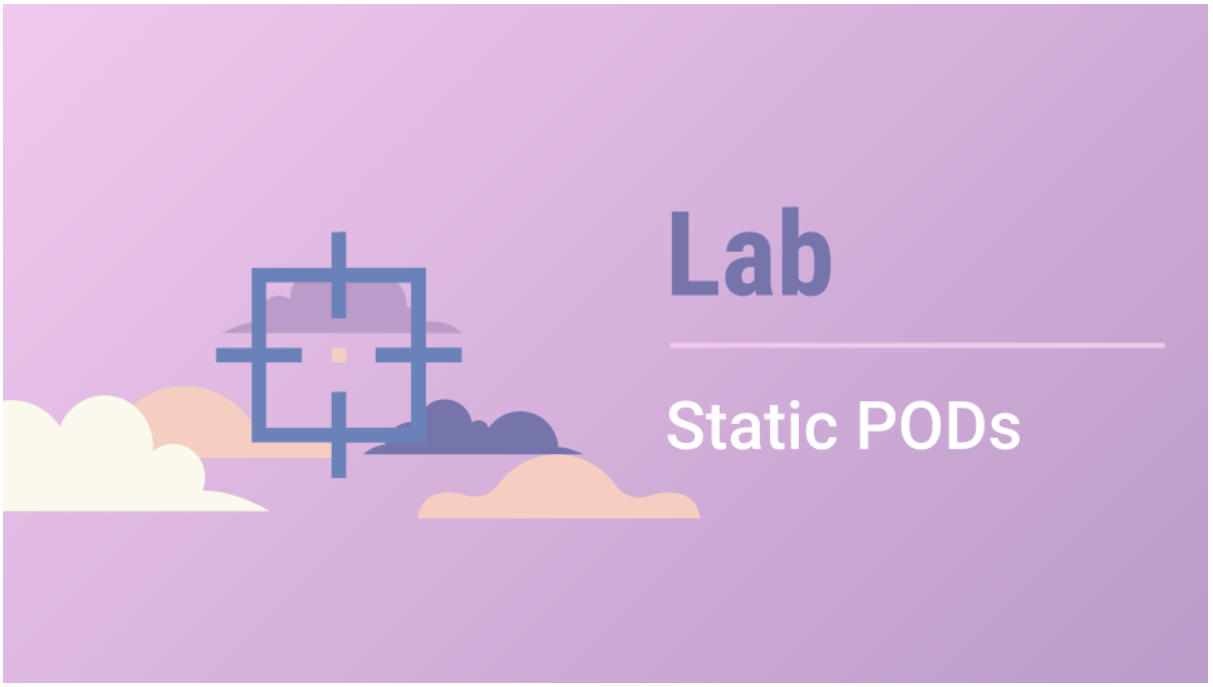
А сейчас мы почти готовы занятся практикой и поработать со static PODs, но прежде есть один вопрос, который мне часто задают:
в чем различия между static PODs и daemonsets?

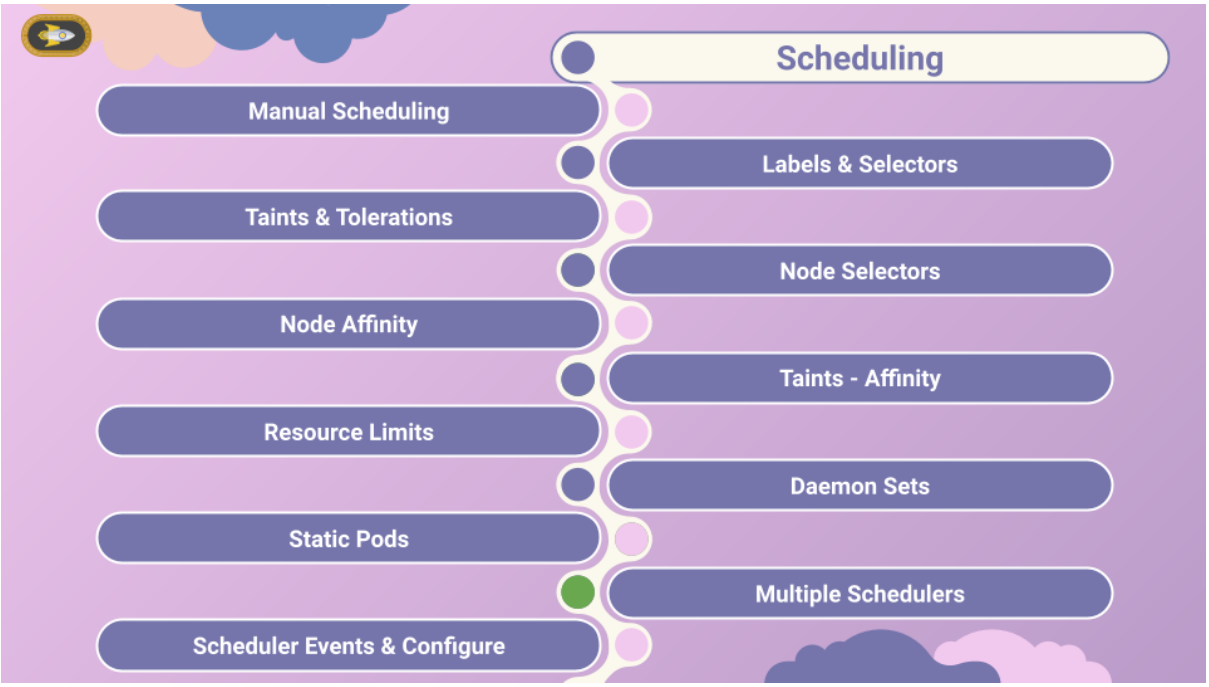
Daemonsets, как мы видели ранее, используются для обеспечения доступности одного экземпляра приложения на всех узлах кластера. За это отвечает контроллер daemonset, он обрабатыват ситуации, связанные с daemonsets, и делает это через kube-apiserver. В то время как статические PODs, как мы видели в этой лекции, создаются непосредственно kubelet без какого-либо вмешательства со стороны API-сервера или каких-то других компонентов плоскости управления Kubernetes. Static PODs могут использоваться для развертывания самих компонентов controlplane Kubernetes.

Но как static PODs, так и PODs, созданные через daemonset, kube-scheduler игнорирует. Планировщик никак не влияет на эти PODs.

Ну вот и все в этой лекции.

Отправляемся на практику со статическими PODs. Жду в следующей лекции.


Lab
Static PODs



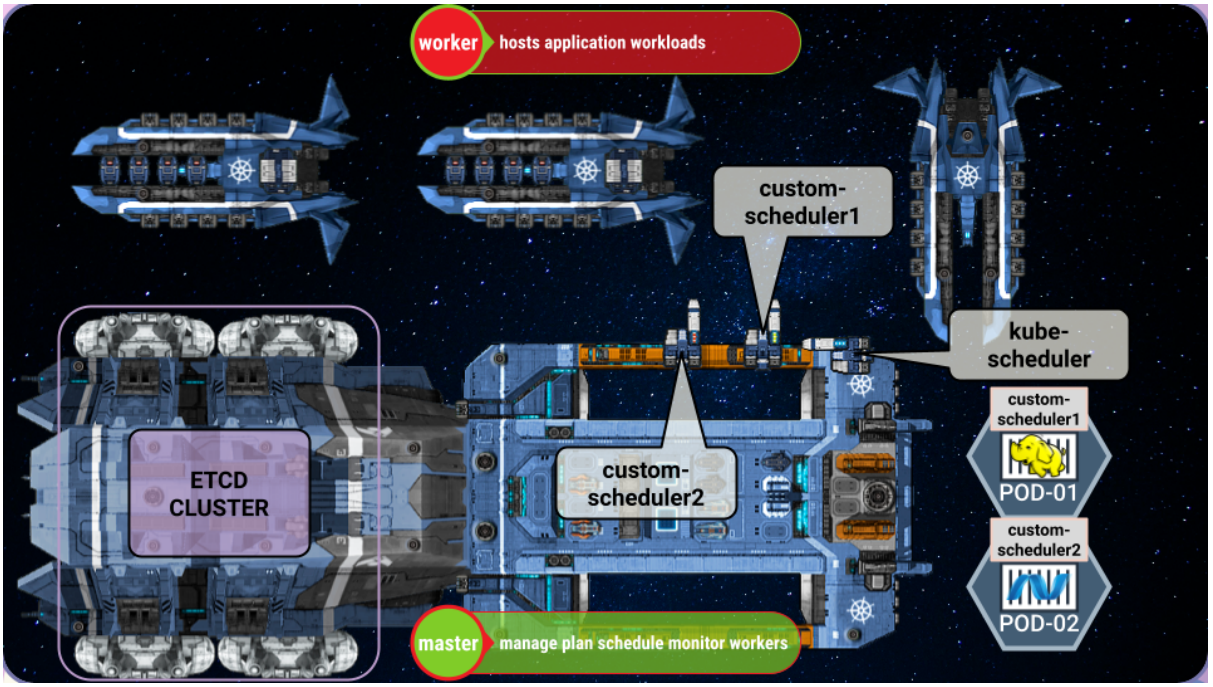
Привет и добро пожаловать на лекцию. В этой лекции мы рассмотрим как в Kubernetes работают сразу несколько механизмов по назначению PODs на узлы. Мы также узнаем, как просматривать события, связанные с планировщиком.

В предыдущих лекциях мы видели, как работает планировщик по умолчанию в среде Kubernetes. У него есть алгоритм, который равномерно распределяет PODs по узлам. Он также принимает во внимание различные условия, которые мы указываем через taints and toleration, node affinity и т. д.

Но что, если ни его работа не удовлетворяет наши потребности?
Предположим, у нас есть конкретное приложение, которое требует, чтобы его компоненты были размещены на узлах после выполнения дополнительных проверок.

Значит ли это, что нам нужно использовать собственный алгоритм назначения для размещения PODs на узлах, чтобы он обрабатывал наши дополнительные условия и проверки?
Kubernetes очень расширяемый. Мы можем написать свою собственную программу планировщика в Kubernetes, упаковать ее и развернуть как в качестве default scheduler, так и дополнительного планировщика в кластере.

Например так, чтобы приложения в кластере могли использовать планировщик по умолчанию как раньше для всех своих нагрузок, однако какое-то конкретное приложение обслуживал бы этот кастомный планировщик.



Таким образом в твоем кластере Kubernetes может быть несколько планировщиков одновременно. При создании PODs или deployments мы можем указать Kubernetes, чтобы данный POD назначается именно определенным планировщиком.

Ранее мы видели, как развернуть kube-scheduler. Мы загружаем двоичный файл kube-scheduler и запускаем его как службу с набором параметров. Одна из опций - `--scheduler-name` - имя планировщика.

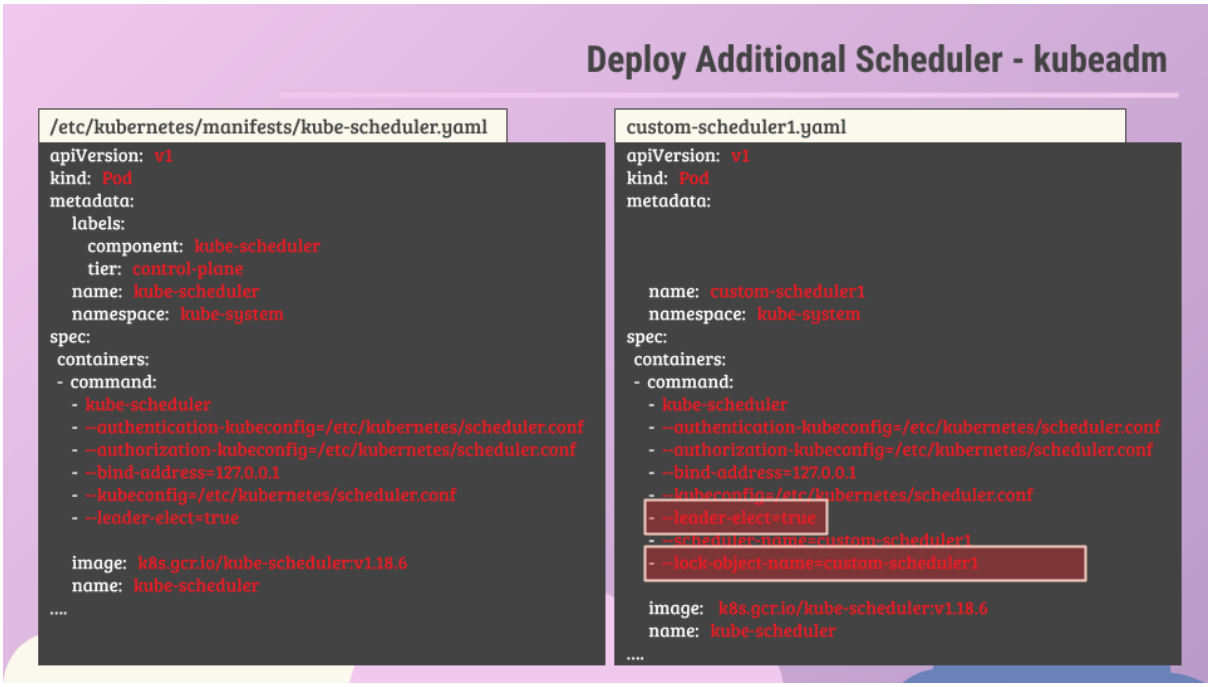
Если не указано, предполагается имя `default-scheduler`.

В Kubernetes по умолчанию планировщиком является kube-scheduler. Чтобы развернуть дополнительный планировщик, ты можешь использовать тот же двоичный файл kube-scheduler или подключить свой особенный исполняемый файл.
В этом случае мы будем использовать тот же двоичный файл для развертывания дополнительного планировщика. Поскольку движок остается прежним, это будет по сути тот же kube-scheduler, но с тюнингованым под наши задачи алгоритмом планирования.

Мы меняем имя планировщика на `custom-scheduler1`.
Это важно, чтобы различать два планировщика, и это имя, которое мы будем указывать в файле определения POD позже.

Теперь давай посмотрим, как это сделать с инструментом kubeadm.
Напомню, если ты забыл, утилита kubeadm развертывает scheduler как POD.

Ты можешь найти файл определения, который он использует в папке манифестов. Я убрал все остальные детали из файла, чтобы мы могли сосредоточиться на ключевых частях.



В разделе `command` есть команда и связанные параметры для запуска планировщика.
Мы можем создать собственный планировщик, сделав копию того же файла и изменив имя файла-манифеста.

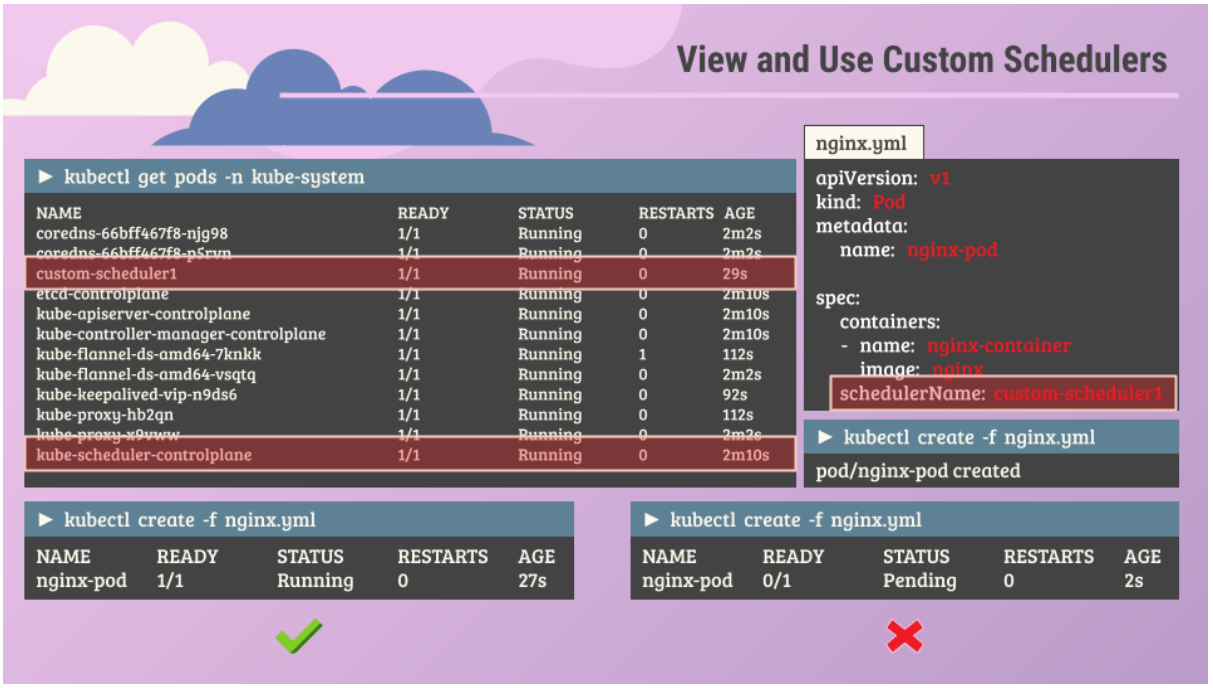
В манифесте мы устанавливаем имя POD custom-scheduler1 и добавляем новую опцию в команду kube-scheduler `--scheduler-name=custom-scheduler1`, чтобы установить собственное имя для нашего планировщика.

Наконец, важный момент, на который стоит обратить внимание, - это опция с избранием лидера - `--leader-elect=true`. Опция выбора лидера используется, когда у нас есть несколько экземпляров планировщика, запущенных на разных мастер-узлах. В сетапах с высокой доступностью у нас будет несколько главных узлов, на которых будет запущен процесс kube-scheduler.

Если на разных узлах запущено несколько копий одного и того же планировщика, единомоментно может быть активным лишь один.
Именно здесь опция leader-elect помогает выбрать лидера, который будет руководить деятельностью по назначению нагрузок.

Мы обсудим больше о настройке HA в другом разделе, но сейчас я хотел бы указать, что для того, чтобы несколько планировщиков работали и не конфликтовали, тебе потребуется установить параметры для выбора лидера. Это будет значение `false` в случае, если у тебя всего один мастер и `true` если их больше.

Еще передадим дополнительный параметр --leader-elect-resource-name, чтобы наш планировщик мог правильно работать без конфликтов со своими экземплярами во время выборов лидера. Обрати внимание, для этого раньше использовался параметр `lock-object-name`, сейчас считается устаревшим, хоть и повсеместно используется.



После этого создай POD помощью команды kubectl create. Запусти команду get pods в пространстве имен kube-system и найди свой кастомный планировщик.
Убедись, что он в рабочем состоянии. Если ты где-то ошибся, он не будет в статусе `running`.

Следующим шагом является настройка нового POD или deployment для использования нового планировщика. В файле спецификации POD добавь новое поле с именем schedulerName и укажи там имя нового планировщика.

Таким образом, когда POD создается, им начнет заниматься правильный планировщик. Создай POD с помощью команды kubectl create. Если планировщик был настроен неправильно, POD продолжит оставаться в состоянии `Pending`. Если все в порядке, то статус будет `running`.

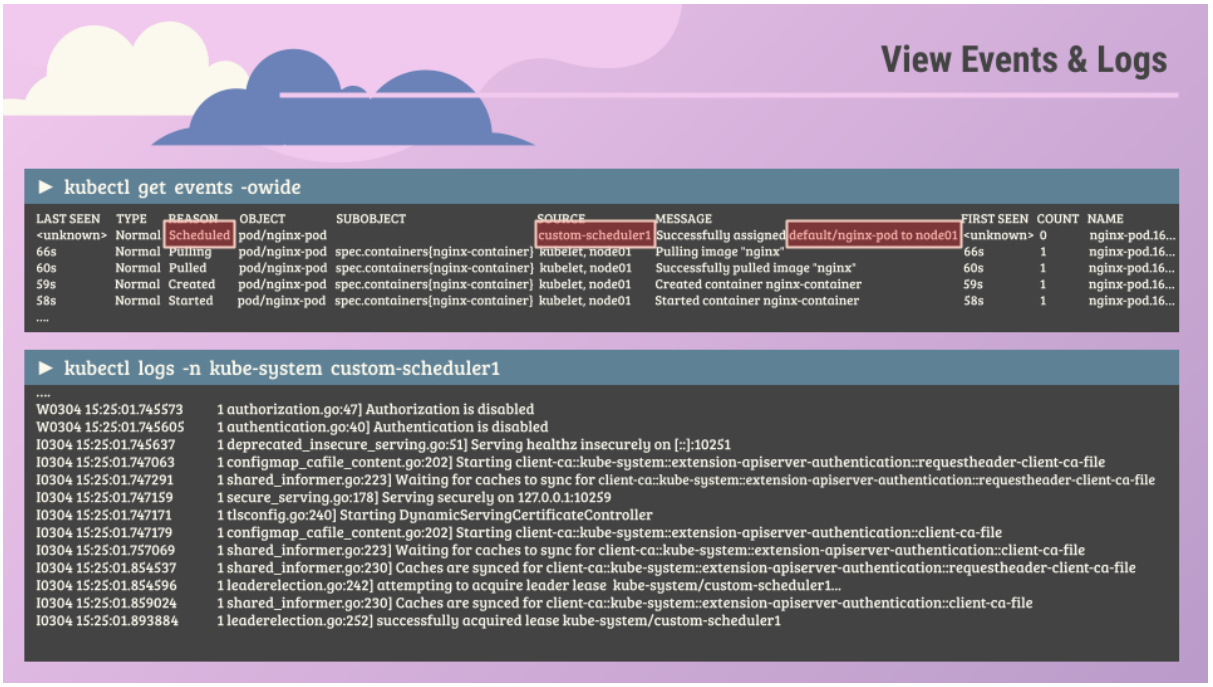
Итак, как узнать, какой планировщик его подхватил? Посмотри события с помощью команды kubectl get events.

Здесь перечислены все события в текущем пространстве имен.

Ищи события шедулига. Как видишь, источником события является созданный нами планировщик custom-scheduler1.

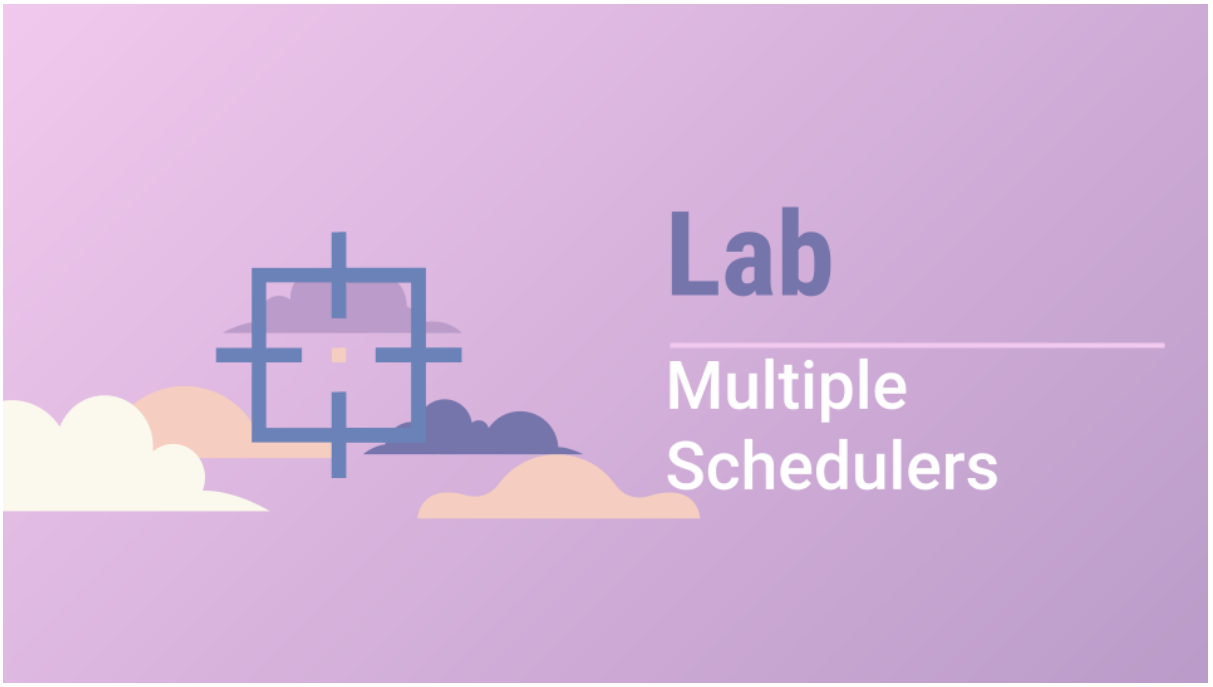
И в сообщении говорится, что nginx-pod из default namespace был успешно назначен.

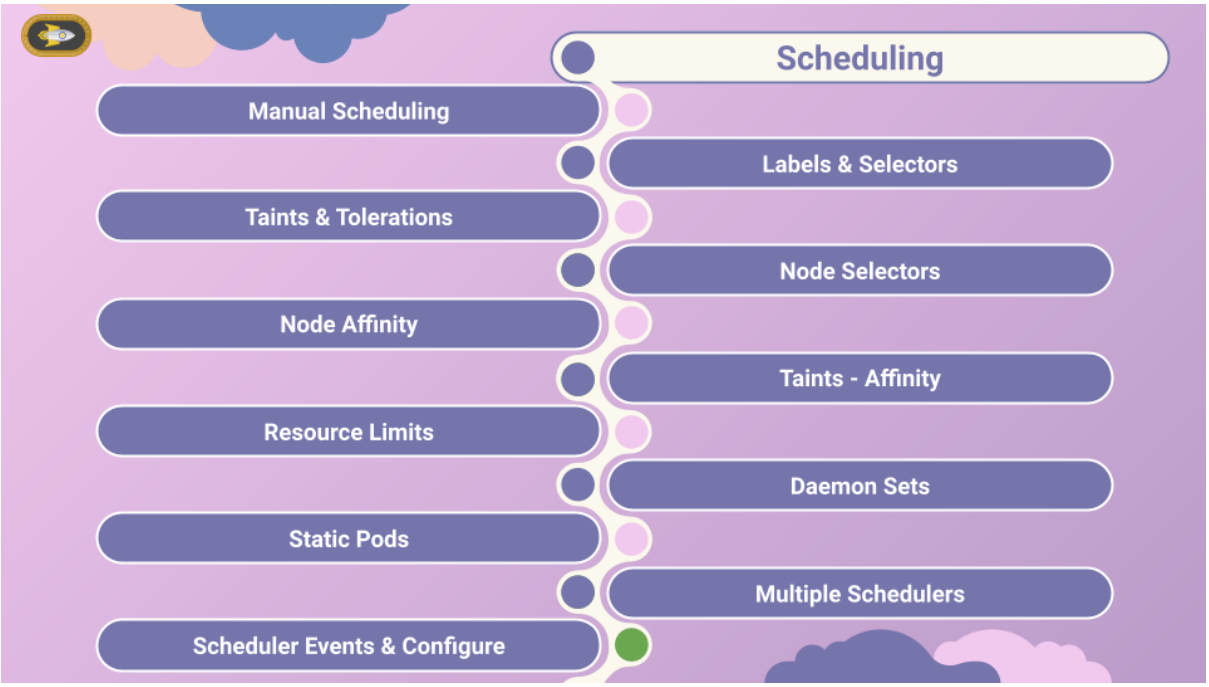
Еще для понимания, что происходит и почему что-то не получается полезно посмотреть логи планировщика.



Это можно сделать с помощью команды kubectl logs. Укажи название планировщика и правильное пространство имен - как ты помнишь наш custom-scheduler1 находится в системном namespace.

Ну вот и все в этой лекции. Перейди к практическим упражнениям и потренируй развертывание нескольких планировщиков. Увидимся на следующей лекции.





Привет и добро пожаловать. Здесь мы разберем пример настройки алгоритма scheduler.
Это необязательная лекция, данный материал не требуется при сдаче экзамена.

Мы уже разобрались, как запустить свой собственный scheduler.

Как ты помнишь, kube-scheduler является неотъемлемым компонентом Kubernetes, который отвечает за планирование PODs по нодам в соответствии с заданными политиками.

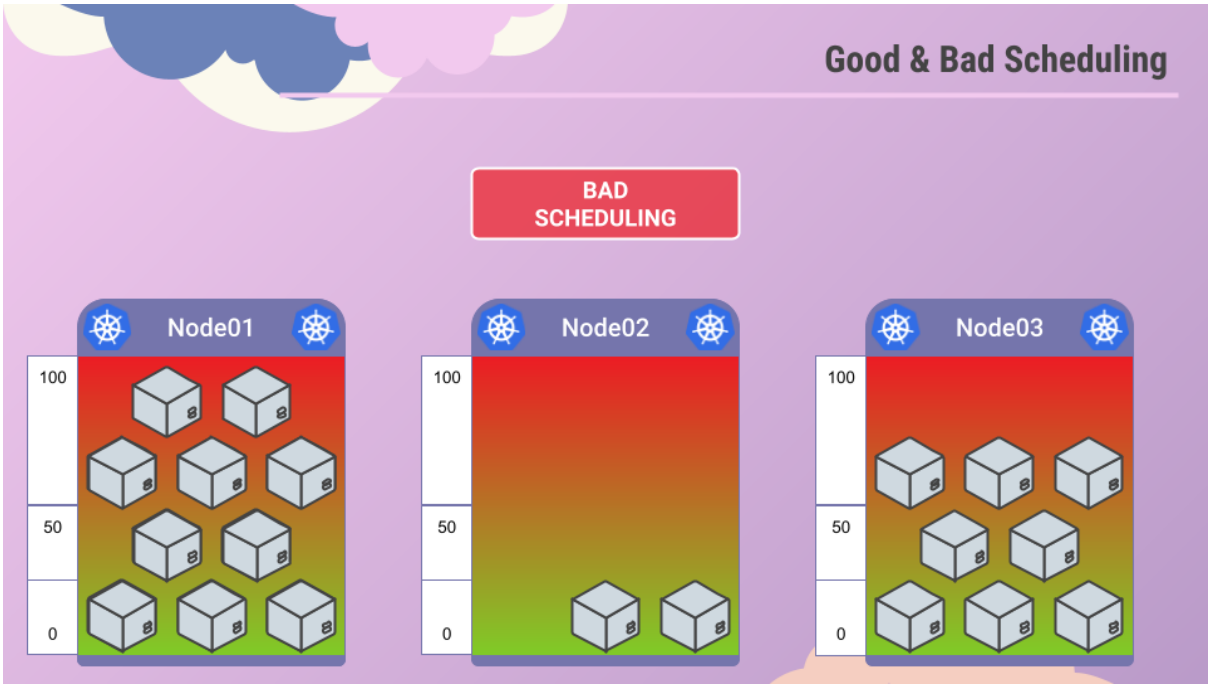
Обычно, в процессе эксплуатации Kubernetes-кластера нам не приходится задумываться о том, по каким именно политикам происходит планирование PODs, так как набор настроек дефолтного kube-scheduler'а подходит для большинства повседневных задач.

Однако встречаются ситуации, когда нам важно тонко управлять процессом распределения PODs, и для выполнения этой задачи есть два пути:

- создать kube-scheduler с кастомным набором правил
- написать свой собственный scheduler и научить его работать с запросами API-сервера

Здесь мы посмотрим, как реализовать первый путь.

Давай представим, что у нас есть три одинаковые по физическим параметрам ноды (часть CICd), на которые каждую минуту прилетает на выполнение cronjob, которая расходует CPU, но сколько выполняется по времени мы точно не знаем.
В среднем одновременно выполняются 5-6 работ на ноде, занимая 50-60% вычислительной мощности.



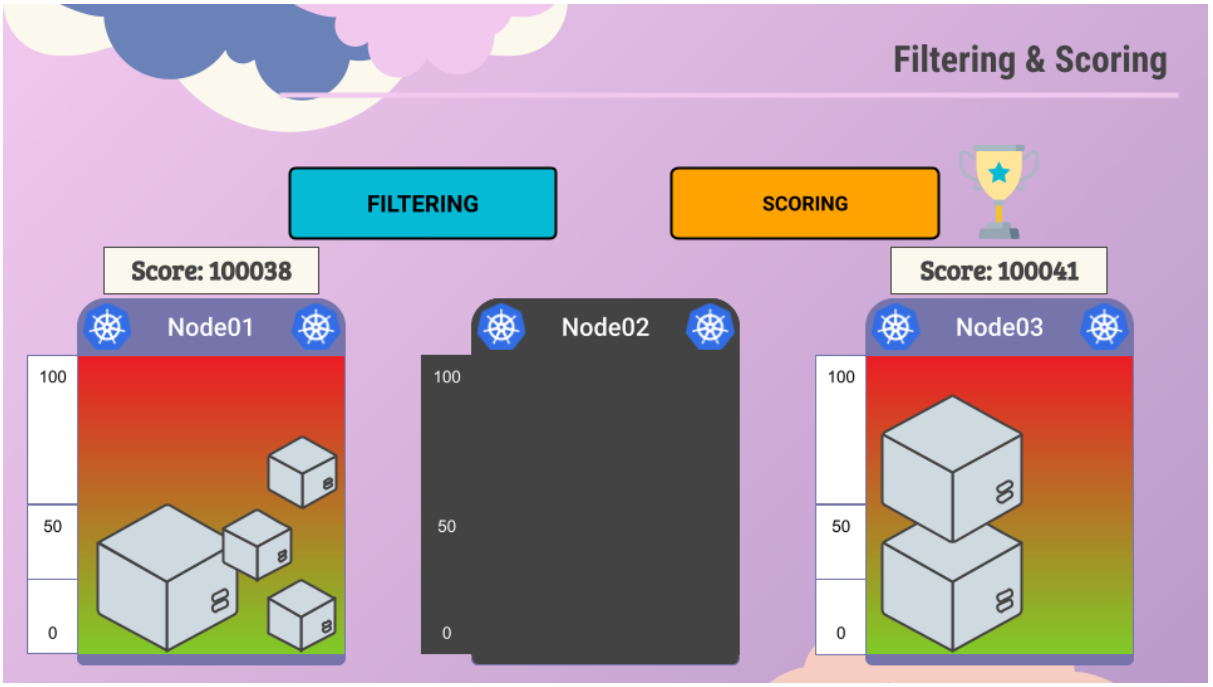
Это происходит большую часть времени и мы назовем это хорошим шедулингом.

Но иногда, одна из нод переставала планироваться на эти задачи, и мы получали такую картину - две ноды с загрузкой 80%, а вторая нода могла быть вообще не загружена. Это был очень плохой шедулинг.

Прежде чем мы двинемся дальше, хочу отметить, что kube-scheduler не отвечает за непосредственное планирование PODs — он отвечает только за определение ноды, на которую нужно разместить POD.
Иначе говоря, результат работы kube-scheduler’a — это имя ноды, которое он возвращает API-серверу на запрос о планировании и на этом его работа заканчивается.

Когда планировщик получает задачу, он приступает к составлению списка нод, на которые может быть запланирован POD в соответствии с политиками predicates. Это известно как filtering.
Как видишь на этом этапе он отсеял ноду номер 2, т.к. на ней нет нужных ресурсов и он считает, что ближайшее время ничего не изменится.

После того как фильтрация выполнена, ноды из этого списка получают определённое количество очков в соответствии с политиками priorities.
В результате выбирается нода, набравшая максимальное количество очков. Если есть ноды, набравшие одинаковый максимальный балл, выбирается случайная.
Со списком и описанием политик predicates (filtering) и priorities (scoring) можно ознакомиться в документации.



Итак у нас победила нода номер 3 и на нее размещается POD.

Вернемся к нашей проблеме с простаивающей нодой.
Чтобы узнать, почему такое происходит, повысим уровень логирования scheduler'a до 10 (--v 10).

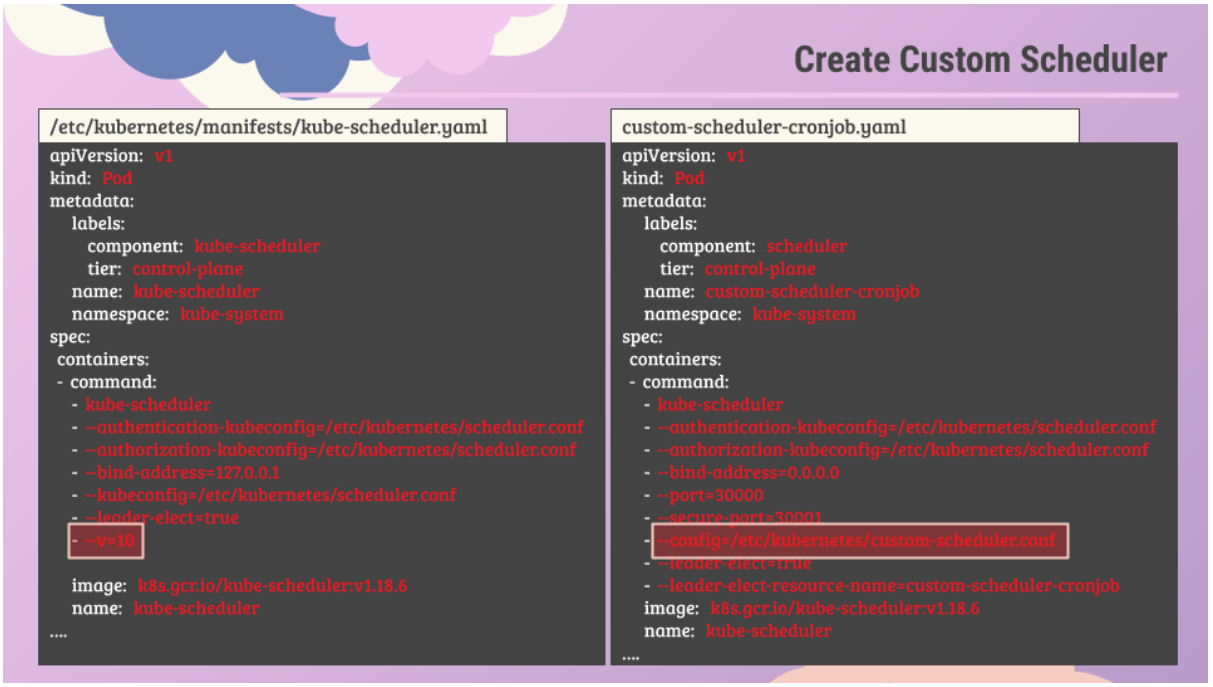
Как видишь в процессе скоринга по стандартным политикам одна нода постоянно отстает от других и соответственно не участвует в планировании.

К сожалению, приоритеты kube-scheduler не очень прозрачны, об этом можно почитать в десятках issues на github. Поэтому можно опытным путем подобрать кастомные политики приоритетов для своего случая.

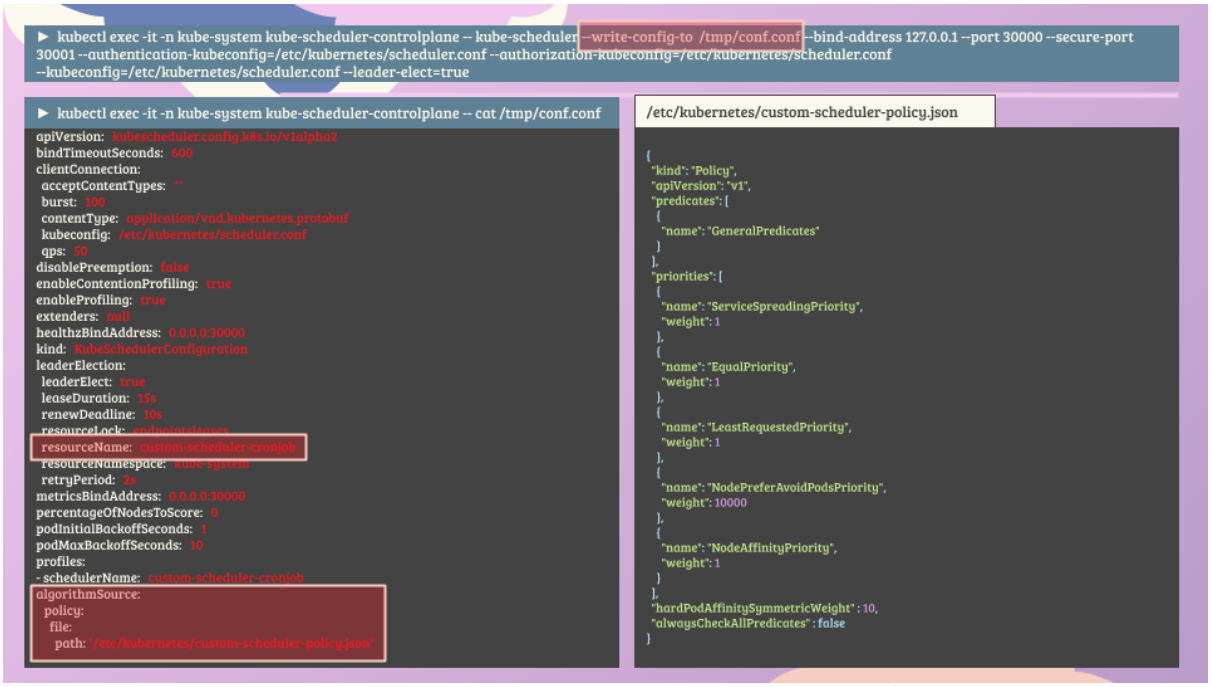
Итак, мы делаем следующее:
- создаем новый scheduler на основе дефолтного
- забираем его настройки для работы в кластере
- тестируем наши приоритеты методом проб и ошибок

Начнём с написания манифеста для нашего нового kube-scheduler’a. За основу возьмём манифест дефолтного (/etc/kubernetes/manifests/kube-scheduler.yaml) и приведём его к виду, как на экране.
Этот файл не полностью, сюда не поместились строки на монтирование файла конфигурации и кастомных политик.

Итак, здесь мы Изменили имя POD, изменили порты и конфиг.
По умолчанию kube-scheduler занимает порт 10251 и секьюрный порт 10259, у нас 30000 и 30001 соответственно.



Чтобы получить этот конфиг, с помощью этой длинной команды с опцией --write-config-to мы сдампим конфигурацию дефолтного kube-scheduler'а под наши настройки.
Здесь поменяем название scheduler на наш, и как и ранее настроим leaderElect. Добавим параметр algorithmSource для оверрайда дефолтных политик.

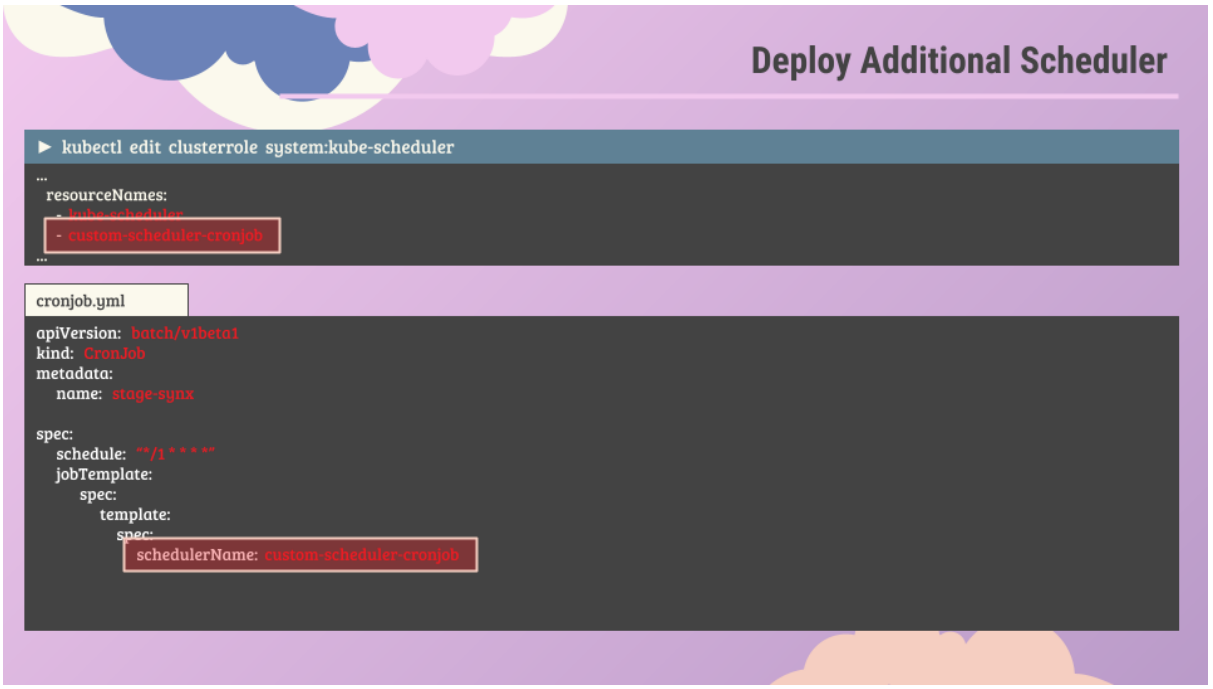


Далее добавим в нее свои новые политики, эти политики были подобраны опытным путем. Все описано в формате JSON.

Нашему scheduler потребуются права, аналогичные дефолтному. Добавим его в кластерную роль system:kube-scheduler.

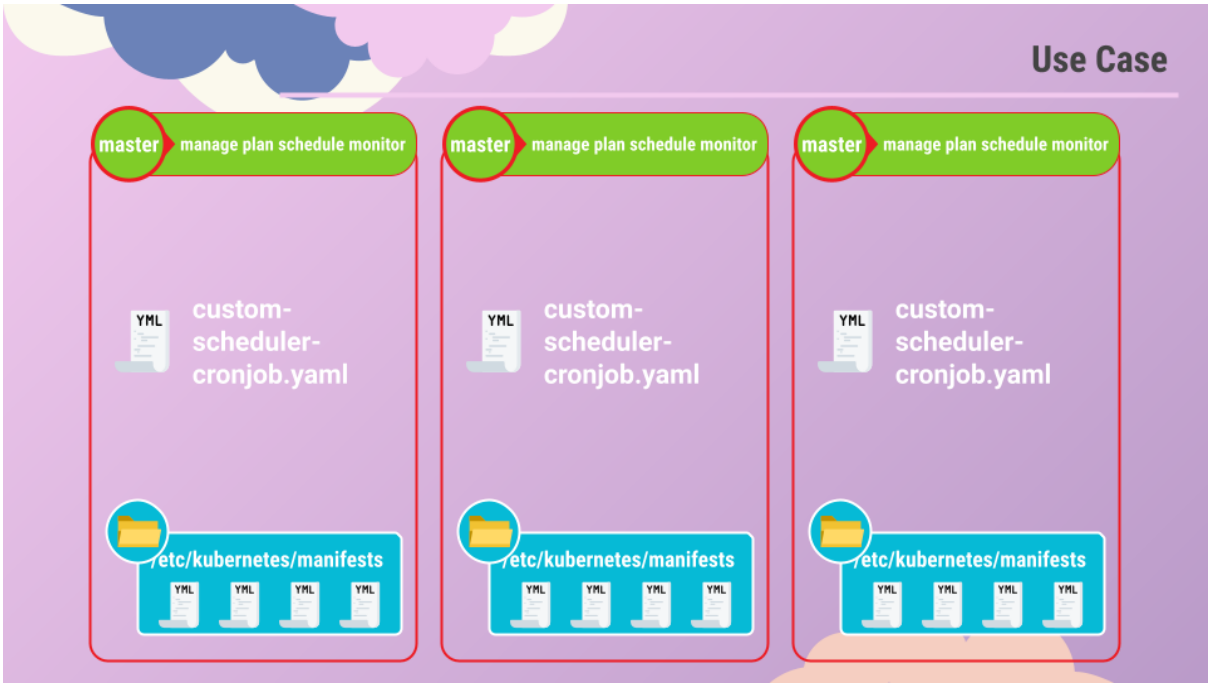
Теперь остаётся только указать в spec нашей CronJob, что все запросы на планирование её PODs должен обрабатывать наш новый scheduler.

Манифест нового custom-scheduler, который мы создали в начале, custom-scheduler-cronjob.yaml разместим по следующему пути /etc/kubernetes/manifests на трёх мастер-нодах. Если всё выполнено правильно, kubelet на каждой ноде запустит этот POD.



Итак, в конечном итоге мы получили дополнительный kube-scheduler с уникальным набором политик планирования, за работой которого следит непосредственно kubelet. Кроме того мы настроили выборы нового лидера между PODs нашего kube-scheduler'а в случае, если старый лидер по каким-то причинам становится недоступен.

Обычные приложения и сервисы продолжают планироваться через дефолтный kube-scheduler, а все cron-задачи полностью переведены на новый.
Нагрузка, создаваемая cron-задачами, теперь равномерно распределяется по всем нодам. Учитывая, что большая часть cron-задач выполняется на тех же нодах, что и основные приложения проекта, это позволило значительно снизить риск переезда PODs из-за нехватки ресурсов.



После внедрения дополнительного kube-scheduler'а в наш сетап, проблем с неравномерным планированием cron-задач больше не возникало.

Также обрати внимание, что это одно из решений, например с версии 1.19 ты можешь использовать профили для scheduler. Они позволяют подключать plugins для default-scheduler, которые переопределяют стандартные веса алгоритма и запуска дополнительных schedulers может не потребоваться.

Я привожу ссылки на документацию, где это все описано. И это все о конфигурировании кастомного планировщика, увидимся в следующей лекции.