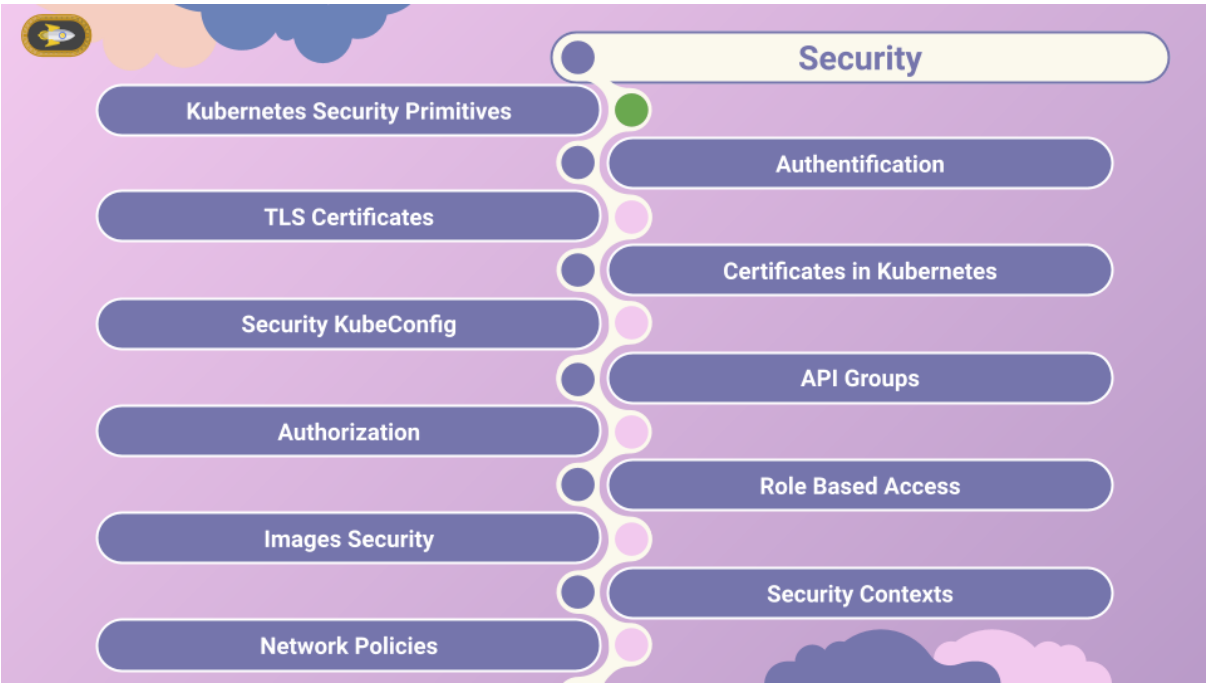


Привет и добро пожаловать на лекцию. Мы сделаем общий обзор подходов к безопасности в Kubernetes.

Kubernetes - это платформа для размещения grid-приложений промышленного уровня. Безопасность здесь, что называется Job #0. Она имеет первостепенное значение.

В этой лекции мы рассмотрим различные примитивы безопасности в Kubernetes на высоком уровне, прежде чем углубиться в них в следующих лекциях.

Начнем с хостов, которые формируют сам кластер.

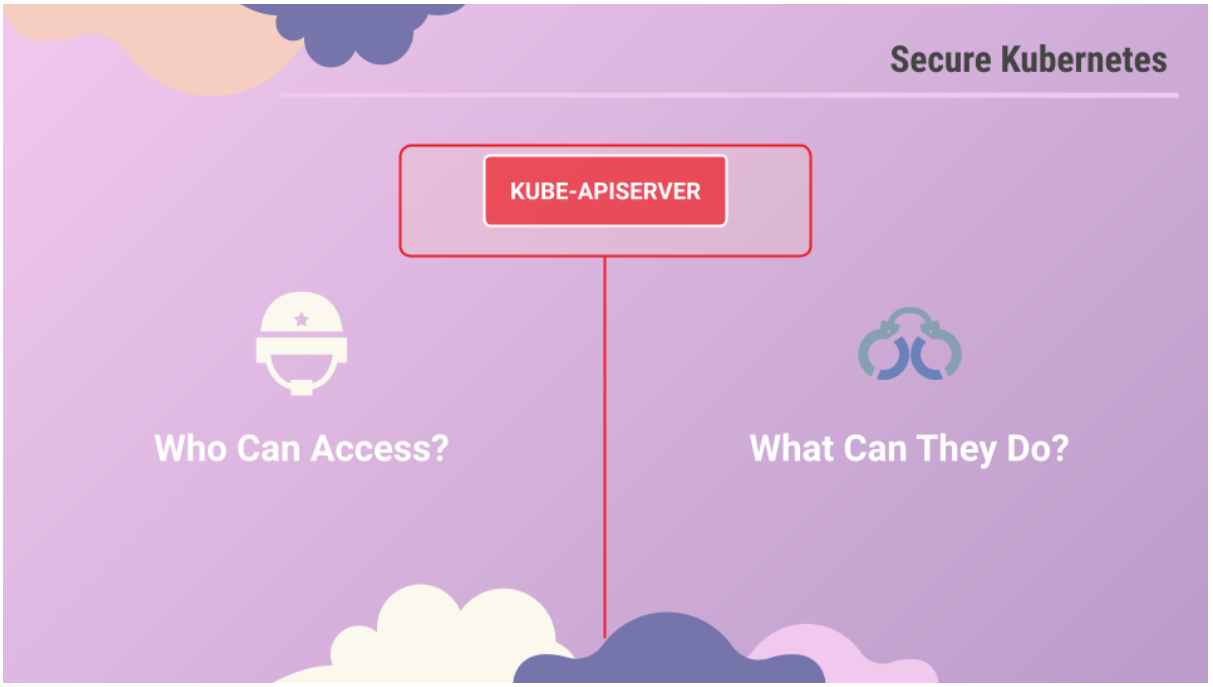
Конечно, весь доступ к этим хостам должен быть защищен, это базовые вещи, вроде отключенного root-доступа, отсутствие аутентификации на основе пароля, реализация аутентификации посредством SSH-ключа.

И, конечно же, все другие меры из арсенала безопасника, которые необходимо предпринять для защиты нашей физической или виртуальной инфраструктуры, в которой размещен Kubernetes. Как ты знаешь, если скомпрометирован базовый уровень, все уровни выше тут же становятся скомпрометированными.

В этом разделе мы уделим больше внимания безопасности, связанной с Kubernetes. Каковы риски и какие меры необходимо предпринять для защиты кластера.

Как мы уже видели, сервер kube-api находится в центре всех операций в Kubernetes. Мы взаимодействуем с ним через утилиту kubectl или напрямую обращаясь к API, благодаря чему можем выполнять практически любую операцию в кластере.

Так что это первая линия защиты. Контроль доступа к самому серверу API.



Нам нужно принять два типа решений:

- кто может получить доступ к кластеру
- и что они могут делать

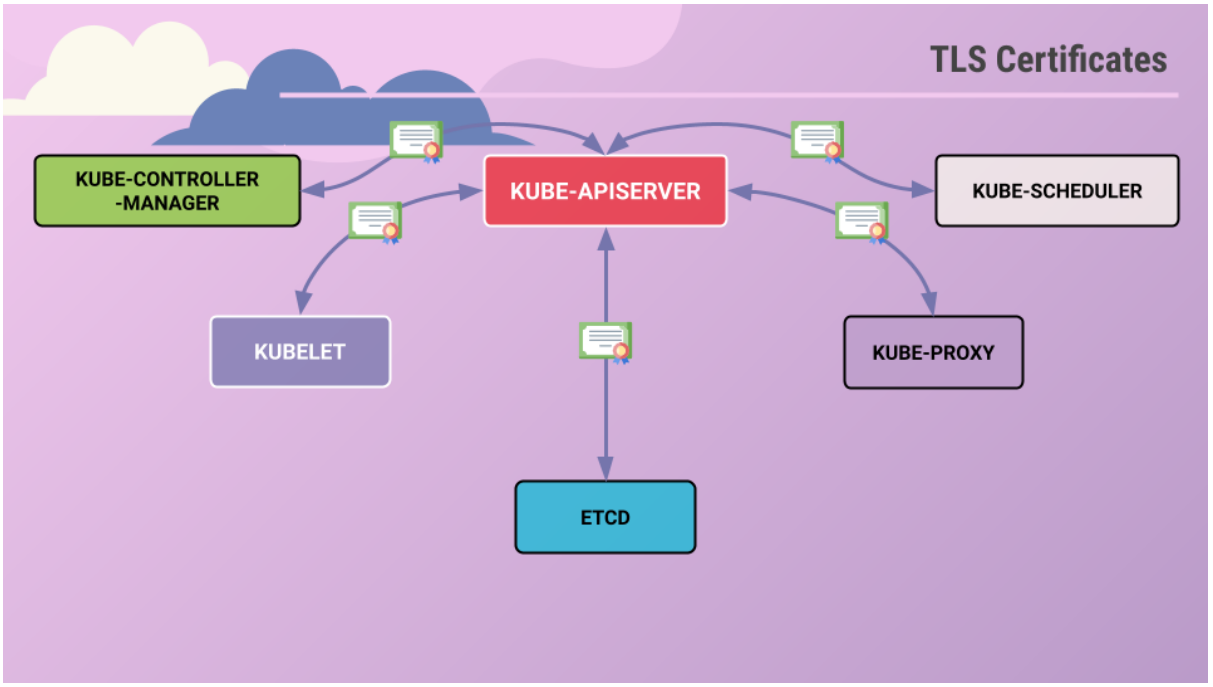
Кто может получить доступ к серверу API, определяется механизмами аутентификации. Существуют разные способы аутентификации на сервере API.

Начиная с идентификаторов пользователей и паролей, хранящихся в статическом файле, до токенов, сертификатов или даже интеграции с внешними поставщиками аутентификации, такими как LDAP. Наконец есть механизмы для аутентификации служебных сущностей, которые не являются людьми, но им также необходимо работать с kube-apiserver и их требуется удостоверять. Мы рассмотрим это более подробно в следующих лекциях.

Далее, мы понимаем, кто это, мы даем ему доступ к кластеру, и его действия определяются механизмами авторизации.

Авторизация реализуется с помощью управления доступом на основе ролей (RBAC), когда пользователи связаны с группами с определенными разрешениями.
Кроме того, существуют другие модели авторизации, такие как управление доступом на основе атрибутов (ABAC), авторизация узла, веб-хуки и т. д.

Мы рассмотрим их более подробно в следующих лекциях. Вся связь с кластером между различными компонентами, такими как кластер ETCD, kube-controller-manager, scheduler, API-сервер, а также те, которые работают на рабочих узлах, таких как kubelet и kube-proxy, защищаются с помощью TLS-шифрования.



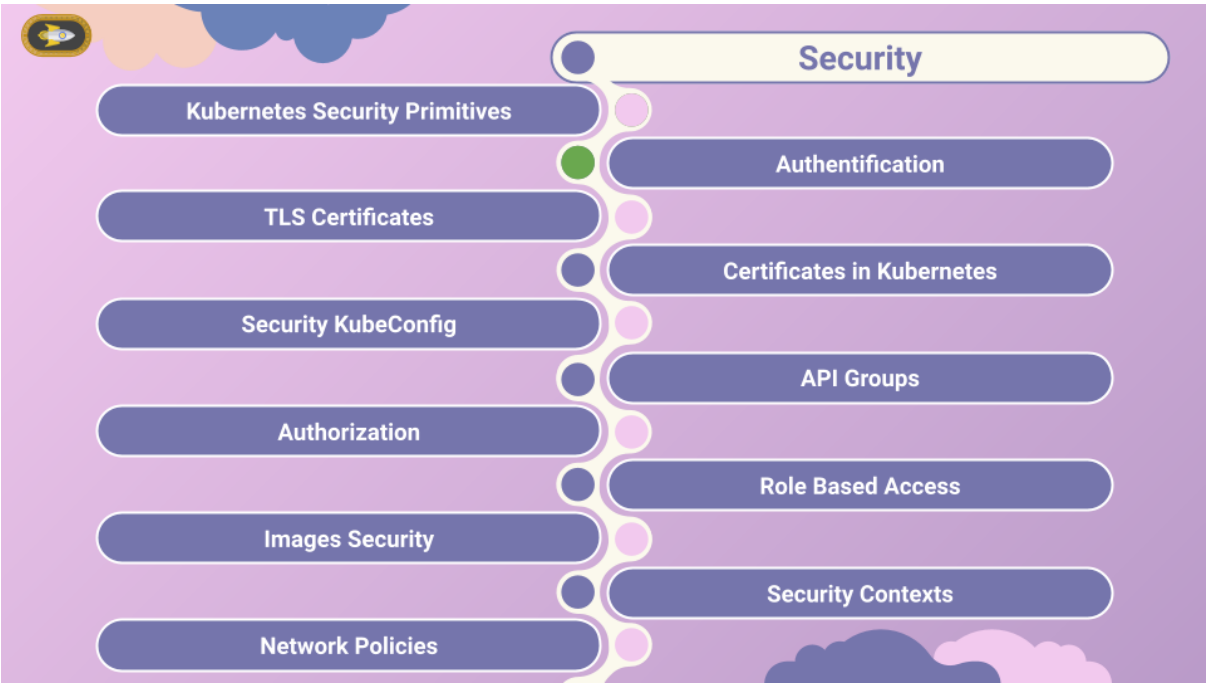
У нас есть серия лекций в этом разделе, специально предназначенная для этого, где мы обсуждаем и практикуем настройку сертификатов между различными компонентами.

Что насчет связи между приложениями внутри кластера?

По умолчанию все PODs имеют доступ ко всем остальным PODs в кластере.
Мы можем ограничить доступ между ними с помощью сетевых политик (net policies).
Мы посмотрим и попробуем, как именно это делается позже в разделе net policies.

Это был вертолетный взгляд на различные подходы и примитивы безопасности в Kubernetes. В дальнейшем мы рассмотрим их более подробно, погружаясь в эти темы в различных лекциях.

Жду тебя на них.



Привет и добро пожаловать на лекцию об аутентификации в кластере Kubernetes

Как мы уже не раз видели и знаем, кластер состоит из нескольких узлов, физических или виртуальных, и компонентов, которые работают вместе этих узлах.

Разные группы пользователей взаимодействуют с кластером по-разному:

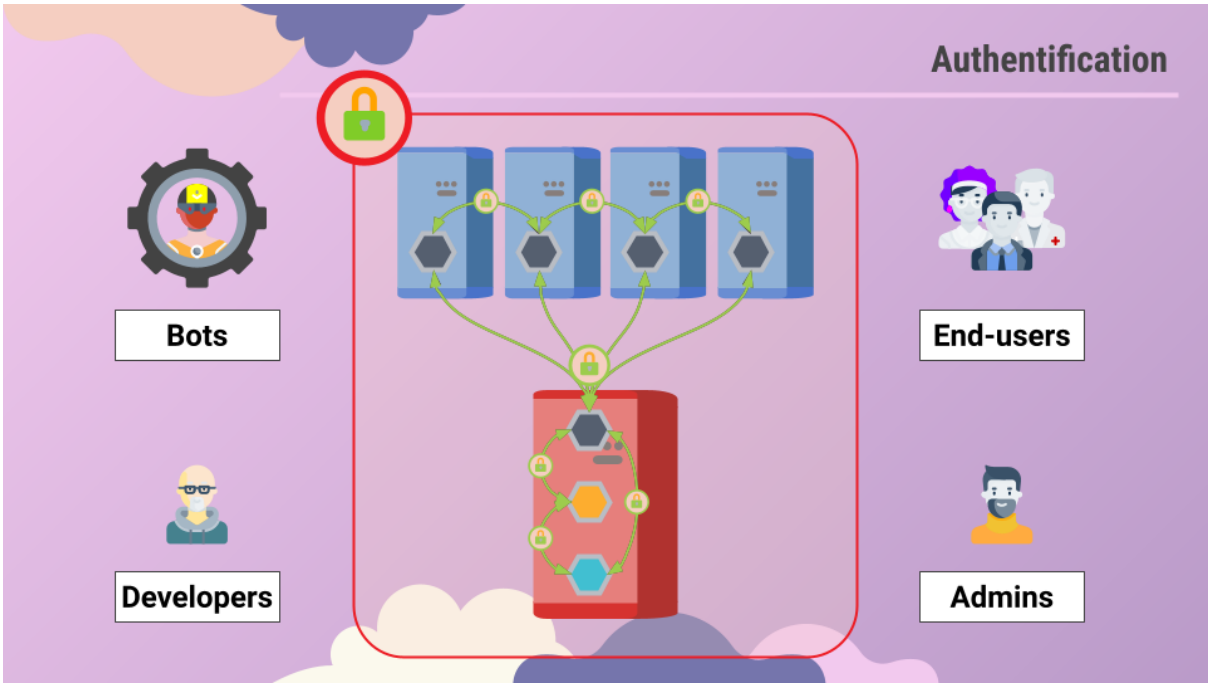
- администраторы обращаются к кластеру для выполнения административных задач
- разработчики обращаются к кластеру, больше для тестирования или развертывания приложений
- конечные пользователи, обращаются к приложениям, развернутым в кластере
- сторонние приложения, обращающиеся к кластеру в целях интеграции

На протяжении нескольких лекций мы будем говорить о том, как защитить наш кластер, обеспечив безопасность связей между внутренними компонентами и правильно реализовать управленческий доступ к кластеру с помощью механизмов аутентификации и авторизации.

Сейчас наше внимание сосредоточено на обеспечении доступа к кластеру с помощью механизмов аутентификации.

Итак, мы говорили о различных пользователях, которые могут получать доступ в кластер.

Безопасность конечных пользователей, которые взаимодействуют с приложениям, развернутыми в кластере, обеспечивается самими приложениями, это их внутренняя кухня, и мы исключим это из нашего обсуждения.



Наше внимание будет сосредоточено на доступе пользователей к кластеру для административных целей.

Таким образом, у нас остается два типа пользователей: люди, такие как администраторы и разработчики, плюс роботы, т.е. другие процессы, службы или приложения, которым требуется доступ к кластеру.

Kubernetes нативно не управляет учетными записями пользователей, он полагается на внешний источник, такой как файл с данными пользователя, или сертификаты, или стороннюю службу идентификации, вроде LDAP, для управления этими пользователями.

Таким образом, у тебя не получится создавать пользователей в кластере или просматривать список пользователей как ты видишь здесь.

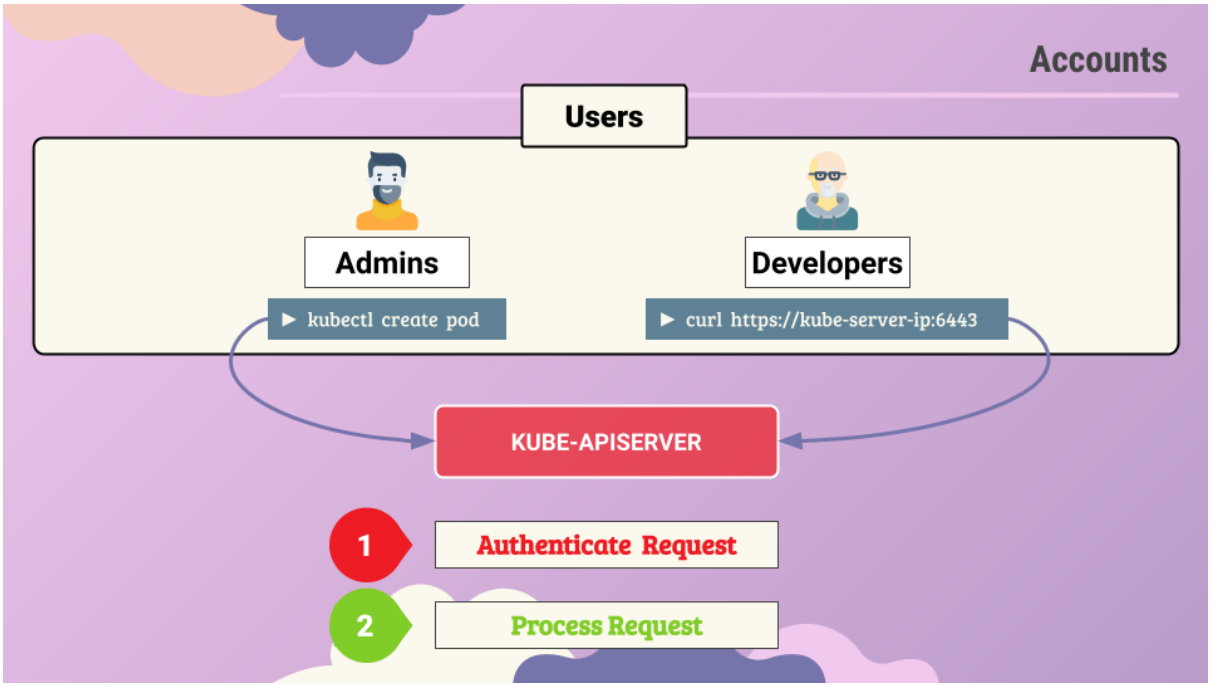
Однако, в случае служебных учетных записей - сервисных аккаунтов Kubernetes может ими управлять.

Мы можем создавать и управлять serviceaccounts помощью Kubernetes API.

Разговор о serviceaccounts это больше о разработке, чем об администрировании, и мы более подробно рассмотрим их в курсе CKAD.

В этой лекции мы сосредоточимся на пользователях в Kubernetes. Весь доступ пользователей в кластер управляется API-сервером, независимо от того, обращаемся ли мы к кластеру через инструмент kubectl или напрямую через API, все эти запросы проходят через kube-apiserver.

Сервер API аутентифицирует запрос перед его обработкой.



Так как же производится эта аутентификация на сервере API?

Можно настроить различные механизмы аутентификации.

Ты можешь иметь список имен пользователей и паролей в статическом файле паролей или имена пользователей их временные токены в статическом файле токенов, или ты можешь использовать аутентификацию с помощью сертификатов. Еще один вариант - настроить подключение к сторонним протоколам аутентификации, таким как Kerberos, OpenID и т. д.

Мы рассмотрим некоторые из них позже.

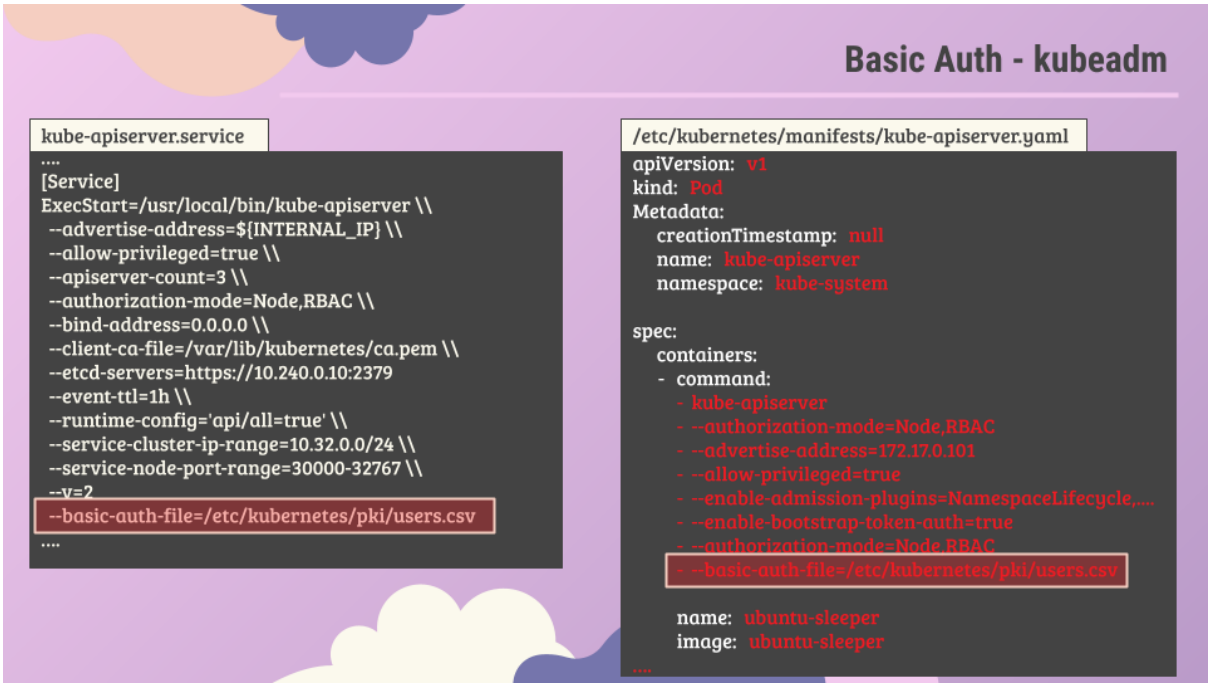
А начнем со статических файлов паролей и токенов, так как они самые простые для понимания.

Ок, простейшая форма аутентификации: мы создаем список пользователей и их пароли в форме простого csv-файла и используем его в качестве источника информации о пользователях.

В этом файле у нас три столбца: пароль, имя пользователя и его ID.

Чтобы это использовать, нам нужно сообщить kube-apiserver об этом файле с помощью опции запуска `--basic-auth-file=` а делее путь к файлу.

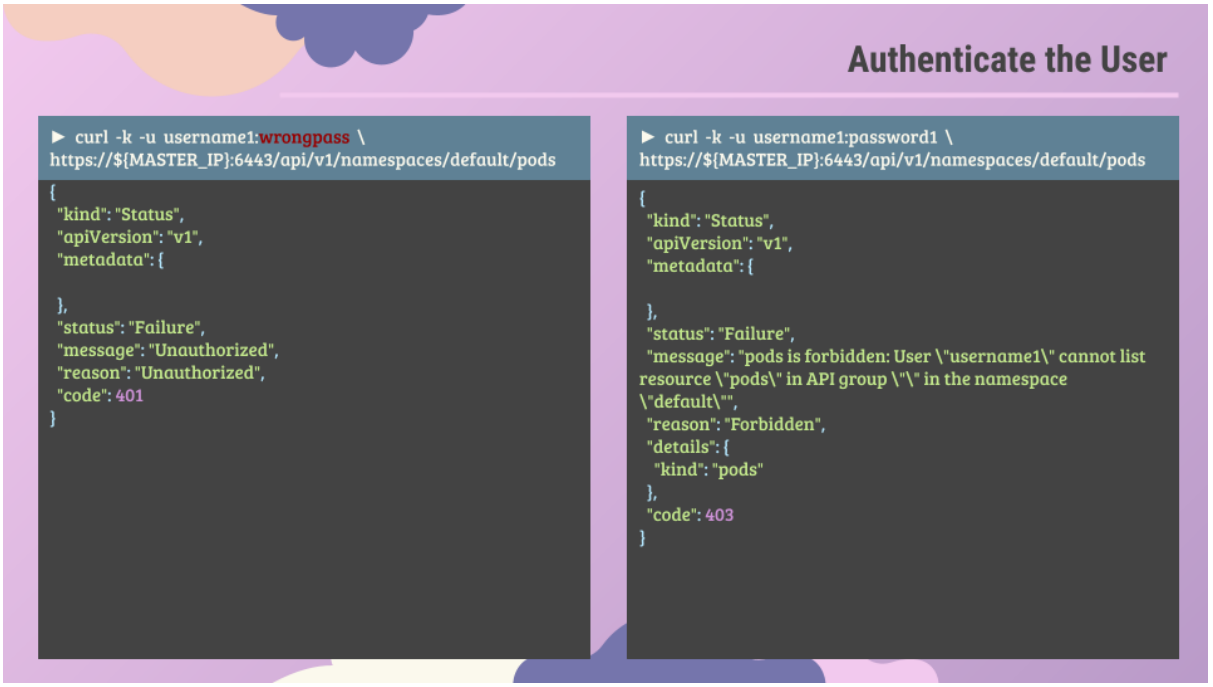
Этот файл службы нам уже знаком, ранее в курсе мы уже видели его. Именно сюда нам и следует добавить эту опцию, чтобы API-сервер запустился с ней. После редактирования перезапустим службу kube-api, чтобы изменения стали актуальными.



Если мы настраивали свой кластер с помощью kubeadm, то нужно изменить файл определения POD kube-apiserver. Kubelet автоматически перезапустит сервер API после обновления этого файла.

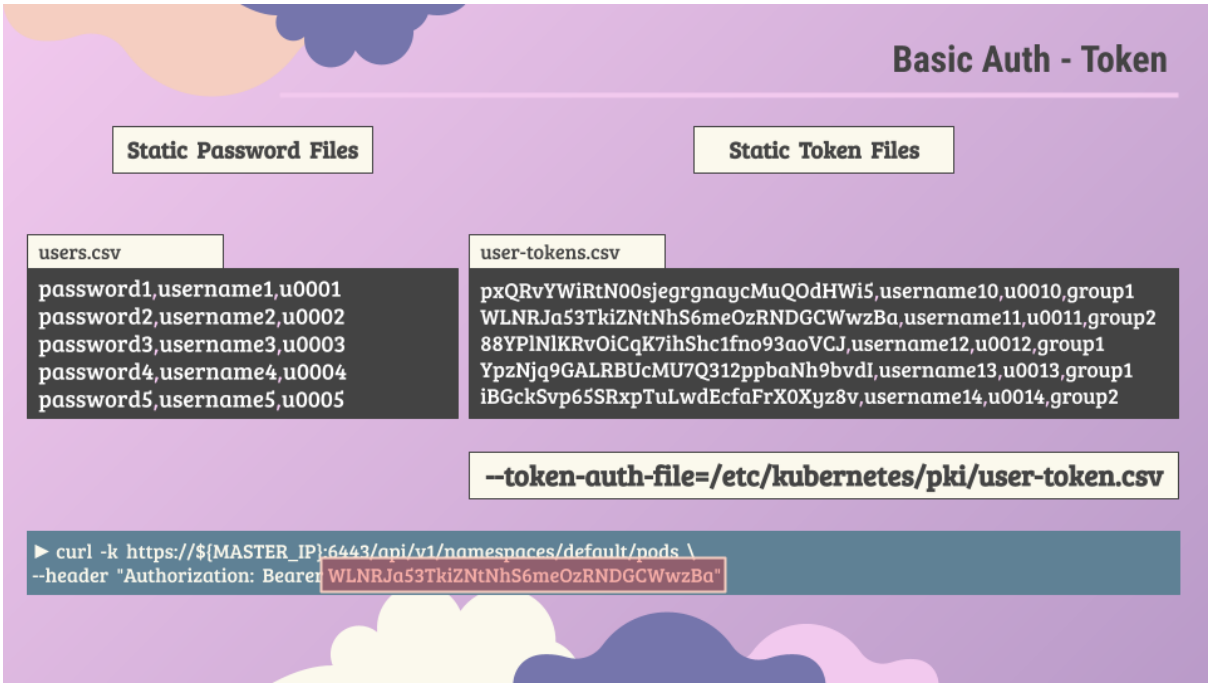
Небольшая особенность сетапа с kubeadm. Статические файлы со списками паролей должны быть видны из контейнера с kube-apiserver. Именно поэтому я положил этот файл в подключенной к контейнеру директории /etc/kubernetes/pki.

На самом деле не играет значения откуда файл, тебе просто нужно добавить в манифест дополнительный volume, где лежит твой файл. В презентации я выложу набор манифестов, где покажу как смонтировать список паролей из своей директории и авторизовать этого пользователя на работу в кластере.



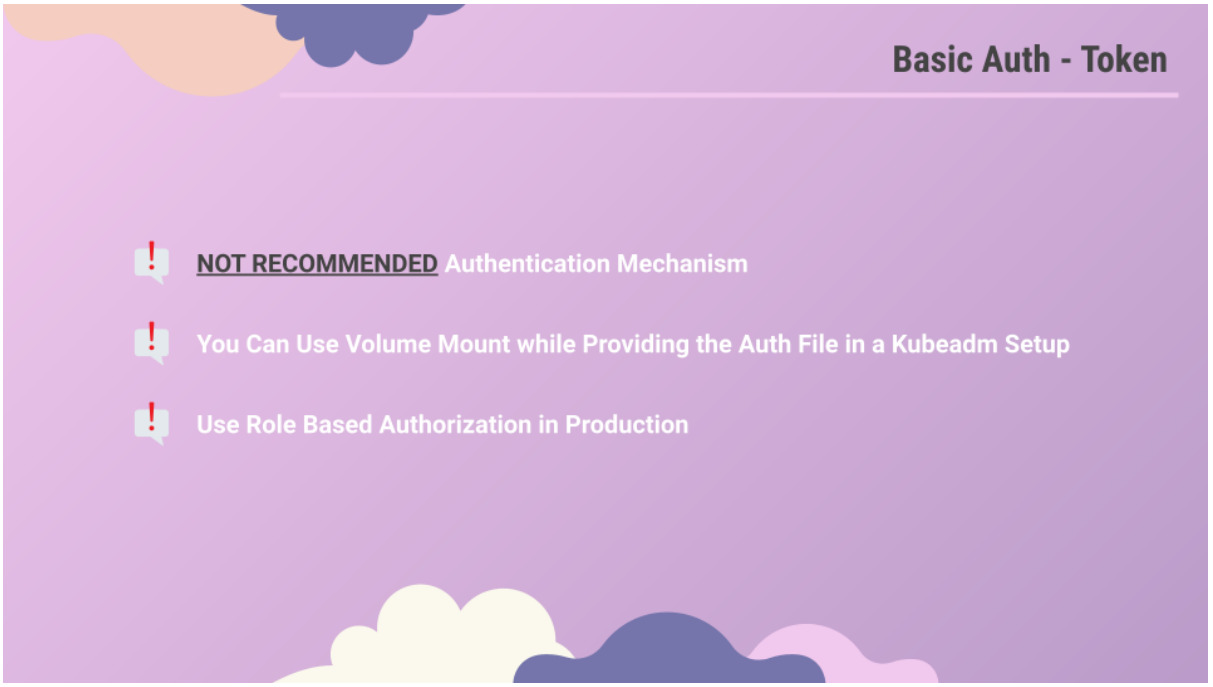
Для аутентификации с использованием basic credentials укажи в запросе к серверу API имя пользователя и пароль в такой команде, как ты видишь. Здесь я специально указал неверный пароль и получил 401 ошибку. В следующей команде нас пустило в кластер, но все же ответ - 403 ошибка. Здесь дело в авторизации, т.к. у пользователя `username1` не хватает прав на просмотр PODs в этом namespace. Об авторизации мы поговорим в одной из следующих лекций.

В файле Kubernetes с данными о пользователях, который мы видели, мы можем дополнительно иметь четвертый столбец с данными о группах, чтобы назначать пользователей в определенные группы.



Точно так же вместо файла статических паролей ты можешь иметь статический файл токенов. Как видишь, вместо пароля у нас указан токен, далее структура файла сходная. При запросе файл токена передается в виде заголовка запроса в таком виде.

Кроме статических токенов в Kubernetes существует также внутренний механизм Bootstrap Tokens, это динамические жетоны, они используются при присоединении нод. Пожалуйста различай эти две истории.



Давай подытожим о basic authentication:

- это механизм аутентификации, который хранит имена пользователей, пароли, токены как открытый текст в статическом файле. Это не является рекомендуемым подходом, поскольку он небезопасен. Также с версии 1.19 он стал устаревшим.

На самом деле это самый простой способ понять основы аутентификации в Kubernetes, а как все это происходит в реальном мире, мы рассмотрим далее.

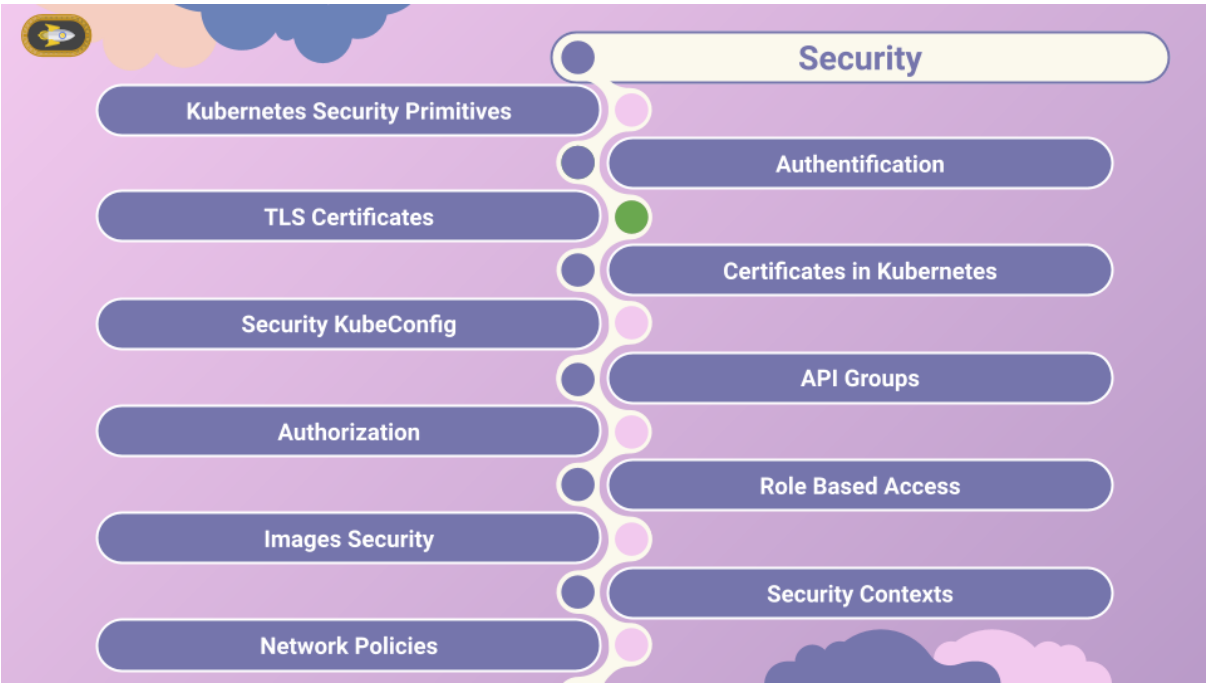
- Еще раз отмечу, что если ты все же выбрал такую аутентификацию, не забудь прокинуть тома в контейнер kube-apiserver, если устанавливаешь с помощью инструмента kubeadm.

Ну и как я уже иллюстрировал, требуется настроить авторизацию для новых пользователей и отменить анонимные запросы.

- Хорошее решение - использовать RBAC.

Мы обсудим авторизацию позже в этом курсе в следующих лекциях, еще мы обсудим аутентификацию на основе сертификатов и то, как различные компоненты в этом кластере защищены с помощью сертификатов.

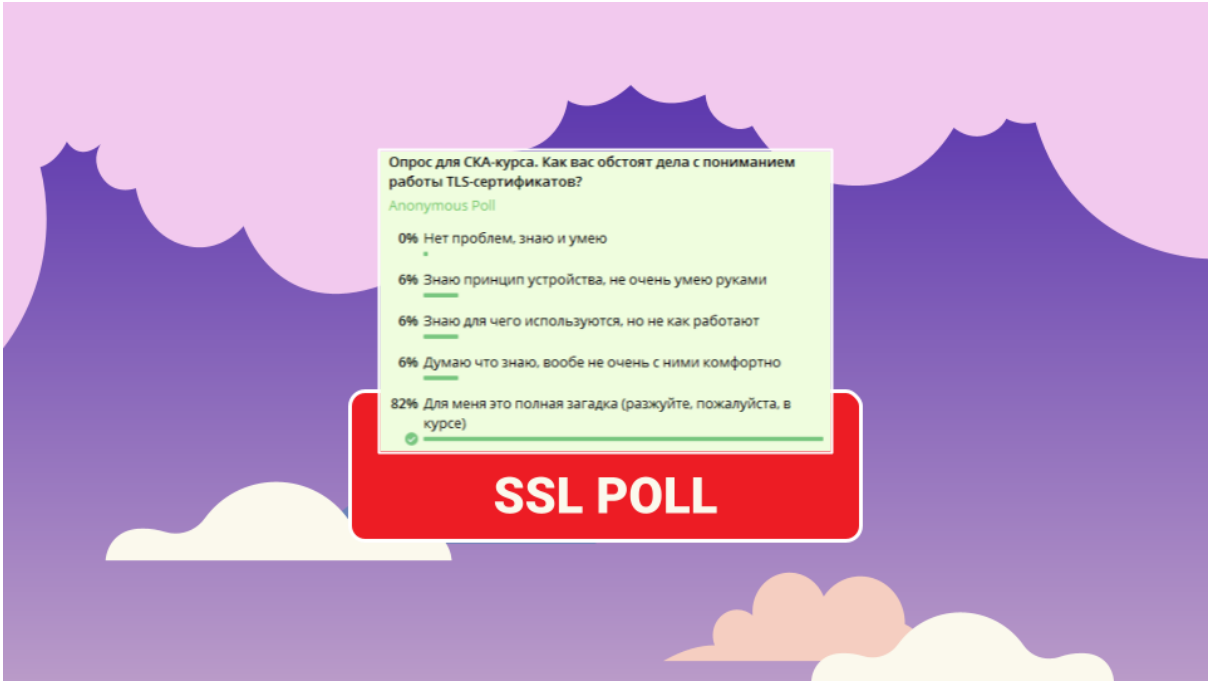
А на этом лекция закончилась. Жду тебя в следующей.



Привет и добро пожаловать на вводную лекцию о сертификатах TLS в Kubernetes.

Защита кластера с помощью TLS и траблешутинг связанный с TLS, могут быть особенно сложными, если ты не знаком или только думаешь, что знаком с основами TLS-сертификатов.

И я бы сказал, что не зная, как работает база TLS, невозможно погрузиться в понимание большинства процессов в области cybersecurity.



Во время создания этого курса я устроил опрос в нашей телеграм-группе, чтобы собрать данные о ваших знаниях в области сертификатов.

Большинство из участников группы решили, что сертификаты для них - полная загадка или же с ними не комфортно. В связи с этим я решил добавить материала в лекции о сертификатах TLS. Думаю эта серия лекций поможет тебе получить достаточно знаний для работы с сертификатами в KUbernetes.

И, конечно же, эти знания помогут и в другом, т.к. механизмы удостоверения с помощью сертификатов используются повсеместно.

Если ты знаком с базовыми вещами, не стесняйся пропустить эти лекции и сразу переходить к тем, которые имеют отношение к Kubernetes.

К концу этого раздела ты должен хорошо понимать, как работают сертификаты в целом, а также с как это устроено в Kubernetes. Также ты сможешь настраивать и устранять проблемы, связанные с сертификатами.

Но единственный способ сделать это, изучить и понять, как это работает, от начала до конца. А если не понял - просмотреть и прорешать команды из слайдов заново.

Итак, такими вещами мы займемся далее. И начнем с основ сертификатов в целом.

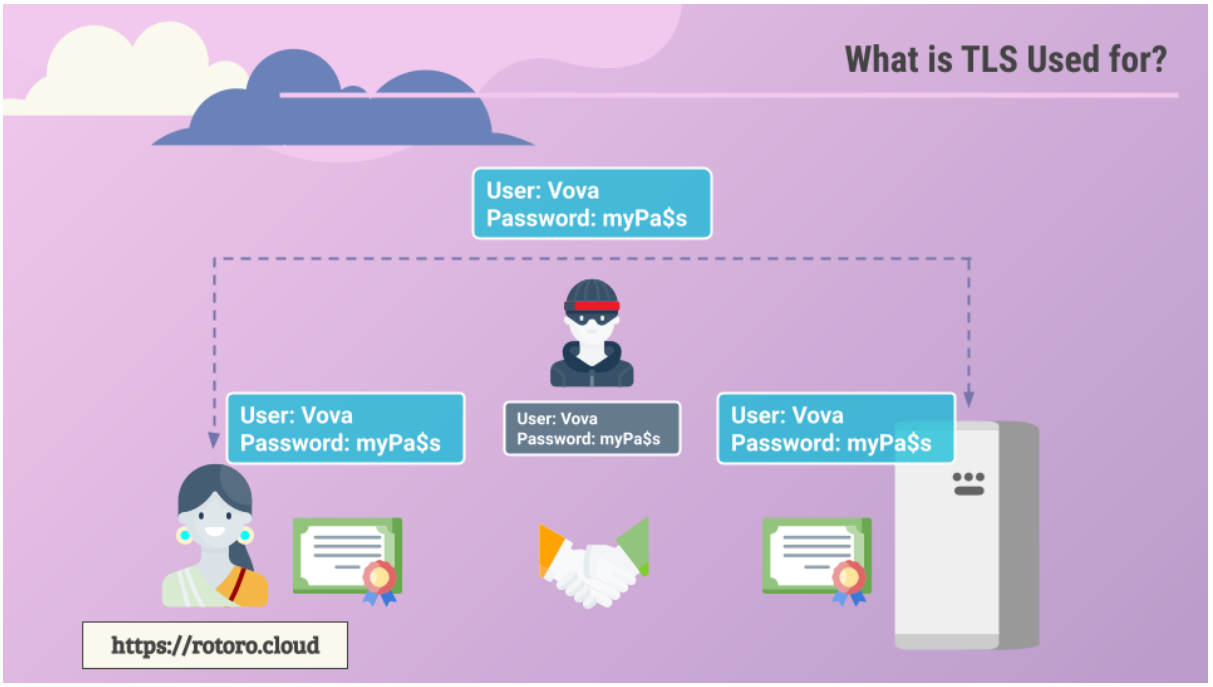
Если ты уже знаком с темами, которые ты здесь видишь, просто пропусти эту лекцию и перейти к следующей, где мы обсудим, как это живет Kubernetes.

Жду тебя на лекции.

Привет, и добро пожаловать в лекцию о понимании сертификатов SSL/TLS.

В этом видео мы рассмотрим основы того, что такое сертификаты TLS, зачем они нам нужны и как настраиваются сертификаты для защиты SSH или веб-серверов.

Сертификат используется для гарантии доверия между двумя сторонами во время транзакции. Например, когда пользователь пытается получить доступ к веб-серверу, сертификаты TLS гарантируют, что связь между пользователем и сервером зашифрована, а сервер является тем, кем он является.



Давай посмотрим на сценарий. Если бы пользователь онлайн-курсов rorotoro.cloud вводил свои логин и пароль без безопасного подключения, то введенные учетные данные будут отправлены в текстовом формате. Хакер, настроивший перехватчик сетевого трафика может легко получить логин и пароль и использовать их для взлома аккаунта пользователя.

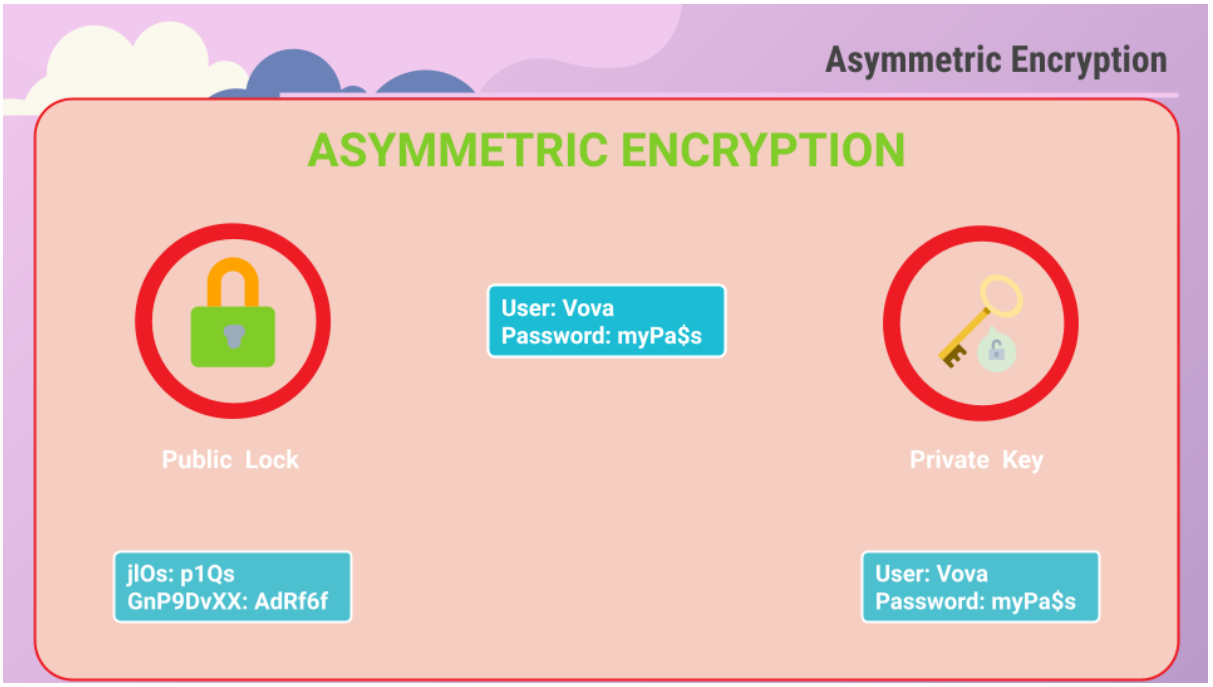
Что ж, это явно небезопасно, поэтому мы должны зашифровать передаваемые данные с помощью ключей шифрования.

Данные зашифрованы с помощью ключа, который в основном представляет собой набор случайных чисел и букв.

Мы добавляем случайное число к своим данным и шифруем их в формате, который невозможно распознать. Затем данные отправляются на сервер. Хакер, снифящий сеть, получает данные, но ничего не может с ними поделать. Однако то же самое и в случае с сервером, получающим данные. Он не может расшифровать данные без ключа. Поэтому копия ключа также должна быть отправлена на сервер, чтобы сервер мог расшифровать и прочитать это сообщение.

Поскольку ключ также пересылается по этой же самой незащищенной сети, злоумышленник может его перехватить и расшифровать с помощью него данные.

Это называется симметричным шифрованием. Это безопасный способ шифрования, но поскольку он использует один и тот же ключ для шифрования и дешифрования данных, следовательно, сторонам нужно им обменяться и появляется риск того, что хакер получит доступ к ключу и расшифрует данные.



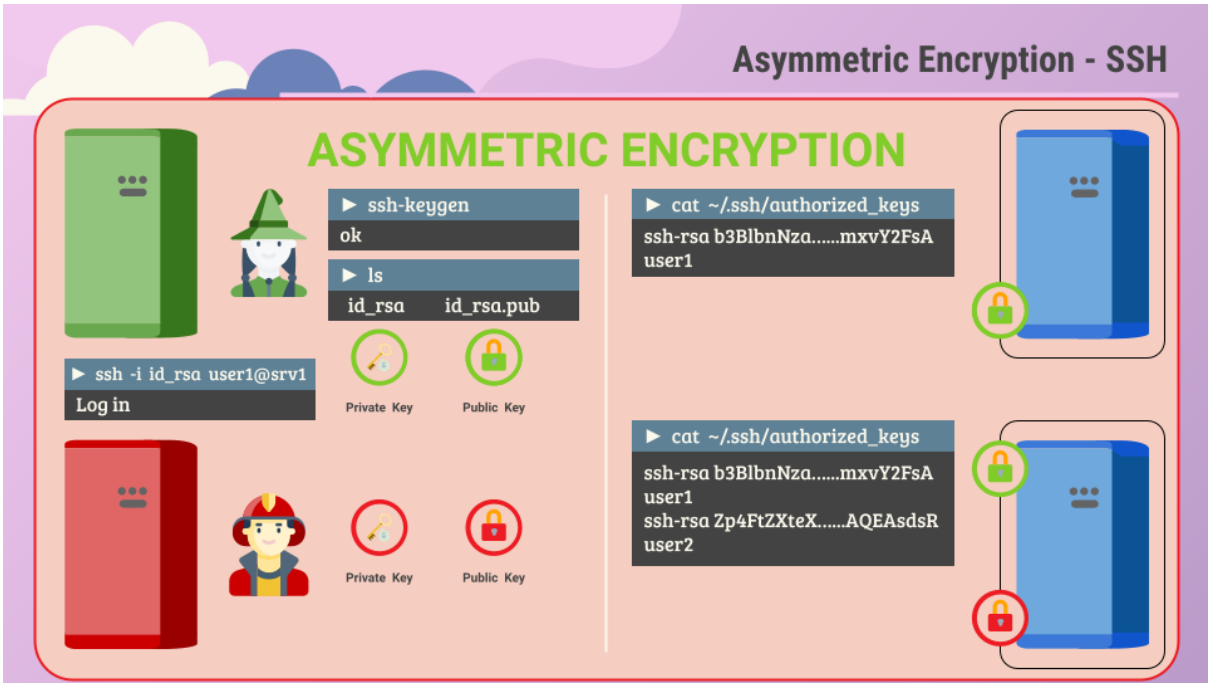
Вот здесь появляется асимметричное шифрование. Вместо использования одного ключа для шифрования и дешифрования данных асимметричное шифрование использует пару ключей, закрытый ключ и открытый ключ - private and public keys.

Давай для лучшего понимания в примере будем называть public key - публичным замком, а pirvate key - частным ключом или закрытым. Т.е. пока думай об этом как о замке и ключе, что у тебя в кармане. Этот ключ, есть только у тебя, поэтому он частный, а замок висит на улице, к нему любой может получить доступ, поэтому он общедоступен - публичен. И если ты захлопнешь дверь и защелкнешь замок, то открыть сможешь лишь только с помощью ключа из своего кармана.

В ассиметричном шифровании такой же смысл - если ты зашифруешь данные с помощью своего замка, то получить доступ к ним можно лишь при помощи соответствующего ключа - частного ключа. Этот ключ всегда должен быть в безопасности и никому не передаваться, он закрытый - твой частный, но замок является общедоступным и может быть передан другим.

Они могут запереть что-то на этот замок, т.е. только зашифровать данные с его помощью. И независимо от того, что заперто на публичный замок, кто повесил замок уже не сможет заглянуть в ящик, лишь обладатель ключа всегда сможет его открыть и увидеть, что внутри.

Прежде чем мы вернемся к нашему примеру с веб-сервером, давай рассмотрим еще более простой вариант использования защиты SSH-доступа к серверам с помощью пар ключей.



В нашей среде есть сервер, к которому потребовался доступ. Мы не хотим использовать пароли, так как они слишком опасны, поэтому мы решили использовать ключевые пары.

Итак, мы создаем пару: открытый и закрытый ключ. Это можно сделать, выполнив команду ssh-keygen. Он создает два файла: `id_rsa` - это закрытый ключ и `id_rsa.pub` - это открытый ключ, ну, как мы договорились, не открытый ключ, а публичный замок.

Затем ты защищаешь свой сервер, блокируя весь доступ к нему, кроме двери, которая запирается на этот наш общедоступный замок. Обычно это делается путем добавления записи с нашим открытым ключом в файл `.ssh/authorized_keys` на сервере.

Как видишь, замок висит на общем обозрении и любой может попытаться прорваться вовнутрь. Но до тех пор, пока у него не будет нашего закрытого ключа, который находится в безопасности в файле на твоем компьютере, никто не сможет получить доступ к серверу.

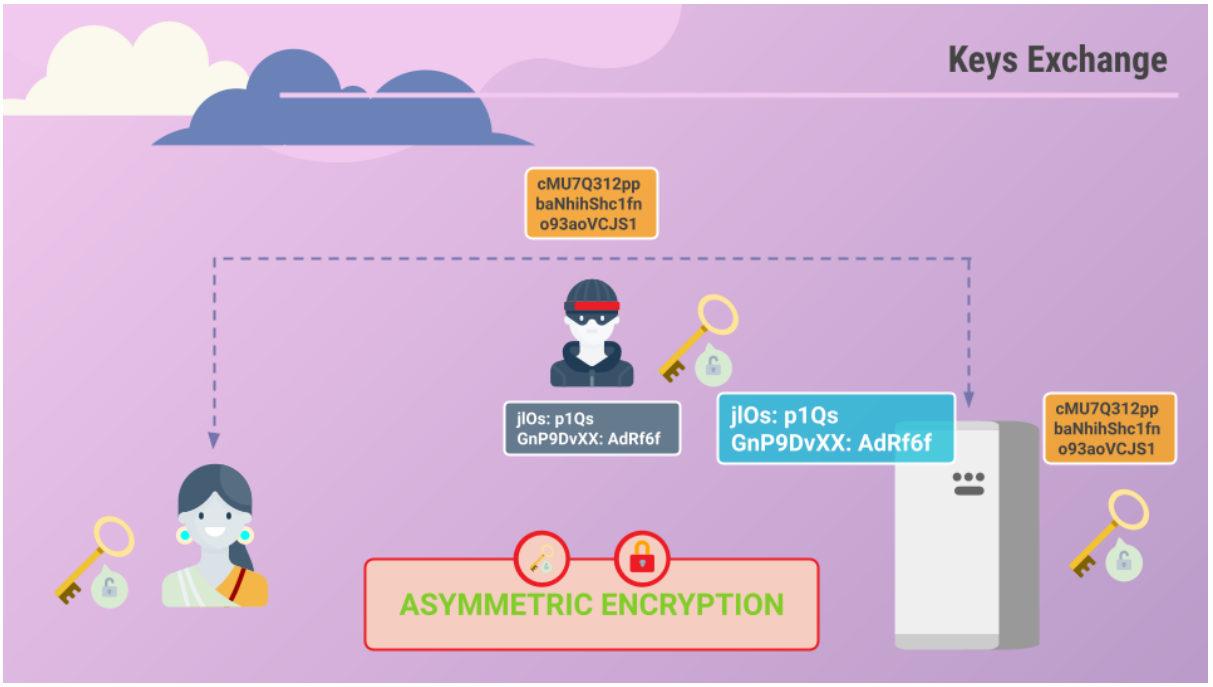
Когда ты используешь SSH, ты указываешь местоположение своего частного ключа в своей команде SSH с помощью опции `-i`.

Что делать, если у тебя в среде есть другие серверы? Как защитить несколько серверов с помощью пар ключей?

Ну можно создать копии своих public keys и разместить их на любом количестве серверов. Ты сможешь безопасно использовать один и тот же закрытый ключ для SSH на всех своих серверах. А что делать, если другим пользователям нужен доступ к этим серверам?

Что ж, они могут делать то же самое. Они могут генерировать свои собственные пары открытого и закрытого ключей.

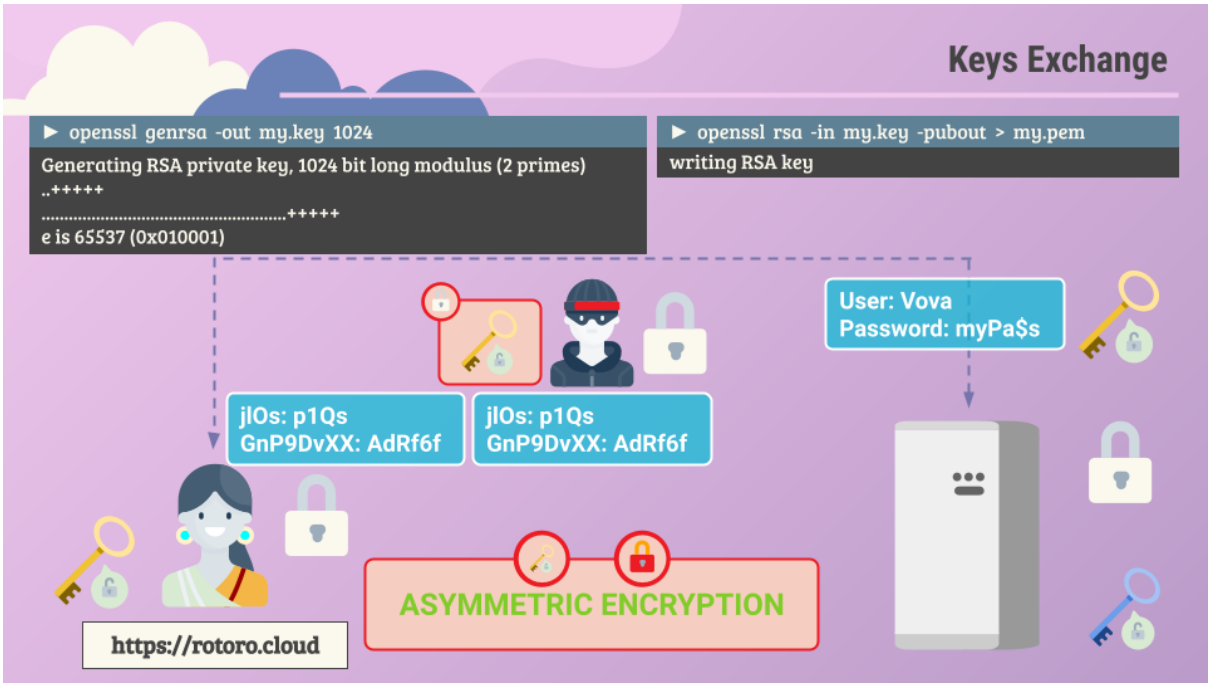
Как пока единственный человек, имеющий доступ к этим серверам, ты мог бы создать для них дополнительную дверь и заблокировать ее публичным замком. Скопируй публичные замки этих пользователей на все серверы, и теперь пользователи смогут общаться с серверами, используя свои закрытые ключи.



Теперь вернемся к нашему примеру с веб-сервером. Как ты не успел забыть, проблема, с которой мы столкнулись ранее с симметричным шифрованием, заключалась в том, что ключ, используемый для шифрования данных, должен был быть отправлен на сервер по сети вместе с зашифрованными данными, и поэтому был риск того, что хакер получит ключ для дешифрования данных.

Что бы мы могли предпринять, чтобы каким-то образом безопасно отправить ключ на этот сервер?

Для безопасной передачи симметричного ключа от клиента к серверу, мы используем асимметричное шифрование. И как только ключ будет безопасно предоставлен серверу, сервер и клиент начнут обмен данными друг с другом, используя симметричное шифрование.



Итак, мы генерируем пару открытого и закрытого ключей на сервере. Теперь, когда ты уже немного в теме, аналогия

с замком уже не нужна, и я буду называть публичный замок открытым ключом.

Команда ssh-keygen ранее использовалась для создания пары ключей для SSH, поэтому формат немного отличается.

Здесь мы используем команду openssl для генерации пары закрытого и открытого ключей, и вот так они выглядят.

Когда пользователь впервые обращается к веб-серверу с помощью HTTPS, он получает открытый ключ с сервера.

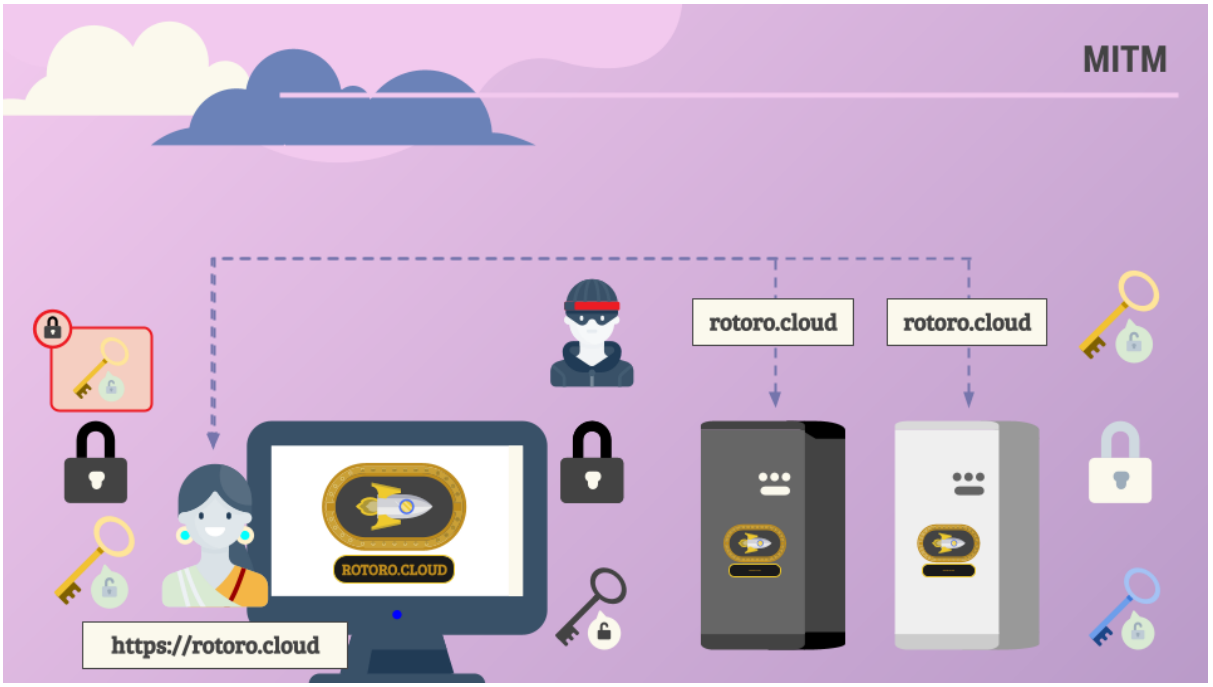
Поскольку хакер перехватывает весь трафик, предположим, что он тоже получает копию открытого ключа. Посмотрим, что он сможет с этим сделать.

Пользователь, по сути, браузер пользователя затем шифрует симметричный ключ, используя открытый ключ, предоставленный сервером. Симметричный ключ теперь безопасен. Затем пользователь отправляет это на сервер. Хакер также получает копию. Сервер использует закрытый ключ для расшифровки сообщения и извлечения из него симметричного ключа. Однако у хакера нет закрытого ключа для расшифровки и извлечения симметричного ключа из полученного сообщения. У хакера есть только открытый ключ, с помощью которого он может только запереть, т.е. зашифровать сообщения, но не расшифровать.

Симметричный ключ теперь безопасно доступен только пользователю и серверу. Теперь они могут использовать симметричный ключ для шифрования данных и отправки их друг другу. Получатель может использовать тот же симметричный ключ

для расшифровки данных и получения информации. Хакеру остаются зашифрованные сообщения и открытые ключи, с помощью которых он не может расшифровать никакие данные.

С помощью асимметричного шифрования мы успешно передали симметричные ключи от пользователя на сервер, а что касается симметричного шифрования, мы обеспечили безопасность всей будущей связи между ними. Идеально.



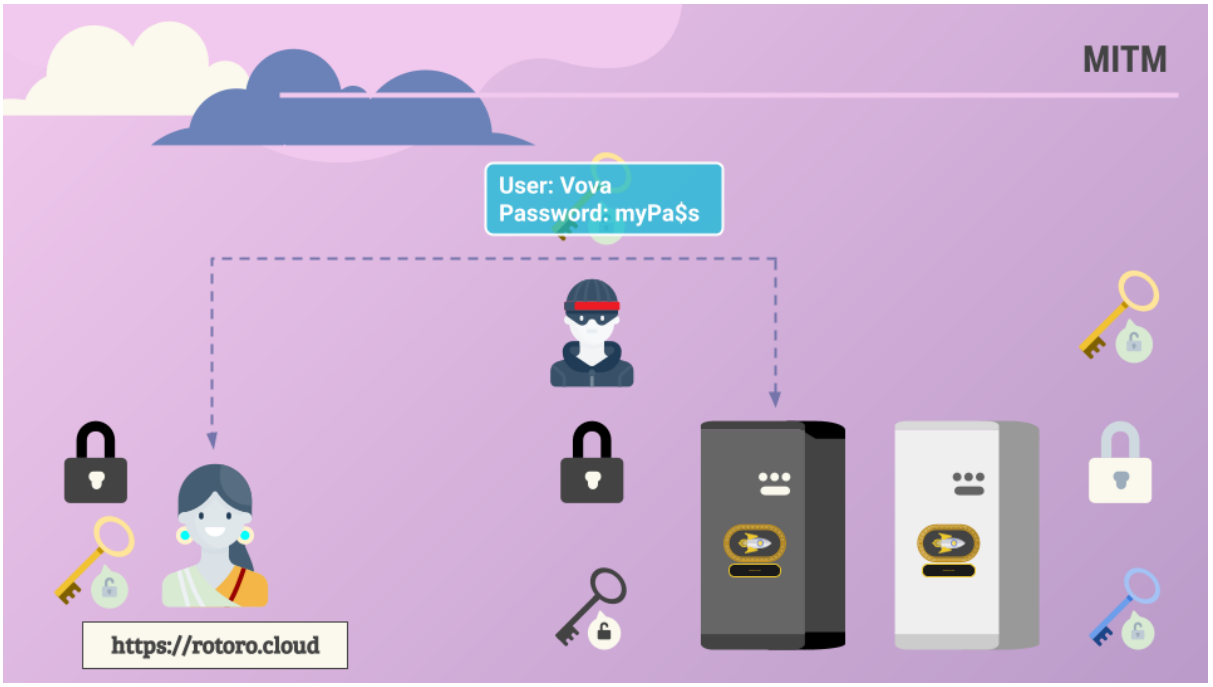
Теперь хакер ищет новые способы взломать учетную запись пользователя, и он понимает, что единственный способ получить учетные данные - это заставить ввести их в форму, которую представит он сам, т.е. обмануть пользователя, который будет уверен, что вводит данные на правильном сайте.

Поэтому он создает веб-сайт, который выглядит точно так же, как и наш сайт курсов. Дизайн веб-сайта такой же, графика такая же. Веб-сайт является точной копией реального веб-сайта rotoro.cloud. Он размещает сайт на своем собственном сервере.

Ему нужно, чтобы пользователь тоже думал, что это безопасно, поэтому он генерирует свой собственный набор пар открытых и закрытых ключей и настраивает их на своем веб-сервере.

Наконец, ему каким-то образом удается настроить среду пользователя или его сеть, чтобы направить запрос, идущий на веб-сайт курсов, на свой черный сервер.

Когда пользователь открывает браузер и вводит адрес веб-сайта, то видит очень знакомую страницу. Та же самая страница входа в аккаунт, к которой но уже привык.



Итак, прежде чем пользователь введет свои имя и пароль, ему следует убедиться, что в URL-адресе присутствует https. Это значит, что связь безопасна и зашифрована.

Далее браузер получает открытый ключ. Шифрует и отправляет свой симметричный ключ, а затем отправляет свои учетные данные, зашифрованные с помощью этого ключа. Получатель дешифрует учетные данные тем же симметричным ключом.

Все хорошо, за исключением, того что мы вели безопасный обмен данными в зашифрованном виде с сервером хакера.

Как только пользователь отправит свои учетные данные, появится раздел личного кабинета, который уже не очень похож на привычную панель управления. Ведь задача хакера была только сделать парадный фасад сайта, и но добился своего - получил логин и пароль.

А что, если бы мы могли посмотреть на ключ, полученный с сервера, и проверить, является ли он легитимным ключом с сервера rotoro.cloud?

Когда сервер отправляет ключ, он отправляет не просто ключ, он отправляет сертификат, внутри которого находится ключ.

Если ты внимательно посмотришь на сертификат, то увидишь, что он похож на настоящий бумажный сертификат, но в цифровом формате. Он содержит информацию о том, кому выдан сертификат, открытый ключ этого сервера, местонахождение этого сервера и так далее. Справа ты видишь, как он выглядит сертификат в реальности.

На каждом сертификате указано имя, лицо или субъект, которому сертификат был выдан. Это очень важно, так как это поле поможет нам подтвердить их личность.

Если это для веб-сервера, то должно соответствовать тому, что пользователь вводит в URL-адресе в своем браузере. Если сайт известен под любыми другими именами, и если нужно, чтобы их пользователи получали доступ к своим приложениям также с другими именами, тогда все эти имена должны быть указаны в сертификате в разделе альтернативных имен субъектов.

Как видишь, любой может сгенерировать такой сертификат. Ты можешь создать его для себя, сказав, что ты Google, и именно это хакер сделал в данном случае. Он сгенерировал сертификат, в котором говорится, что он веб-сайт rotoro.cloud. Так как же нам проверить этот сертификат и убедиться, что он законный?

Именно здесь вступает в игру самая важная часть сертификата, а именно кто подписал и выдал сертификат.

Если ты сам его сгенерировал, то тебе придется подписать сертификат самостоятельно, что называется самозаверяющим сертификатом. Любой, кто посмотрит на созданный тобой сертификат, сразу поймет, что это небезопасный сертификат, потому ты его выписал сам себе. Т.о., если мы внимательно посмотрим на сертификат, который пользователь получил от хакера, то заметим, что это был поддельный сертификат, подписанный самим хакером.

Фактически, наш браузер делает это за нас. Все веб-браузеры имеют встроенный механизм проверки, при котором браузер верифицирует сертификат, полученный от сервера. Он проверяет, чтобы убедиться, что полученный сертификат законный и действительный в данный момент времени. А если он определяет, что это поддельный сертификат, то он сразу предупреждает нас.

Но как тогда создать законный сертификат для своих веб-серверов, такой, которому веб-браузеры будут доверять?
Как получить сертификаты, подписанные авторитетным лицом?

Вот где на помощь приходят `центры сертификации`, `certificate authorities` или просто `CA`. Это хорошо известные организации, которые могут подписывать и проверять наши сертификаты для нас. Некоторые из самых популярных - Comodo, DigiCert, Symantec, GeoTrust, Lets Encrypt и др.

Это работает следующим образом: мы генерируем запрос на подпись сертификата, certificate signing request или CSR, используя ранее созданный ключ и доменное имя вашего веб-сайта.

Для этого мы используем команду openssl. Она создает файл my-cert.csr, который представляет собой запрос на подпись сертификата, который должен быть отправлен в CA для подписи, он выглядит следующим образом. Центр сертификации проверяет наши данные и после проверки подписывает сертификат и отправляет его нам.

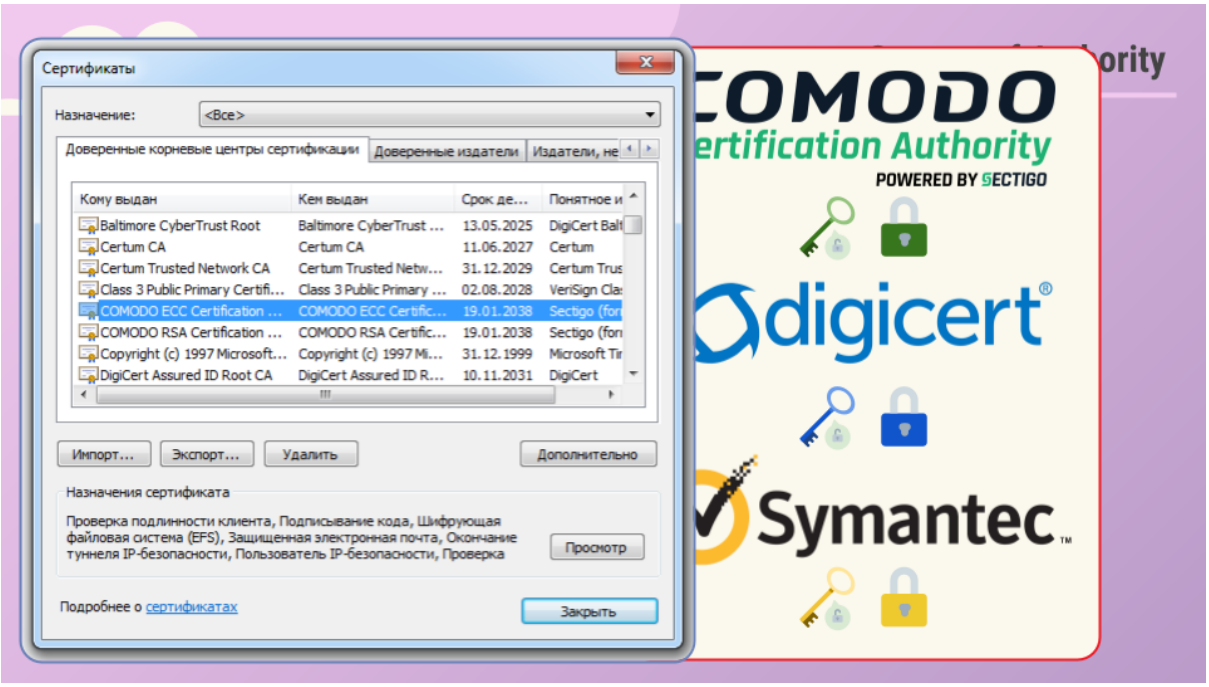
Теперь у нас есть сертификат, подписанный центром сертификации, которому доверяют браузеры. Если хакер попытается подписать свой сертификат таким же образом, он потерпит неудачу на этапе проверки, и его сертификат будет отклонен CA, поэтому его подложный веб-сайт, который он размещает, не будет иметь действительного сертификата.

Центры сертификации используют разные методы, чтобы убедиться, что ты являешься фактическим владельцем этого домена. Теперь у нас есть сертификат, подписанный центром сертификации, которому доверяют браузеры, но как браузеры узнают, что сам CA был законным?

Например, один из сертификатов был подписан поддельным CA. В данном случае наш сертификат был подписан Comodo.

Как браузер узнает, что Comodo является действующим центром сертификации и что сертификат на самом деле подписан

Comodo, а не кем-то, кто называет себя Comodo?



Сами центры сертификации имеют набор пар открытого и закрытого ключей. Certificate Authority используют свои закрытые ключи для подписи сертификатов. Открытые ключи всех центров сертификации встроены в браузеры.

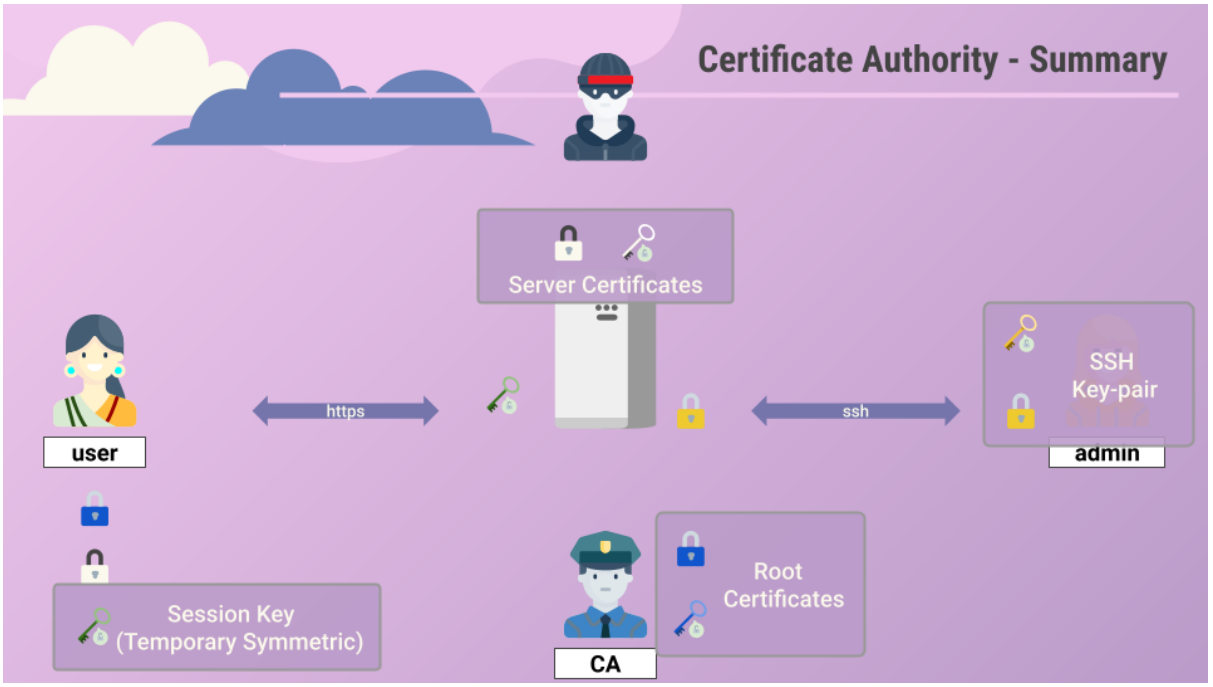
Браузер использует открытый ключ CA для проверки того, что сертификат действительно подписан самим CA.

Фактически ты можешь увидеть их в настройках своего веб-браузера в разделе сертификатов, они находятся на вкладке доверенных центров сертификации.

Это публичные центры сертификации, которые помогают нам гарантировать, что общедоступные веб-сайты, которые мы посещаем, такие как наши банки, электронные почтовые сервисы или поисковики являются теми, за кого себя выдают.

Однако они не помогают нам проверять сайты, размещенные в частном порядке, например, в твоей организации, которые мы используем, скажем, для доступа к своим платежным ведомостям или внутренним приложениям электронной почты. Для этого ты можешь разместить свои собственные частные центры сертификации.

Большинство из компаний, перечисленных здесь, имеют частное предложение своих услуг, сервер CA, который ты можешь развернуть внутри своей компании. Затем тебе потребуется установить открытый ключ своего внутреннего CA-сервера во всех браузерах сотрудников. После этого появится возможность установить безопасное соединение в организации.



Итак, давай кратко резюмируем о чем мы говорили.

Мы увидели, почему нам приходится шифровать сообщения, отправляемые по сети. Для шифрования сообщений мы используем асимметричное шифрование с парой открытого и закрытого ключей.

Администратор использует пару ключей для защиты SSH-соединения с серверами. Сервер использует пару ключей для защиты трафика HTTPS, но для этого сервер сначала отправляет запрос на подпись сертификата в CA, а CA использует свой закрытый ключ для подписи CSR.

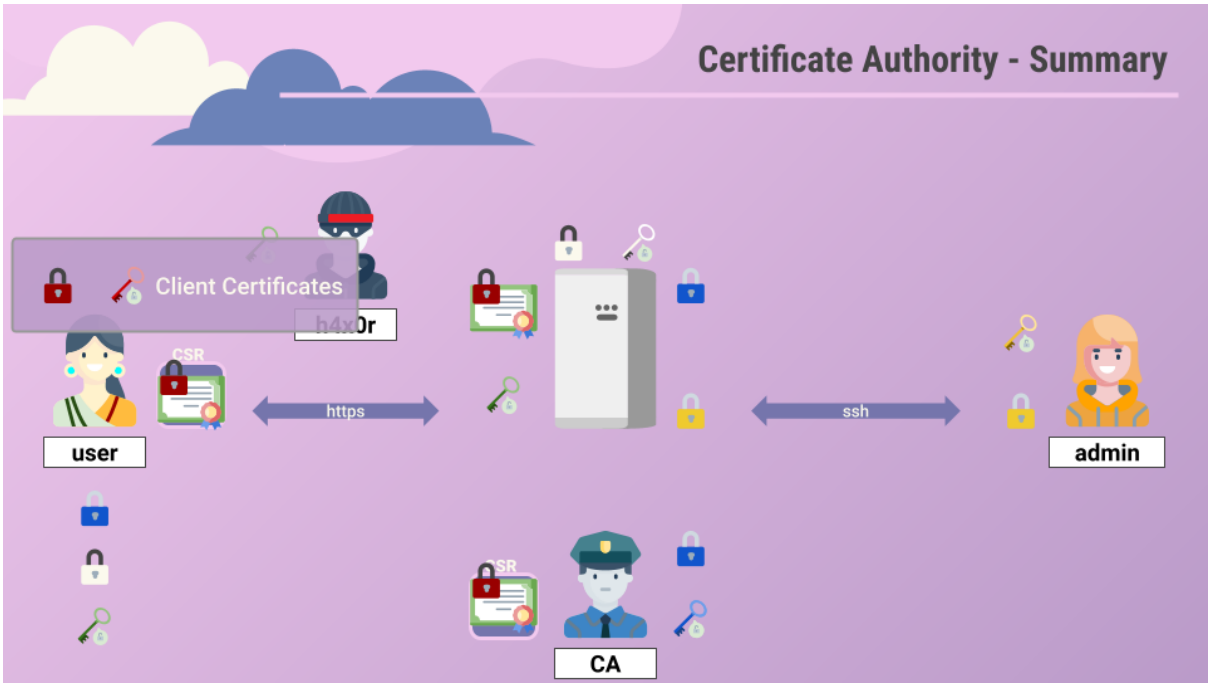
Не забудь, что у всех пользователей есть копия открытого ключа этого центра сертификации. Подписанный сертификат затем отправляется обратно на сервер, и там встраивается в работу веб-приложения.

Когда пользователь обращается к веб-приложению, сервер сначала отправляет сертификат со своим открытым ключом. Пользователь, ну т.е. его браузер, проверяет сертификат, используя открытый ключ CA для проверки и получения открытого ключа сервера. Затем он генерирует симметричный ключ, который желает использовать в дальнейшем для всех коммуникаций.

Симметричный ключ шифруется с использованием открытого ключа сервера и отправляется обратно на сервер. Сервер использует свой закрытый ключ для расшифровки сообщения и получения симметричного ключа для связи. Далее в течение сессии связь идет через этот более быстрый симметричный ключ.

Таким образом администратор генерирует пару ключей для защиты SSH, веб-сервер генерирует пару ключей для защиты веб-сайта с помощью HTTPS, а центр сертификации генерирует свой собственный набор пары ключей для подписи

сертификатов. Из всей этой компании конечный пользователь генерирует только один симметричный ключ, после того как он устанавливает доверительные отношения с веб-сайтом.



Он использует свое имя пользователя и пароль для аутентификации на веб-сервере. С помощью пар ключей сервера клиент смог установить, что сервер является тем, кем он себя называет, но сервер не знает наверняка, является ли клиент тем, за кого он себя выдает.

Это может быть и хакер, выдающий себя за пользователя, каким-то образом получив доступ к его учетным данным. Но это точно не через сеть, поскольку мы уже защитили ее с помощью TLS, видимо каким-то другим способом. В любом случае, что может сделать сервер, чтобы подтвердить, что клиент является тем, кем он себя называл?

Для этого в рамках начальной процедуры по установлению доверия сервер может запросить сертификат у клиента.

Предварительно клиент должен сгенерировать пару ключей и подписанный сертификат из действительного центра сертификации. Затем клиент отправляет сертификат на сервер для проверки того, что клиент является тем, за кого себя выдает.

Ты, должно быть, думаешь, что мы никогда не создавали клиентский сертификат для доступа к веб-сайту. Это потому, что сертификаты клиентов TLS обычно не реализуются на веб-серверах. В обычном веб-серфинге это так, но внутри мира онлайн-банкинга, например, где процедуры сопряжены с серьезным compliance это вполне рядовое явление.

В любом случае, все это реализовано внутри, поэтому обычному пользователю не нужно создавать сертификаты и управлять ими вручную.

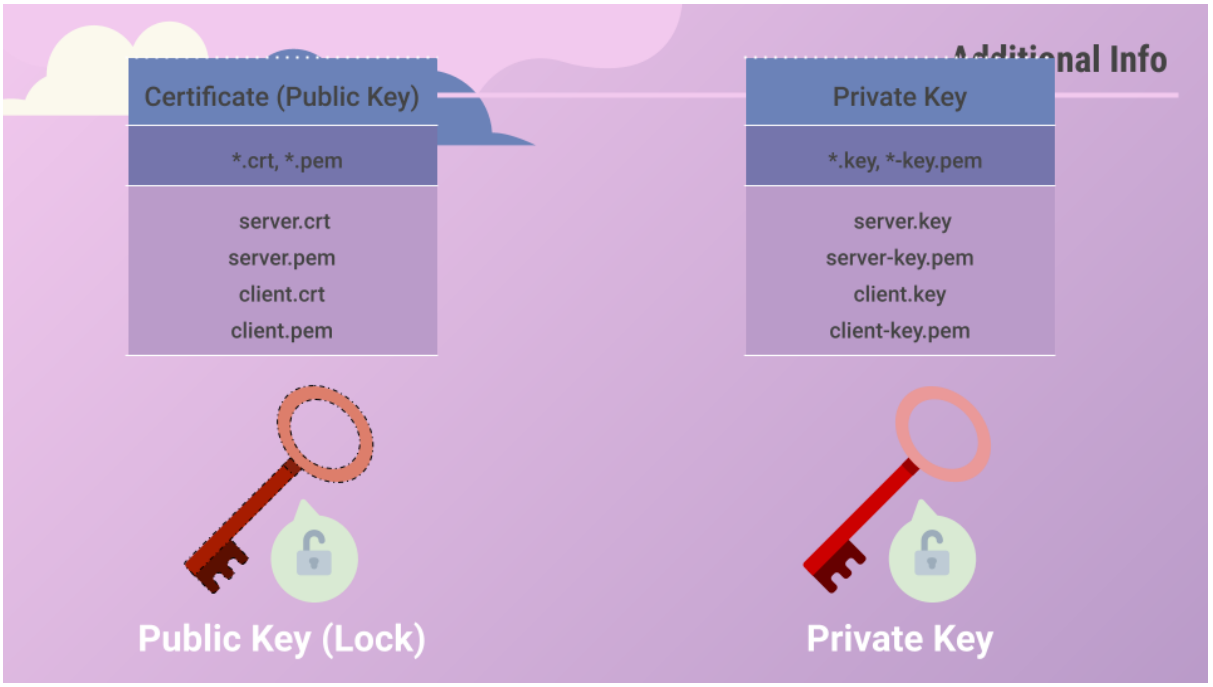
Ок, эта часть о клиентских сертификатах была последняя. Вся эта инфраструктура, включая центр сертификации, серверы, людей, а также процесс создания, распространения и обслуживания цифровых сертификатов, известна как инфраструктура открытых ключей, public key infrastructure или PKI.

Наконец, позволь мне кое-что прояснить, прежде чем мы закончим. Я здесь использовал аналогию ключа и замка для закрытых и открытых ключей.

Если у тебя создалось впечатление, что только открытый ключ, или, как мы его называли, публичный замок могут зашифровать данные, то, на самом деле, это неправда. Фактически это два связанных или парных ключа.

Ты можешь зашифровать данные с помощью любого из них, а расшифровать данные только с помощью противоположного.

Ты не можешь зашифровать данные одним ключом и расшифровать их с помощью него же, поэтому тебе нужно быть осторожным при шифровании своих данных. Если ты зашифруешь данные своим закрытым ключом, запомни, что любой, кто владеет твоим открытым ключом, а это действительно может быть кто угодно, сможет расшифровать и прочитать твое сообщение.



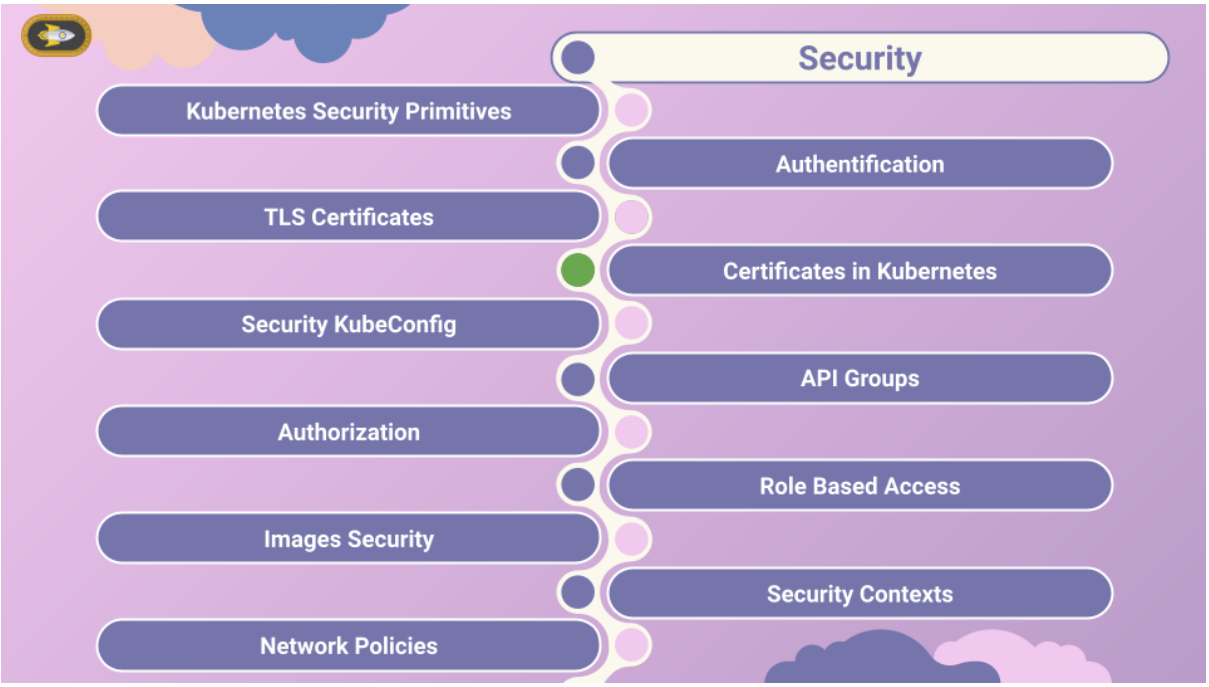
И напоследок, небольшое замечание по соглашениям об именах. Обычно сертификаты с открытым ключом называются расширением CRT или PEM. Т.е. server.crt, server.pem для наших сертификатов серверов. А client.crt или client.pem для клиентских сертификатов.

Закрытые ключи обычно имеют расширение .key или -key.pem. Например, server.key или server-key.pem.

Просто помни, что в закрытых ключах слово `ключ` обычно присутствует либо в качестве расширения, либо в имени сертификата.

Тот файл, в котором нет слова `key`, обычно является открытым ключом или сертификатом.

И это все в этой длинной лекции. Увидимся на следующей.



Привет и добро пожаловать на лекцию. В этой лекции мы рассмотрим защиту кластера Kubernetes с помощью сертификатов TLS.

В предыдущей лекции мы разбирали, что такое открытый и закрытый ключи, как сервер использует открытые и закрытые ключи для защиты подключения. Будем называть их сертификатами серверов или serving certificates.

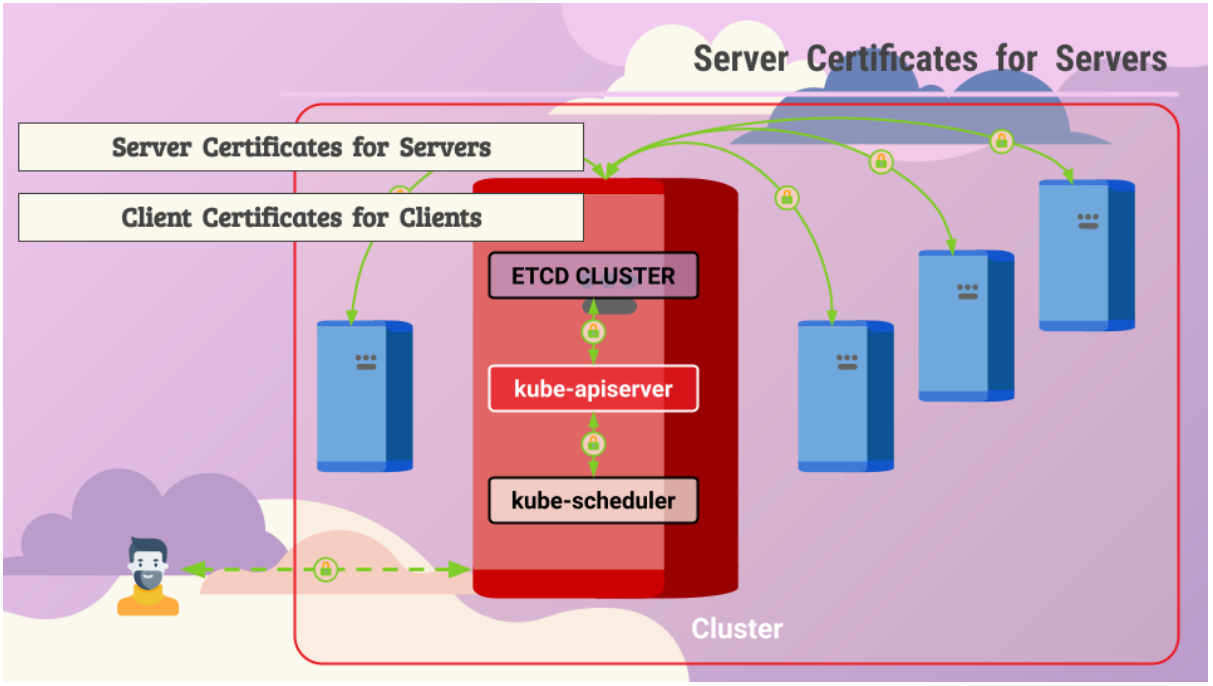
Мы видели, что такое центр сертификации. Мы узнали, что CA имеет свой собственный набор пар открытого и закрытого ключей, которые он использует для подписи сертификатов сервера. Мы будем называть их корневыми или root-сертификатами. Мы также увидели, как сервер может запросить у клиента подтверждение себя с помощью клиентских сертификатов.

Итак, прежде чем мы продолжим, освежи в голове две вещи:

- всего три типа сертификатов - несколько, настроенных на серверах, корневой сертификат, настроенный на серверах CA, а затем сертификаты клиентов, настроенные на клиентах
- нюансы по соглашению об именах

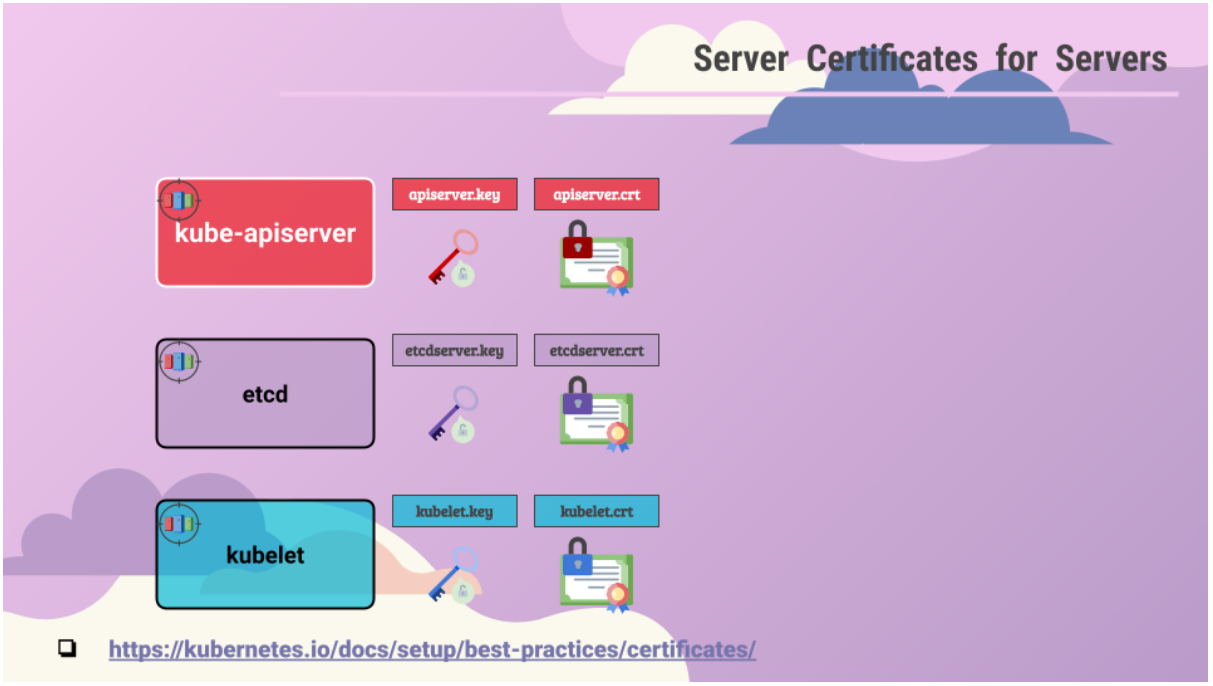
В этой лекции ты увидишь много файлов сертификатов, и это может сбить с толку. Поэтому используй этот метод, чтобы узнать, какой из них кто.

Обычно сертификаты с открытыми ключами называются расширением crt или pem. Это server.crt, server.pem для сертификатов сервера или client.crt или client.pem для клиентских сертификатов. А закрытые ключи обычно имеют расширение `.key` или `–key` в имени файла. Например `server.key` или `server-key.pem`. Так что просто помни, что в закрытых ключах есть слово `key` или в расширении, или в имени сертификата, а тот, в котором отсутствует это слово, обычно является открытым ключом или сертификатом. Вот так это запоминаю я.



Теперь мы увидим, как эти концепции относятся к кластеру Kubernetes. Кластер Kubernetes состоит из набора главных и рабочих узлов. Конечно, все коммуникации между этими узлами должны быть безопасными, и следовательно должны быть зашифрованы при передаче. Все взаимодействия между всеми службами и их клиентами тоже должны быть безопасными.

Например, администратор, взаимодействующий с kubernetes-кластером через утилиту kubectl или напрямую обращаясь к API Kubernetes, должен установить безопасное соединение TLS. Связь между всеми компонентами в кластере Kubernetes также должна быть защищена, поэтому два основных требования заключаются в том, чтобы все эти очень разные службы в кластере использовали сертификаты сервера, а все клиенты использовали сертификаты клиентов, чтобы убедиться, что они те, за кого себя выдают.



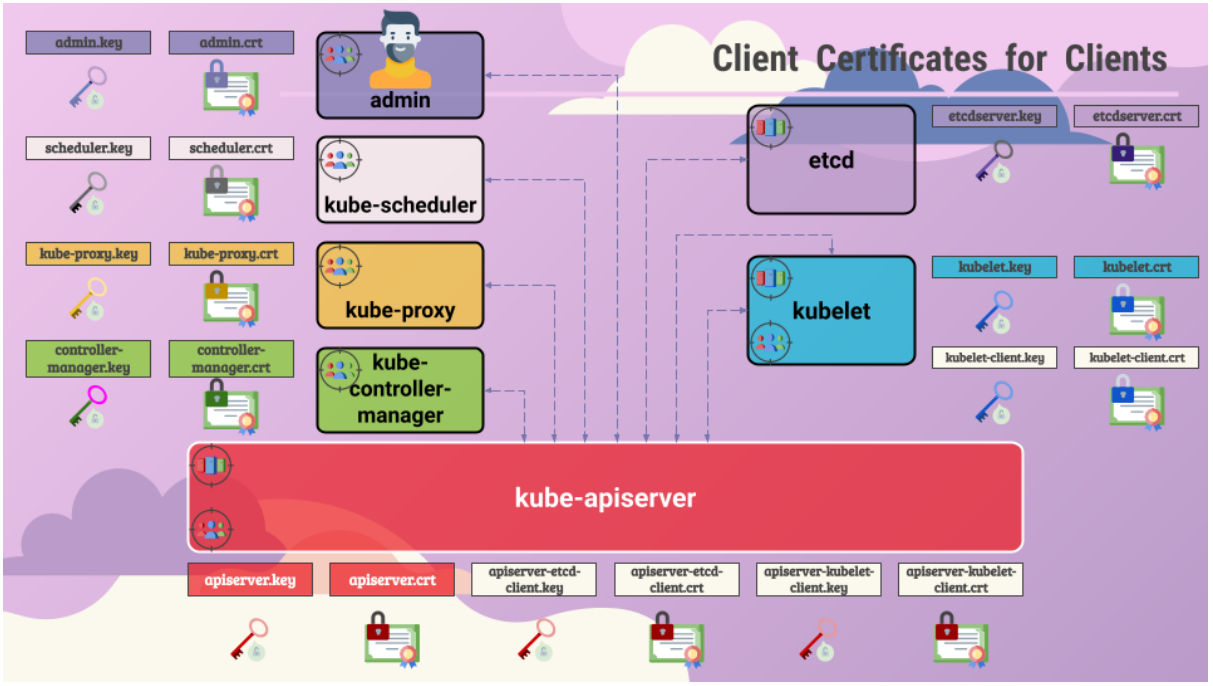
Давай посмотрим на различные компоненты в кластере Kubernetes и определим, кто для кого является сервером, а для кого клиентом, а также кто с кем разговаривает.

Начнем с kube-apiserver. Мы уже знаем, что API-сервер представляет собой HTTP-сервис, который другие компоненты, а также внешние пользователи используют для управления кластером Kubernetes.
Итак, это сервер, и ему требуются серверные сертификаты для защиты связи со своими клиентами. Это значит, что мы генерируем для сертификата пару ключей. Называем их APIserver.crt и APIsever.key. В дальнейшем мы постараемся придерживаться этого соглашения об именах.

Все, что имеет расширение .ctr, является сертификатом, а расширение .key - закрытым ключом. Также помни, что имена сертификатов могут отличаться в разных сетапах Kubernetes в зависимости от того, кто и как настраивал кластер. Значит у тебя эти имена могут быть другими. В этой лекции мы слегка отойдем от канонов и будем использовать имена, которые помогут нам легко идентифицировать файлы сертификатов. О лучших практиках названий для сертификатов читай в этой ссылке внизу.

Другой сервер в кластере - это сервер ETCD. Сервер ETCD хранит всю информацию о кластере, поэтому ему требуется пара сертификат-ключа. Мы назовем его ETCDserver.crt и ETCDserver.key.

Еще один компонент-сервер в кластере находится на рабочих узлах. Это служба kubelet, и у нее доступны API HTTP-endpoints, с которыми kube-apiserver общается для взаимодействия с рабочими нодами. Разумеется для этого потребуется сертификат и ключ. Мы назовем их kubelet.crt и kublet.key.



Это были серверные компоненты в кластере Kubetnetes. Теперь давай посмотрим на клиентские компоненты, которые выступают в роли клиентов при обращении к службам. Клиентам, вроде нас, которые являются администраторами и обращаются к kube-apiserver через kubectl с помощью REST API, требуется сертификат и пара ключей для аутентификации в kube-api. Мы назовем их admin.crt и admin.key.

Планировщик обращается к API-серверу для поиска PODs, требующих назначения, а затем отправляет серверу api информацию куда планировать PODs на подходящие для этого рабочие узлы.

Scheduler обращается к kube-apiserver. Он посылает запрос и, следовательно, это клиент, такой же как пользователь admin. Планировщику необходимо подтвердить свою личность с помощью клиентского TLS-сертификата, поэтому ему нужна собственная клиентская пара сертификата и ключа. Мы назовем его scheduler.crt и scheduler.key.

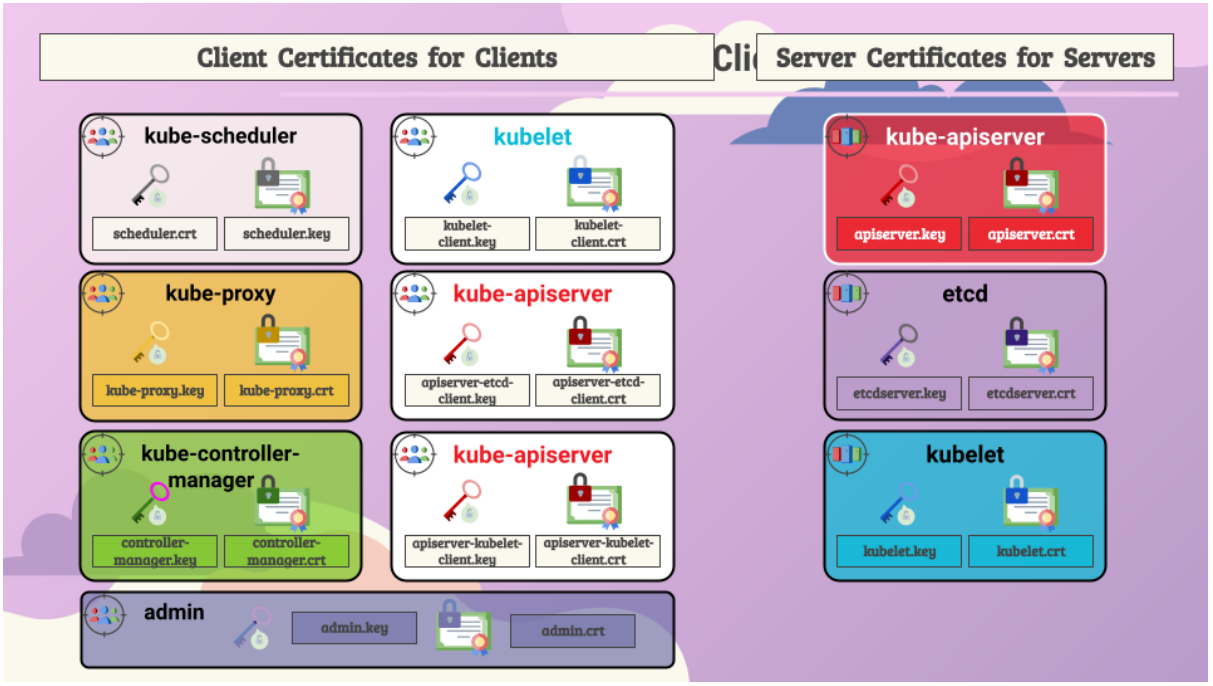
Компонент kube-proxy. Kube-proxy требует сертификата клиента для аутентификации на сервере kube-api. Для этого требуется собственная пара сертификата и ключа. Мы назовем их kube-proxy.crt и kube-proxy.key.

Последний клиентский компонент kube-controller-manager. Это еще один клиент, который обращается к API-серверу. Таким образом, ему также требуется сертификат для аутентификации на сервере kube-api. Итак, мы создаем для него пару сертификатов.

Серверы также обмениваются данными между собой. Например, kube-apieserver взаимодействует с сервером ETCD. Фактически из всех компонентов kube-api является единственным, который общается с сервером ETCD. Итак, что касается сервера ETCD, уже kube-apiserver является его клиентом. Значит необходима аутентификация.

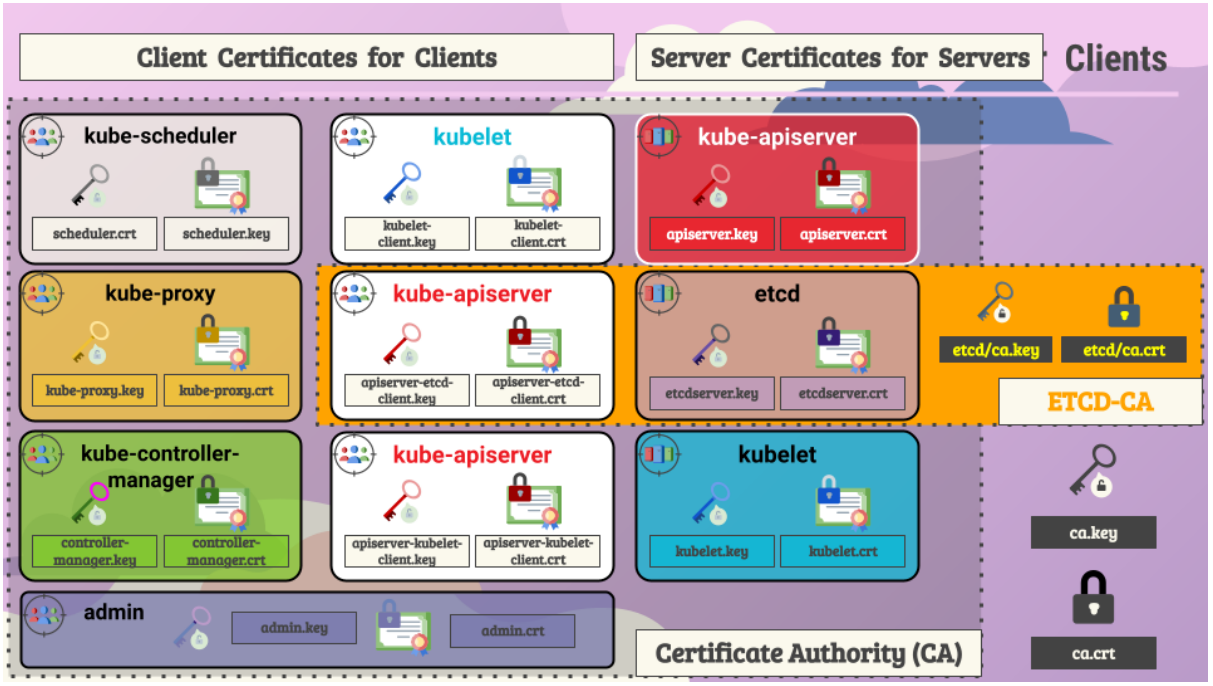
Сервер kubeapi может использовать те же ключи, которые он использовал ранее для обслуживания собственной службы API. Это файлы apiserver.crt и apiserver.key. Или мы можем создать новую пару сертификатов-ключей специально для kube-apiserver для аутентификации его на сервере etcd как клиента.

Сервер kube-api также общается с сервером kubelet на каждом из отдельных узлов. Так он следит за ситуацией на рабочих нодах. Опять же, он может использовать оригинальные сертификаты или мы можем сгенерировать новые специально для этой цели.



Как видишь, даже в таком простом случае сертификатов слишком много. Попробуем сгруппировать их. Существует набор serving сертификатов, которые в основном используются для подключения к kube-apiserver. И есть набор клиентских сертификатов, используемых сервером kube-api, etcd-server и kubelet для аутентификации своих клиентов.

Теперь посмотрим, как сгенерировать эти сертификаты. Как мы уже знаем, нам нужен центр сертификации, чтобы подписать все эти сертификаты. Kubernetes требует, чтобы у нас был хотя бы один центр сертификации для кластера. Фактически у тебя может быть более одного для всех компонентов кластера и еще один выделенный специально для ETCD.



В таком случае сертификаты серверов ETCD и сертификаты клиентов серверов ETCD, будут подписаны CA сервера ETCD. Пока мы будем придерживаться только одного центра сертификации для нашего кластера, поэтому сертификат клиента api-server мы заверим нашим единым CA-сертификатом.

У CA, как мы знаем, есть своя пара сертификата и ключа. Назовем его ca.crt и ca.key. Данная картинка это самый минимум сертификатов, используемых в кластере. Обычно в рабочем кластере их несколько больше.

Что же, это был обзор сертификатов в этой лекции, в следующей посмотрим как их создать.

Привет и добро пожаловать. В этой лекции мы рассмотрим, как сгенерировать сертификаты для кластера.

Для создания сертификатов доступны различные инструменты, такие как популярные EasyRSA, OpenSSL, SFSSL, cert-manager. Также есть многие другие.

В этой лекции мы будем использовать в качестве инструмента для создания сертификатов OpenSSL.

Вернемся и взглянем на схему с прошлой лекции. Вот наш план - создать все требуемые сертификаты для всех компонентов.
Начнем с самого нижнего уровня, а именно с создания root-сертификатов.

Сначала мы создадим закрытые ключи.
Затем мы используем только что созданный ключ для генерации запроса на подпись сертификата.
Потом подпишем сертификаты.

Создаем ключ командой

openssl genrsa -out ca.key 2048

Вот так он выглядит, это вполне длинный ключ, он обеспечит нашему кластеру хорошую стойкость шифрования, но для продуктовых кластеров выбирай не меньше 3072.

Создадим CSR командой openssl req -new -key ca.key -subj "/CN=KUBERNETES-CA" -out ca.csr

Запрос на подпись - это тот же сертификат, со всем данными, но пока без сигнатуры подписанта.

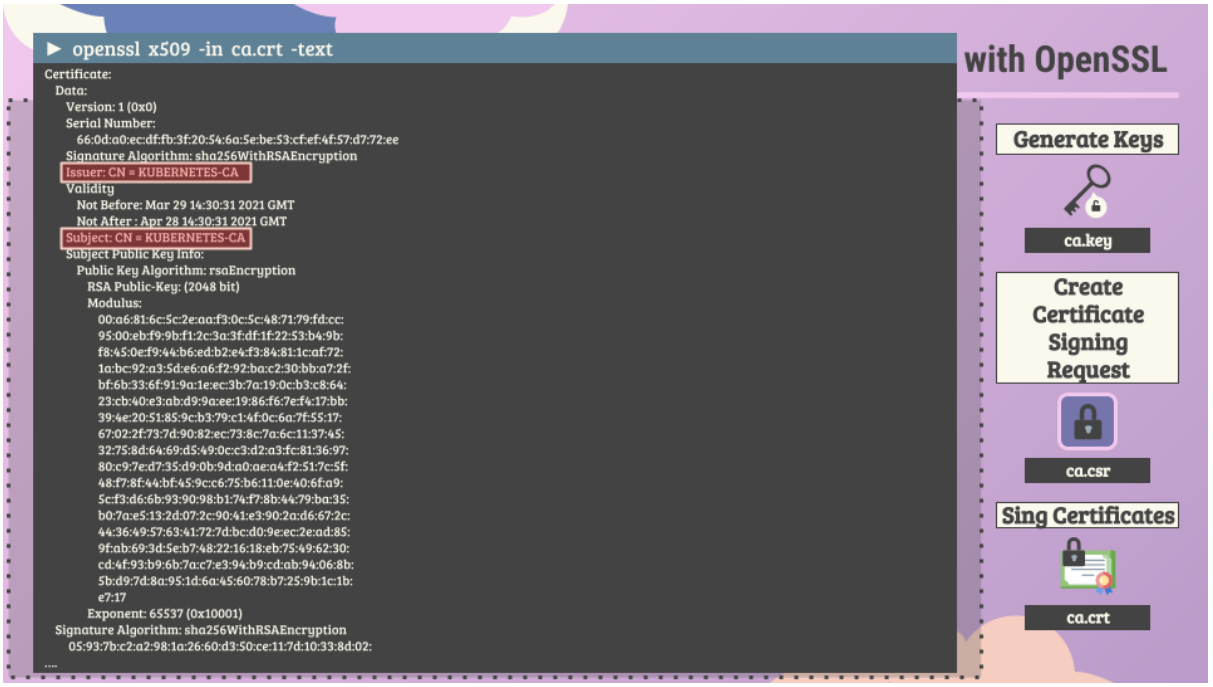
Мы указали здесь в качестве `common name`=KUBERNETES-CA. Если ты смотрел в документацию в ссылках прошлой лекции, то знаешь, основные имена центров сертификации в кластере обычно KUBERNETES-CA и ETCD-CA. Но в этом сетапе у нас будет всего один CA. Вот так выглядит наш запрос на подпись сертификата.

Прежде чем подписать, нужно сделать маленькую манипуляцию. Каждый подписанный сертификат имеет свой порядковый номер. Он хранится в файле ca.srl. Проинициализируем его значением 00.

Теперь взглянем на директорию, у нас есть закрытый ключ, запрос и счетчик для наших сертификатов.

Этот наш корневой сертификат будет иметь статус самой последней правды в кластере, т.к. мы сделаем так, чтобы ему безоговорочно доверяли все в кластере.

Создаем сертификат командой openssl x509 -req -in ca.csr -singkey ca.key -out ca.crt



Теперь у нас готов корневой сертификат. Разложив его по нужным папкам на узлах кластера мы заставим Kubernetes ему доверять. Сделаем это чуть позже, а сейчас давай просмотрим его командой

openssl x509 -in ca.crt -text

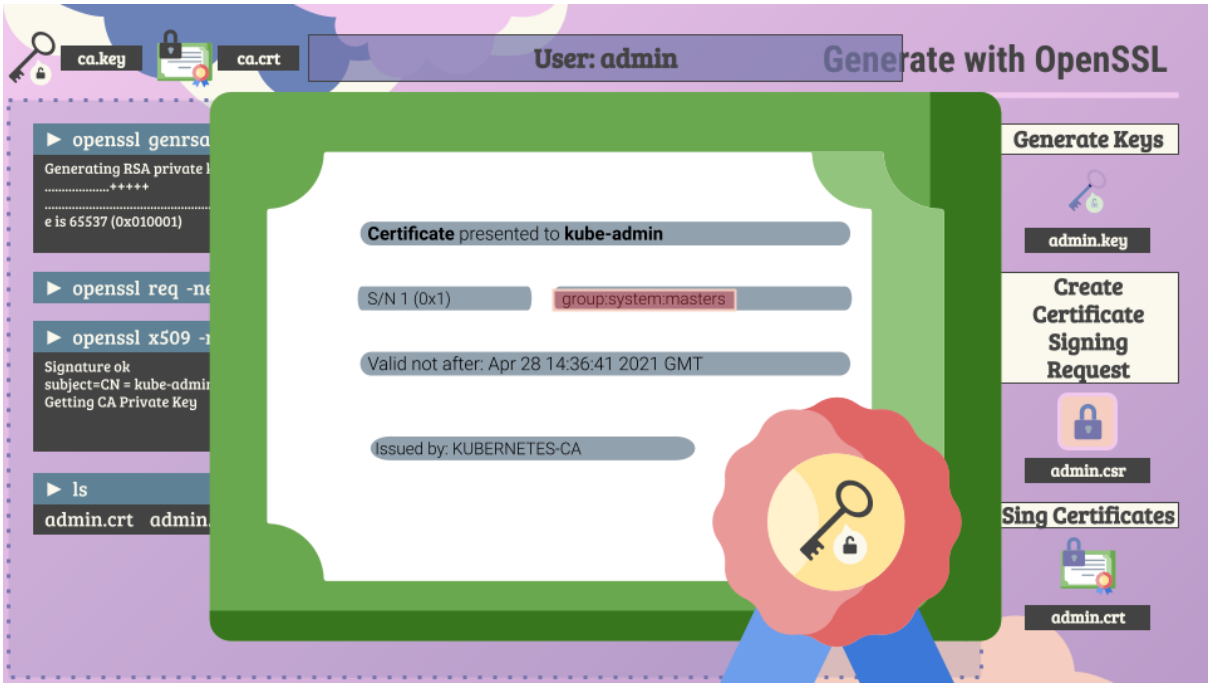
Как видишь поля Issuer и Subject совпадают, ведь мы сами себе подписали этот сертификат. Отныне все остальные сертификаты в кластере будут подписаны вот этим.

Здесь важное поле `valid not after`, которое говорит нам, что после 28 апреля 21 года сертификату перестанут доверять в кластере и его придется перевыпустить. А раз это корневой сертификат, придется переподписывать и все остальные.

Ок, теперь у CA есть закрытый ключ и файл корневого сертификата.
Двигаемся вперед. Давай теперь попробуем создать клиентские сертификаты.

Начнем с администратора. Мы следуем тому же процессу, где мы создаем закрытый ключ для пользователя-администратора с помощью команды openSSL.

Затем мы генерируем CSR и здесь указываем имя администратора, у нас это будет kube-admin.



Тут пара моментов.
На самом деле это не обязательно должен быть kube-admin или просто admin. Это может быть что угодно, но помни, что это то имя, под которым аутентифицируется клиент kubectl в кластере. И когда ты запустишь какую-нибудь команду через kubectl в логах ты увидишь это имя. Поэтому укажи его соответствующе, чтобы потом не путаться.

Второй момент в том, что сертификат принадлежит не рядовому пользователю.

Это сертификат, который пользователь-администратор будет использовать для аутентификации в кластере Kubernetes.

Если ты присмотришься, весь процесс генерации ключа и сертификата похож на создание учетной записи для нового пользователя. Здесь сертификат как бы является подтвержденным идентификатором пользователя, а ключ подобен паролю. Так и есть, и это намного безопаснее, чем простое имя пользователя и пароль.

Итак, kube-admin - это имя для пользователя с правами администратора. Но как отличать этого пользователя от других пользователей? Ведь эта учетная запись должна быть идентифицирована как пользователь с правами администратора, а не просто как какой-то другой базовый пользователь. Мы сделаем это, добавляя данные группы для пользователя в сертификат.

В этом случае это группа под названием `system:masters`. О группах мы поговорим позже. Но пока важно отметить, что нам необходимо указать эту информацию при создании запроса на подписание - certificate signing request.

Еще раз: чтобы заверить сертификат, как имеющий админские полномочия, мы должны указать специальную группу при создании запроса на подпись. После того, как эта группа будет заверена в сертификате, его обладатель будет ассоциирован с админской группой с админскими привилегиями. Это достигается добавлением записи `O=system:masters` в сертификат.

Вот теперь мы готовы создать запрос на подписание сертификата.

Мы, генерируем подписанный сертификат, взяв файл запроса на подпись и используя команду, как в первый раз. Но теперь есть отличие - мы используем корневой сертификат нашего центра сертификации в качестве подписанта сертификата админа. Это достигается участием опций -CA и -CAkey в этой команде. Теперь CA сделал этот сертификат действительным в нашем кластере.
Подписанный сертификат будет записан в файл admin.crt.

Сверим часы. У нас в каталоге 7 файлов: 4 относятся к CA и 3 к kube-admin.

Вот сертификат админа, в нем есть указание группы.

Далее повторим тот же процесс создания клиентских сертификатов для всех других компонентов, которые получают доступ в кластер.

Сначала Scheduler. Планировщик - системный компонент, это элемент controlpane. В связи с этим ему понадобится префикс `system:` в common name. Т.е. полный параметр `CN` будет `system:kube-scheduler` и это же укажем в параметре `O` при создании CSR.

Тоже самое для kube-controller-manager. Это тоже системный компонент и его имя начинается с `system:`.

Наконец kube-proxy. Тут небольшое отличие, в параметр `O` впишем `system:node-proxier`.

Итак, что у нас есть?

Мы создали корневые сертификаты. Затем все клиентские сертификаты, включая сертификаты администратора, планировщика, контроллер-менеджера и прокси.

Далее будем следовать той же процедуре, чтобы создать оставшиеся три клиентских сертификата: два для API-сервера и один для kubelet. Давай вернемся к ним, когда будем создавать для них сервисные сертификаты, поэтому пока отложим это в сторону.

А что мы можем сделать с этими сертификатами, которые уже созданы? Кое-что уже можем. Возьмем, к примеру, сертификат администратора для управления кластером.

Используя данные из файлов вместо имени пользователя и пароля в вызовах через kubectl или curl, указав закрытый ключ с сертификатом, а также корневой сертификат в качестве параметров вызова мы можем совершать вызовы в кластер.

Это один простой способ.

Другой способ - переместить все эти параметры в файл конфигурации под названием kube-config.yaml

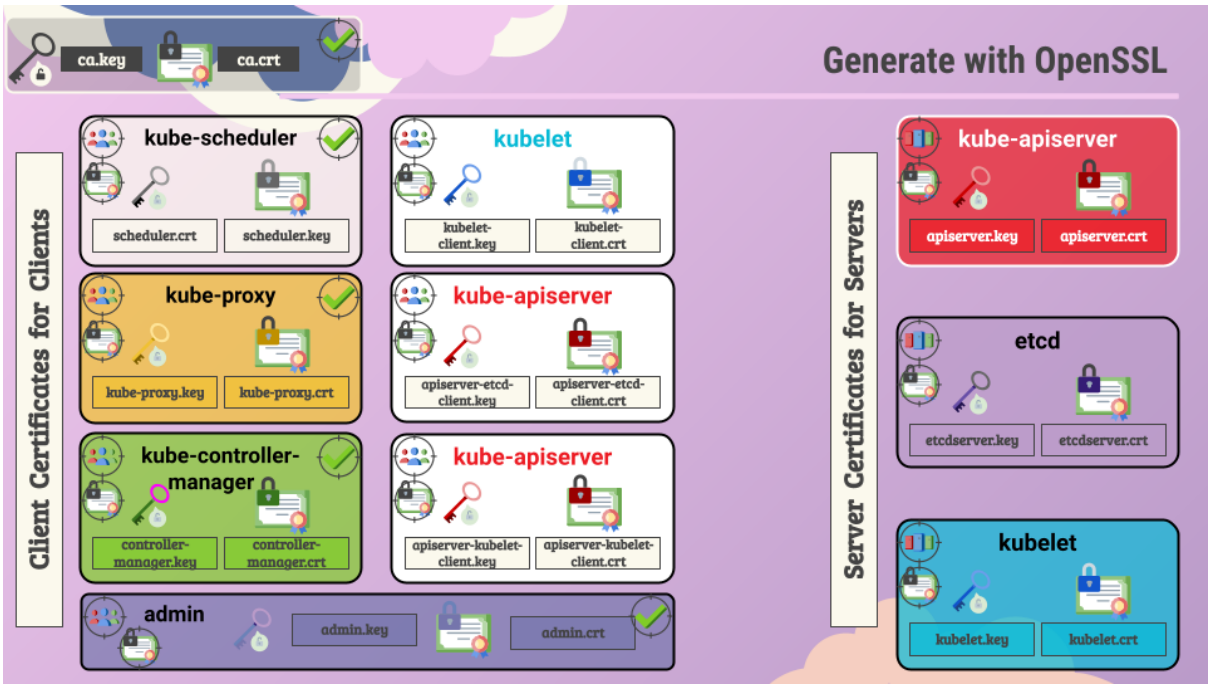
В нем укажем эндпоинт API-сервера, прочие детали, такие как пути к ключам и сертификатам и т. д.

Именно так у большинства работающих с Kubernetes людей все и настроено. Мы подробно рассмотрим kube-config в одной из следующих лекций.

Хорошо, теперь у нас остались серверные сертификаты.

Но прежде чем продолжить, еще небольшая ремарка. В лекции о предварительных условиях мы упоминали, что для того, чтобы клиенты могли проверять сертификаты, отправленные сервером, и наоборот, им всем нужна копия публичного сертификата центра сертификации.

Точно так же и в Kubernetes. Чтобы эти различные компоненты могли проверять друг друга, им всем нужна копия корневого сертификата.
Но в отличии от мира WWW-приложений, в котором root-сертификаты уже установлены в браузерах пользователей, тебе придется всякий раз при настройке сервера или клиента иметь на нем копию этого нулевого сертификата.



Посмотрим на сертификаты на стороне сервера.

Начнем с сервера ETCD. Мы следуем той же процедуре, что и раньше, чтобы сгенерировать для него сертификат, и назовем его etcdserver.crt.

Службу ETCD можно развернуть как кластер на нескольких серверах в среде высокой доступности. В этом случае, чтобы защитить связь между различными участниками ETCD-кластера, мы должны сгенерировать дополнительные сертификаты одноранговых узлов. После создания этих peer-сертификатов укажи их при запуске активного сервера. В файле определения есть параметры для указания файлов ключей и сертификатов самого сервера и его пиров. В этом примере, как ты видишь ETCD использует свой собственный файл корневого сертификата.
Существуют и другие варианты указания сертификатов, это зависит от варианта установки ETCD.

Теперь поговорим о API-сервере.

Мы генерируем сертификат для сервера API, как и раньше, но сервер API является самым популярным из всех компонентов в кластере. Все разговаривают с этим сервером. Каждая операция проходит через API-сервер.

Если что-то движется внутри кластера, сервер API знает об этом.

Если тебе нужна информация, ты берешь kubectl и говоришь с сервером API.

И поэтому он имеет множество имен и псевдонимов в кластере.

Его настоящее имя - kube-apiserver, но некоторые называют его kubernetes, потому что для многих людей, которые понятия не имеют, что происходит под капотом Kubernetes, kube-apiserver и есть Kubernetes.

Для некоторых служб он kubernetes.default, или kubernetes.default.svc, или kubernetes.default.svc.cluster. Однако его полное имя - kubernetes.default.svc.cluster.local

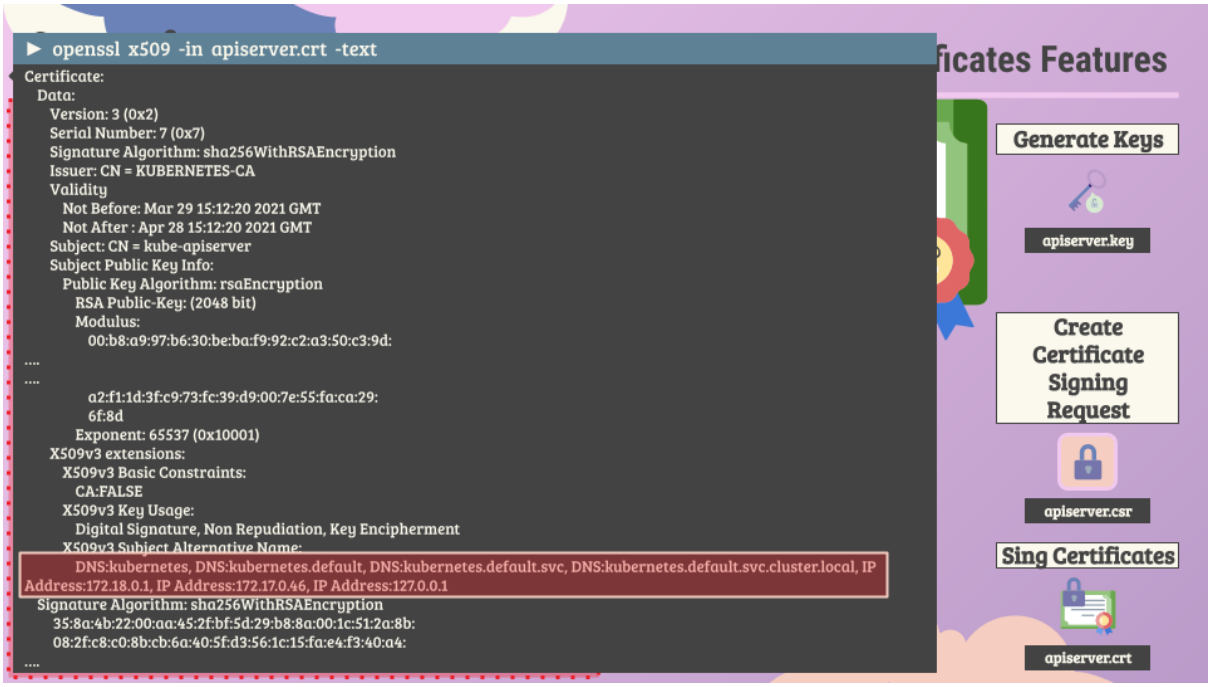
Наконец, в некоторых местах он также упоминается просто по его внешнему IP-адресу, или IP-адресу хоста, на котором запущен сервер kube-apiserver, или POD, в котором он выполняется. А еще это может быть даже просто loopback адрес.

Таким образом, все эти имена должны присутствовать в сертификате, созданном для сервера. Только тогда те, кто обращается к серверу под этими именами, смогут установить действительное соединение.

Итак, мы использовали тот же набор команд, что и раньше для генерации ключа в сертификатах.

Создавая запрос, мы указываем CN = kube-apiserver.

Но как указать все альтернативные имена для этого? Нам требуется создать файл конфигурации для openSSL. Это файл с расширением cnf, он разбит в ini-формате. Там укажем эти альтернативные имена в соответствующей группе. Имя в формате адреса укажем в виде IP.цифры, а имена домена с префиксом DNS.



Теперь при генерации запроса на подпись apiserver.csr мы используем параметр `-config` и укажем наш конфиг-файл.

Просмотрев директорию я насчитал у себя 21 созданный файл, относящийся к генерации сертификатов кластера.

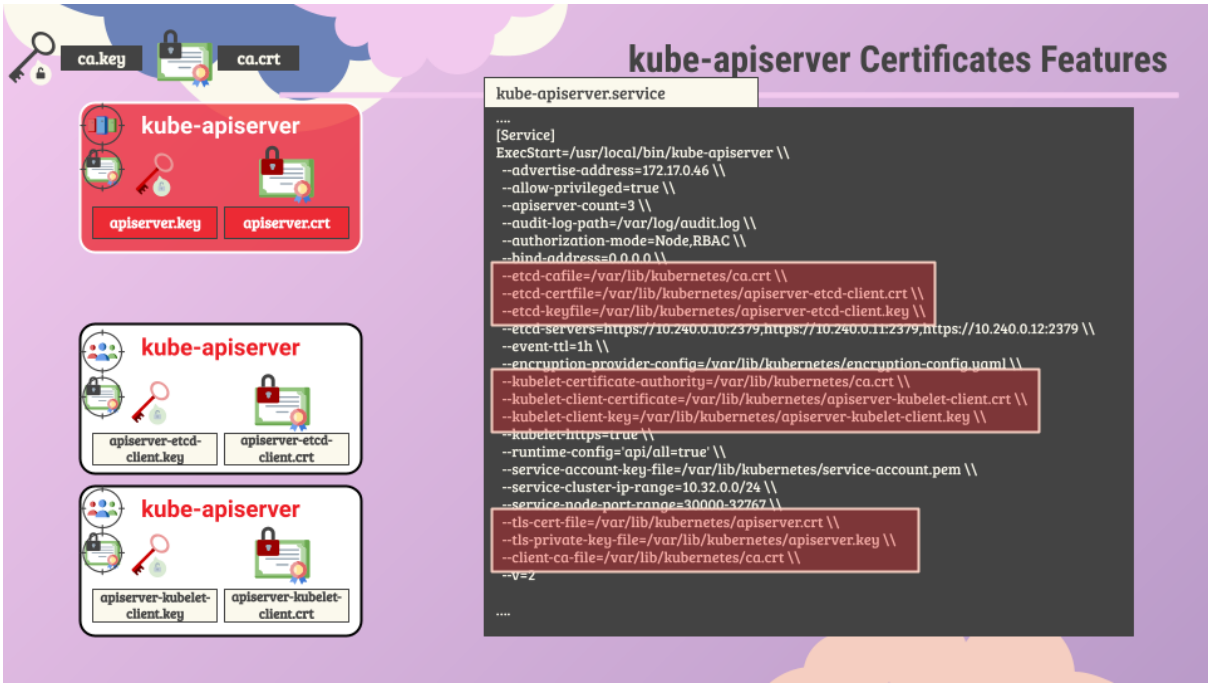
Наконец, подпишем сертификат, как обычно используя корневой сертификат и ключ -CA. Здесь я использовал расширение v3_req, поэтому включим его соответствующей опцией -extensions.

Ну вот, у нас есть сертификат API-сервера, проверим все ли верно указалось с помощью команды просмотра сертификата. Как видишь у нас появились нужные нам subject alternative names.

Давай посмотрим, где мы собираемся указать эти ключи и сертификаты.
Примем здесь во внимание клиентские сертификаты apiserver-etcd-client и apiserver-kubelet-client, которые тоже используются сервером API.
Как помнишь, kube-api общается в качестве клиента с серверами ETCD и kubelet.

Мы укажем расположение этих сертификатов, чтобы параметры были переданы в исполняемый файл kube-apiserver. Это правится в файле-конфигурации службы или статическом POD kube-api.
Если у тебя несколько мастеров, нужно это сделать для каждого.

Итак, первая группа, это сертификат, который apiserver будет предоставлять своим клиентам, далее ключ, при помощи которого он будет расшифровывать ответ. Далее - корневой сертификат, чтобы kubeapi мог проверить сертификаты своих клиентов.



Вторая группа относится к работе с ETCD. Для проверки того, что ETCD-сервер настоящий у нас указан корневой сертификат (в этом сетапе совпадает с root-сертификатом kubernetes-ca). Далее идут сертификат и закрытый ключ, для того, чтобы ETCD сервер мог проверить подлинность самого kubeapi-server.

Третья группа - kubelet. Здесь все похоже на предыдущую группу.

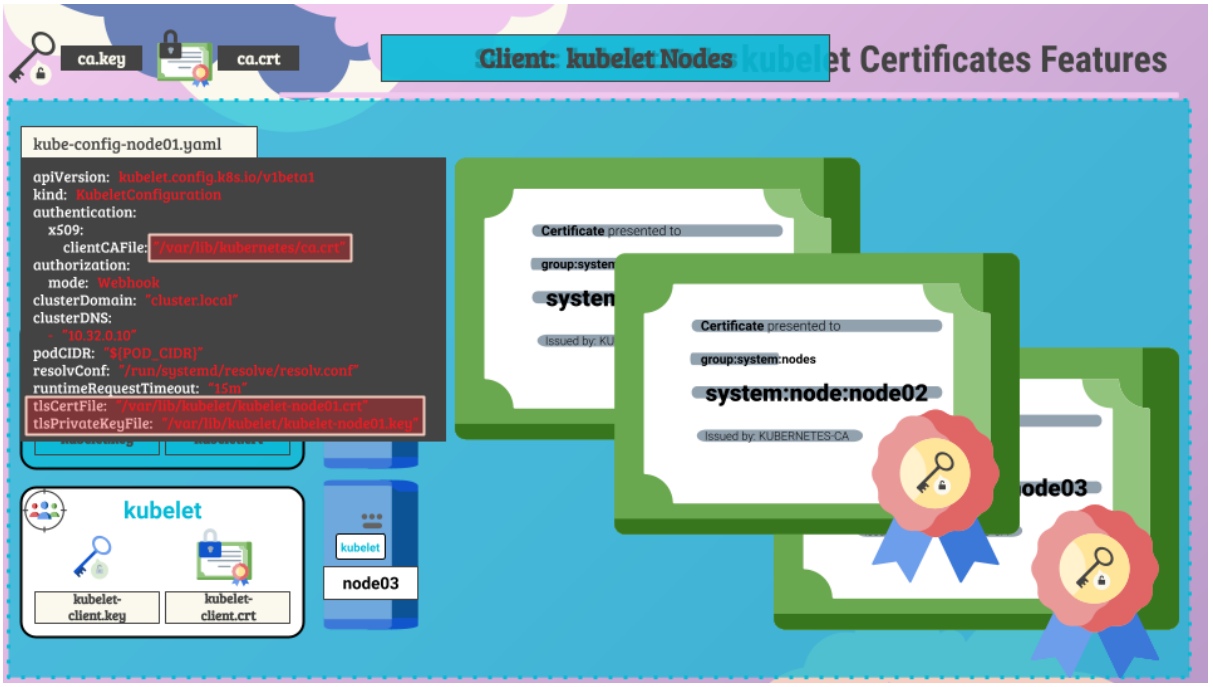
На этом с kube-api все.

Переходим к серверному сертификату kubelet, у него тоже свои особенности.

Kubelet-сервер - это HTTPS-API сервер, часть службы kubelet, которая работает на каждом узле и отвечает за управление этой нодой.

С ним общается kube-api, чтобы мониторить узел, получая информацию, которая далее используется для планирования PODs на эту ноду. Таким образом, нам нужны пары сертификат-ключ для каждого узла в кластере.

Как нам назвать эти сертификаты? Назовем их все KUBELET? Нет, они будут названы в соответствии с именами своих нод. Node01, node02 и node03.



После создания сертификатов используем их в файле конфигурации kubelet.

Как всегда, мы указываем корневой сертификат, чтобы kubelet знал, кто к нему обращается. Затем пропишем сертификаты и ключи соответствующие данной ноде. Мы должны сделать это для каждого узла в кластере.

Мы также говорили о наборе клиентских сертификатов, которые будет использовать kubelet для его обращения к API-серверу.

Их тоже нужно сгенерировать. А как мы назовем эти сертификаты?

Серверу kube-api необходимо знать, какой узел аутентифицирован, и предоставить ему правильный набор разрешений.
Таким образом, это требует, чтобы узлы имели правильные имена в правильных форматах.

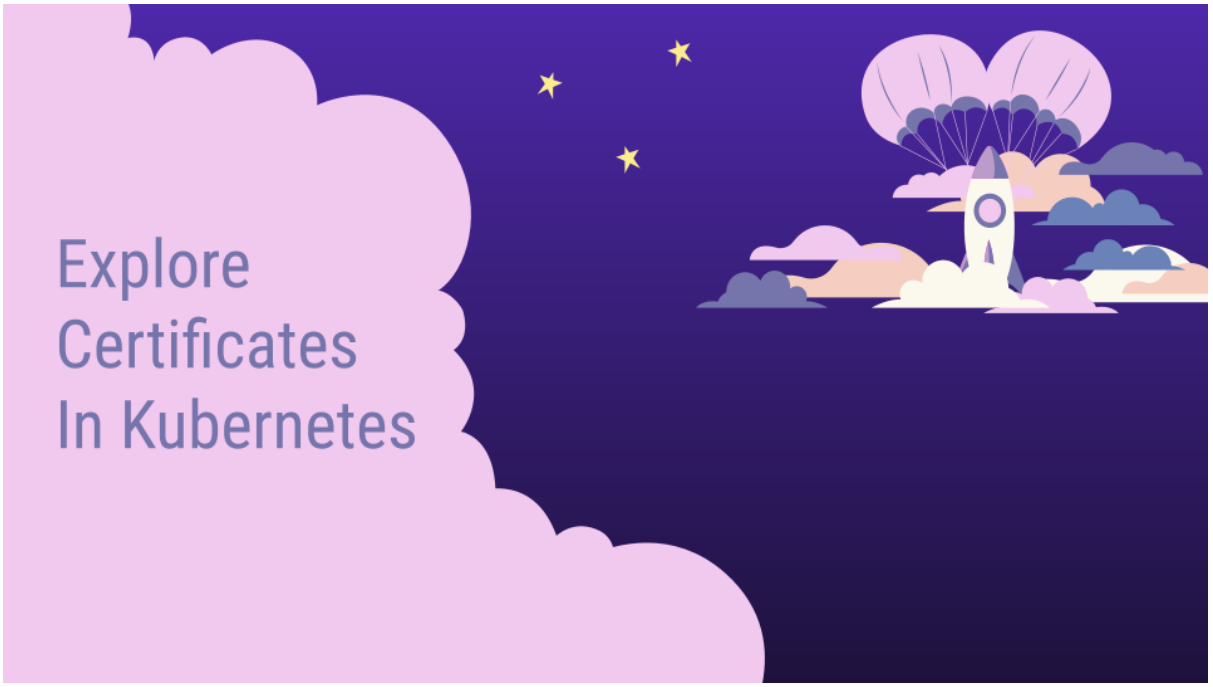
Поскольку узлы являются системными компонентами, такими же как планировщик и менеджер контроллеров, о которых мы говорили ранее, формат начинается с ключевого слова `system`, потом `:node:`, а затем имя узла, как ты видишь на экране у нас от 01 до 03.

А как API-сервер даст ему правильный набор разрешений?
Помнишь, что мы указали имя группы для пользователя с правами администратора, чтобы повысить привилегии пользователя kube-admin в кластере?
Точно так же требуется добавить узлы в группу с именем `system:nodes`.

После создания сертификатов, мы добавляем ссылки на них в kube-config-node файл, как мы обсуждали ранее.

Вот и все в этой лекции. Имей в виду, что это один из возможных сетапов, и тебе решать, для каких элементов и какого рода сертификаты нужны в твоем кластере.

Мы посмотрим на отличия в сертификатах следующей лекции в которой познакомимся, как просматривать информацию о сертификатах и конфигурировать их с помощью kubeadm.



Привет и добро пожаловать на лекцию. В этой лекции мы увидим, как просматривать сертификаты в существующем кластере.

Итак, ты присоединился к новой команде, и твои обязанности в ней - управлять кластером Kubernetes. Тебе сообщают, что есть несколько проблем, связанных с сертификатами в твоей новой среде, поэтому тебе предстоит выполнить проверку работоспособности всех сертификатов во всем кластере.

Что тебе делать? Прежде всего важно знать, как был настроен кластер. Существуют разные решения для развертывания кластера Kubernetes, и они используют разные методы для создания сертификатов и управления ими.

Если тем, кто создавал кластер, по каким-то причинам нужно было развернуть кластер Kubernetes с нуля, они создавали все сертификаты самостоятельно, как мы это делали в предыдущей лекции.

Ну а если они полагались на инструмент автоматического развертывания, такой как kubeadm, то он позаботился о создании и настройке кластера и его сертификатов.

В первом случае ты найдешь компоненты Kubernetes как нативные службы на узлах кластера, во втором случае - как PODs, которые развернул инструмент kubeadm.

Поэтому важно знать, где искать нужную информацию.

В этой лекции в качестве примера рассмотрим кластер, подготовленный kubeadm.

Для проверки работоспособности начнем с определения всех сертификатов, используемых в системе.

Здесь для этой цели я создал такую вот таблицу. Она будет доступна по ссылке на гугл-документ. Идея состоит в том, чтобы создать список файлов сертификатов с разбивкой по компонентам, ролям, типам файла и т.д. Чтобы взглянув на таблицу, можно было быстро разобраться, к чему какой сертификат относится, где лежит, какие поля заверяет и кем выпущен. И важно когда он станет недействительным.

Итак, откуда нам брать информацию для этой таблицы?

Начни с используемых файлов сертификатов. Для этого в кластере, настроенном с помощью kubeadm, найди файл определения для kube-apiserver. Он лежит в папке статичных PODs /etc/kubernetes/manifests.

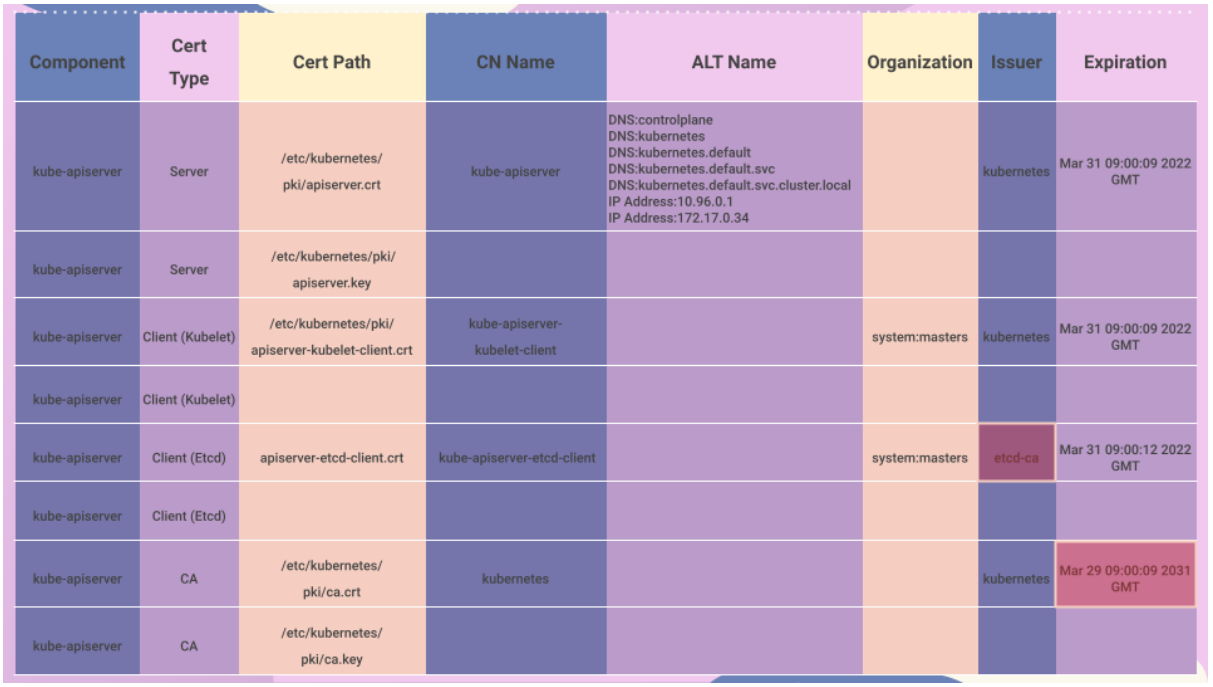
Команда, используемая для запуска api-сервера, содержит информацию обо всех сертификатах. Определим файлы сертификатов, используемый для каждой из целей, и запишем их в таблицу.

Далее возьмем каждый сертификат и заглянем внутрь, для получения более подробной информации о нем.
Мы начнем с файла сертификата apiserver.crt. Запустим команду openssl x509 и укажем файл сертификата в качестве входных данных для декодирования сертификата и просмотра сведений.

Первое - common name сертификата в разделе subject.

В данном случае это kube-apiserver. Теперь альтернативные имена. У сервера kube-api их много, поэтому нужно убедиться, что все они здесь есть. После проверим эмитента сертификата и раздел действительности, чтобы определить дату истечения срока действия сертификата.

CA, выдавший сертификат у нас kubernetes, это отличается от того, как мы делали на прошлой лекции руками. Это потому, что kubeadm называет свой центр сертификации как kubernetes.



Component	Cert Type	Cert Path	CN Name	ALT Name	Organization	Issuer	Expiration
kube-apiserver	Server	/etc/kubernetes/pki/apiserver.crt	kube-apiserver	DNS:controlplane DNS:kubernetes DNS:kubernetes.default DNS:kubernetes.default.svc DNS:kubernetes.default.svc.cluster.local IP Address:10.96.0.1 IP Address:172.17.0.34		kubernetes	Mar 31 09:00:09 2022 GMT
kube-apiserver	Server	/etc/kubernetes/pki/apiserver.key					
kube-apiserver	Client (Kubelet)	/etc/kubernetes/pki/apiserver-kubelet-client.crt	kube-apiserver-kubelet-client		system:masters	kubernetes	Mar 31 09:00:09 2022 GMT
kube-apiserver	Client (Kubelet)						
kube-apiserver	Client (Etcd)	apiserver-etcd-client.crt	kube-apiserver-etcd-client		system:masters	etcd-ca	Mar 31 09:00:12 2022 GMT
kube-apiserver	Client (Etcd)						
kube-apiserver	CA	/etc/kubernetes/pki/ca.crt	kubernetes			kubernetes	Mar 29 09:00:09 2031 GMT
kube-apiserver	CA	/etc/kubernetes/pki/ca.key					

Занесем эту информацию в таблицу. Выполним ту же процедуру, чтобы определить информацию обо всех других сертификатах. Их необходимо проверить все, чтобы убедиться, что наша таблица актуальна и имеет правильные имена, правильные альтернативные имена, что сертификаты принадлежат правильной организации и, что наиболее важно, они выпущены правильным эмитентом и не просрочены.

Требования к сертификату подробно расписаны на страницах документации Kubernetes, посмотри эту ссылку.

Когда мы сталкиваемся с проблемами, обычно мы смотрим логи. Здесь траблешутинг начинается с этого же.

Если кластер настроен с нуля самостоятельно и компоненты настроены как нативные службы в ОС, они используют стандартные функции ведения логов, определенные в этой ОС.

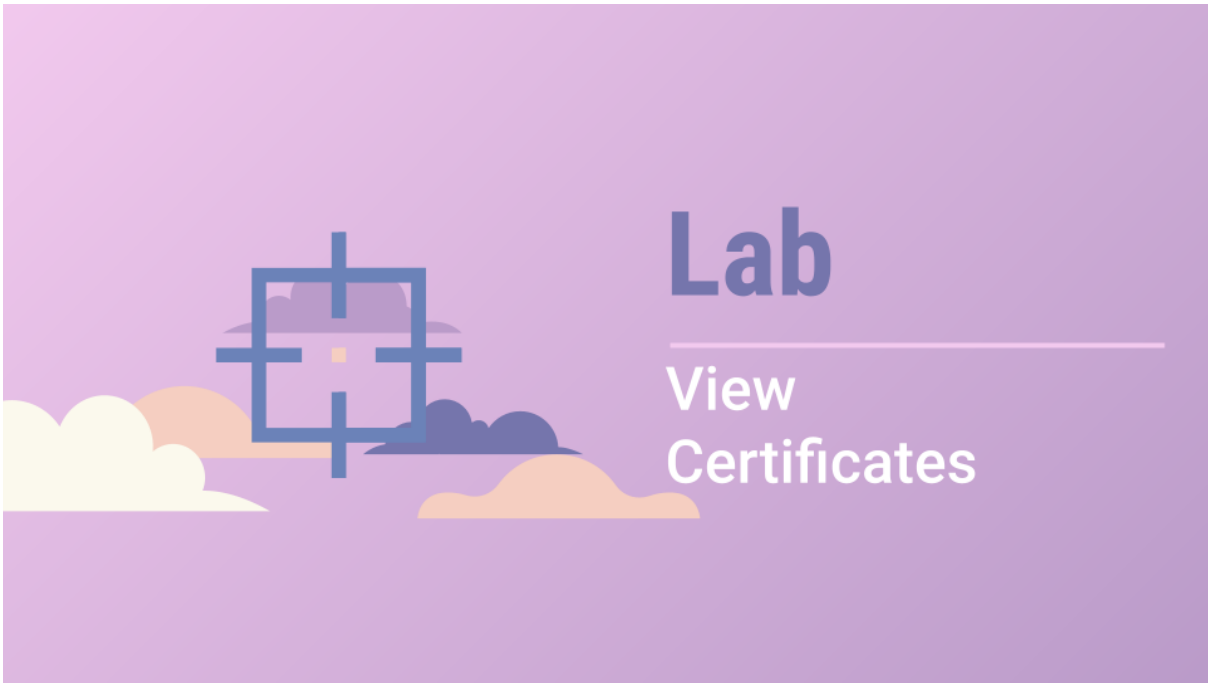
Если был настроен с помощью kubeadm, тогда различные компоненты развертываются как PODs. Таким образом, ты можешь просмотреть логи с помощью команды kubectl logs, за которой следует имя POD.

Иногда, если основные компоненты, такие как API-сервер или ETCD-сервер, не работают, команда kubectl тоже не работает. В этом случае тебе нужно будет спуститься на уровень ниже, в Docker или другой container runtime и посмотреть уже их логи командой контейнера.

Например, здесь я вывел список всех контейнеров с помощью команды docker ps –a. Затем просмотрел журнал логов с помощью команды docker logs, за которой следует идентификатор контейнера.

Ну вот и все в этой лекции. Переходи к тесту и закрепи в существующем кластере то, о чем мы тут говорили.

Увидимся на следующей.



Привет и добро пожаловать на лекцию. В этой лекции мы рассмотрим, как управлять сертификатами и что такое API сертификатов в Kubernetes.

Итак, что у нас происходит в данный момент.

Администратор кластера по имени Вова в процессе настройки всего кластера создал сертификаты центра сертификации на мастере и издал множество сертификатов для различных компонентов.

Затем он запустил службы, используя правильные сертификаты, и все это работает. И он единственный администратор и пользователь кластера, у него есть собственный сертификат администратора и ключ.

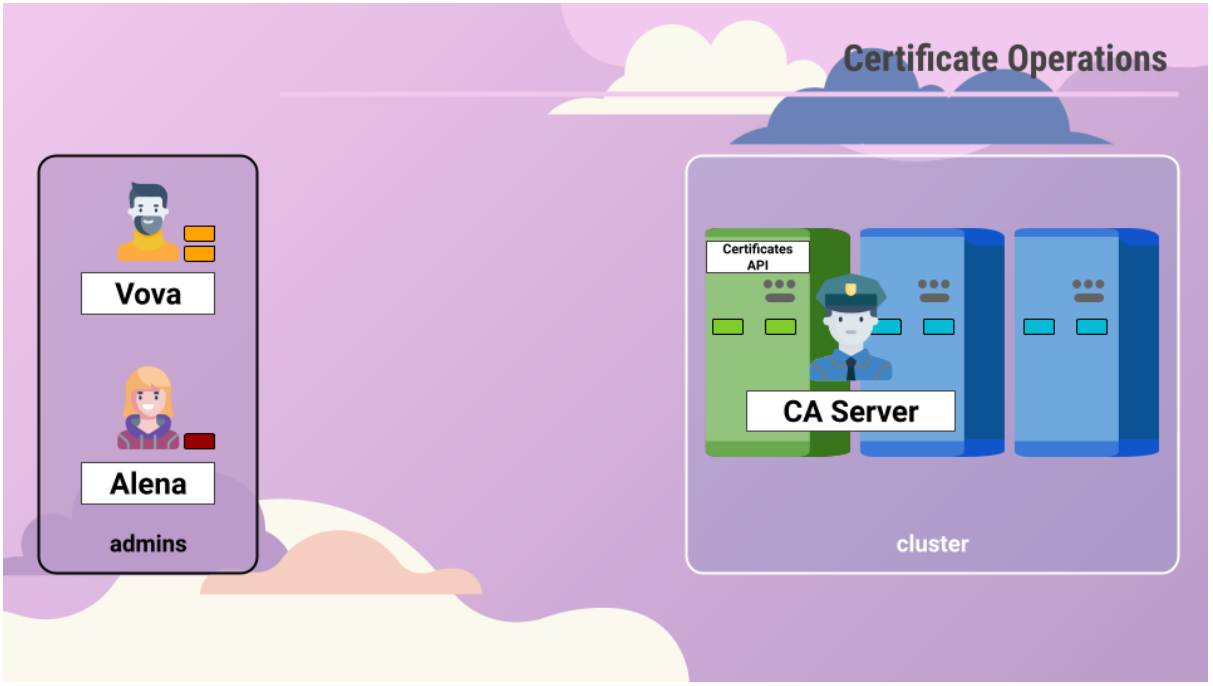
Через некоторое время к нему в команду пришла новая сотрудница-администратор, ее зовут Алена. Ей нужен доступ к кластеру. Вове нужно предоставить Алене заверенный сертификат, чтобы она могла получить доступ к кластеру.

Она создает свой закрытый ключ, генерирует запрос на подпись сертификата и отправляет его Вове, поскольку только он единственный администратор в кластере.

Вова пересылает CSR-запрос на своей сервер центра сертификации, где он подписывает его с помощью закрытого ключа и корневого сертификата сервера CA. Тем самым он создает сертификат, а затем отправляет его Алене.

Теперь у нее есть собственная действующая пара сертификата и ключа, которые она может использовать для доступа к кластеру.

Сертификаты имеют подтвержденный период, который заканчивается по прошествии определенного промежутка времени.



Каждый раз, когда он истекает, мы следуем тому же процессу создания нового CSR и получения подписи в CA. И так мы продолжаем ротацию файлов сертификатов на постоянной основе.

Теперь поговорим о сервере CA.

Что такое служба CA и где она находится в Kubernetes? Центр сертификации Kubernetes на самом деле лишь пара файлов ключ-сертификат, которые мы сами сгенерировали.

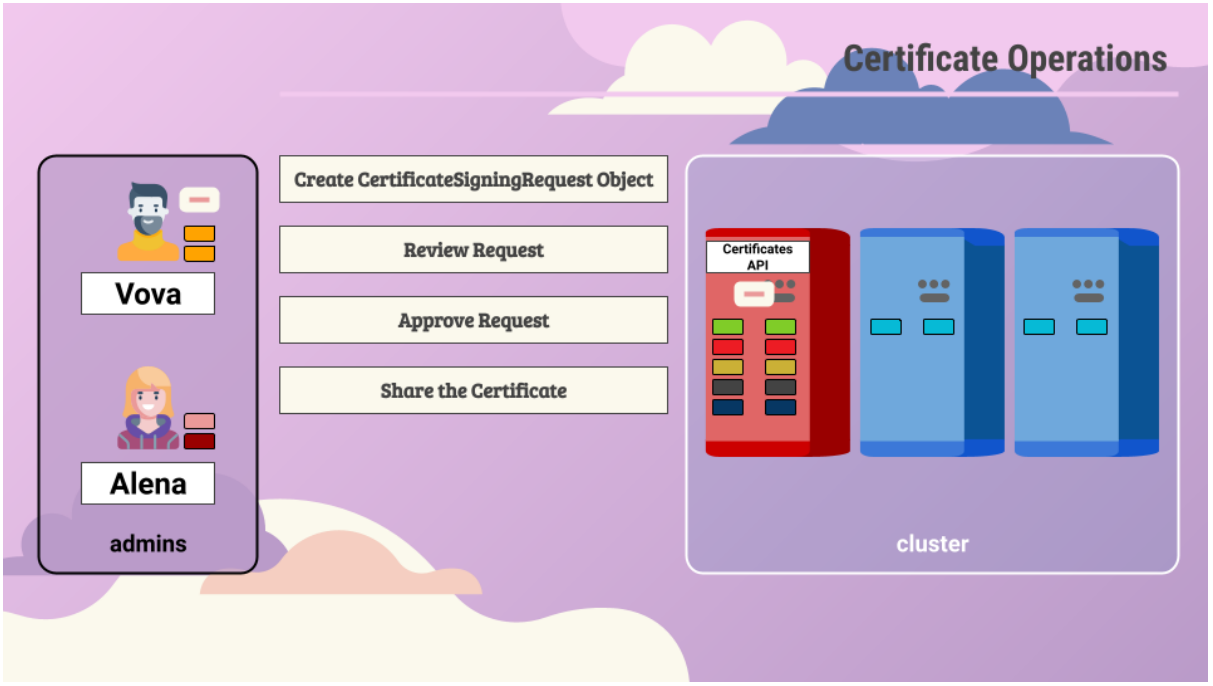
И это значит, что кто бы ни получил доступ к этой паре файлов, он сможет подписать любой сертификат для кластера, где утверждены эти корневые сертификаты.
Они могут создать столько пользователей, сколько захотят, и с любыми привилегиями.

Таким образом, эти файлы необходимо защитить и хранить в безопасной среде, скажем, разместить их на полностью защищенном сервере.

И этот сервер становится нашим сервером CA. Файл ключа сертификата надежно хранится на этом сервере, и не покидает его. Каждый раз, когда нам нужно подписать сертификат, вы можем сделать это, только войдя на этот сервер.

На данный момент у нас есть сертификаты, размещенные на самом мастер-узле Kubernetes. Таким образом, этот главный узел также является нашим сервером CA.

Ок, утилита kubeadm делает то же самое. Она создает пару файлов CA и сохраняет ее на мастер-узле.



До сих пор мы подписывали запросы вручную, но по мере того, как количество пользователей увеличивалось, а наша команда росла, нам потребовался более эффективный способ управления запросами на подписание сертификатов, а также ротацией сертификатов по истечении срока их действия. Нам понадобилась автоматизация.

Kubernetes имеет встроенный API сертификатов, который может снять с нас рутинные операции. С помощью Certificates API мы просто отправляем CertificateSigningRequest прямо в Kubernetes через вызов API.

В этот раз, когда администратор получает запрос на подпись сертификата вместо того, чтобы войти на мастер и подписать сертификат самостоятельно, он создает объект API Kubernetes под названием CertificateSigningRequest.

После создания объекта все сертификаты и запросы на подписание могут быть просмотрены администраторами кластера. Запрос можно легко просмотреть и утвердить с помощью команд kubectl, затем этот сертификат можно извлечь и передать пользователю.

Посмотрим, как это делается.

Пользователь Алена сначала создает ключ. Затем генерирует запрос на подпись сертификата, используя ключ с ее именем в нем. Затем отправляет запрос администратору.

Администратор берет ключ и создает объект запроса сертификата.

CertificateSigningRequest создается, как и любой другой объект Kuberenetes, с использованием файла манифеста с обычными полями.

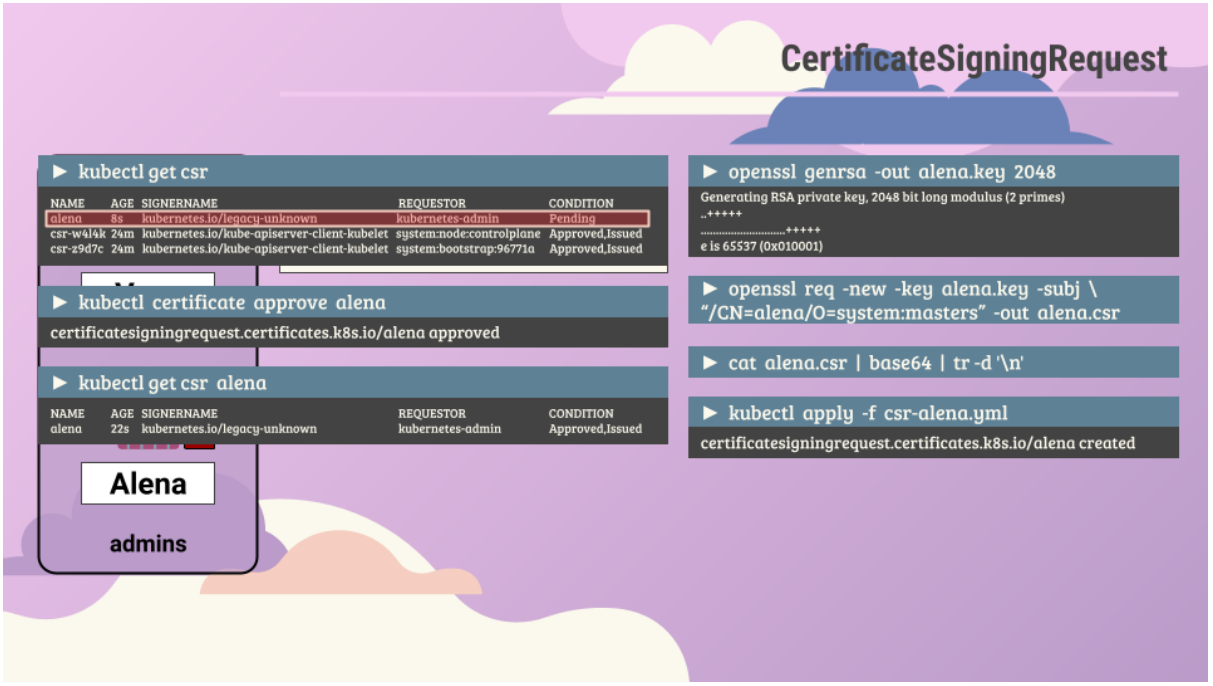
Kind - CerificateSingingRequest. В разделе спецификации укажи группы, в которые должен входить пользователь. Далее укажи, как будет использоваться эта учетная запись.

Эти два последние параметра представляют собой list of strings в терминах YAML.

В поле запроса мы указываем запрос на подпись сертификата от пользователя, но не в виде простого текста. Вместо этого он должен быть закодирован с помощью команды base64, а уже потом эта кодированая версия должна быть вставлена в поле `request`. Теперь отправим этот запрос командой создания объекта Kubernetes.

Все запросы на подпись сертификата могут быть просмотрены администраторами, для этого выполни команду

kubectl get csr



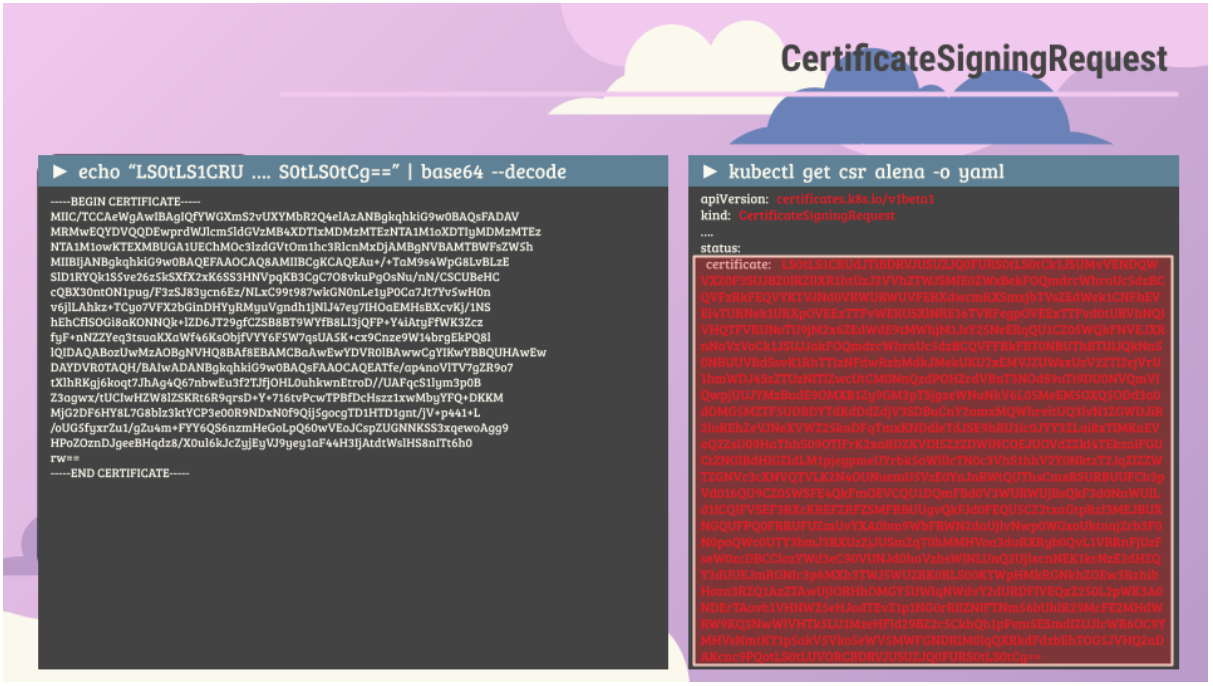
Выберем из списка новый запрос, его состояние будет как `pending`. И далее утвердим его, выполнив команду

kubectl certificate approve

Kubernetes подписывает сертификат, используя пары ключей CA, и генерирует

сертификат для пользователя.

После этот сертификат может быть извлечен и предоставлен пользователю.



Чтобы просмотреть содержимое сертификата, добавь в команду опцию вывода в формате YAML. Сгенерированный сертификат будет находится под полем `certificate`. Но, как и CSR, будет закодирован. Для его декодирования используй команду base64 с опцией --decode. Это даст сертификат в простом текстовом виде. Теперь он может быть передан конечному пользователю.

Теперь, когда мы увидели, как это работает, давай посмотрим, кто все это делает за нас.

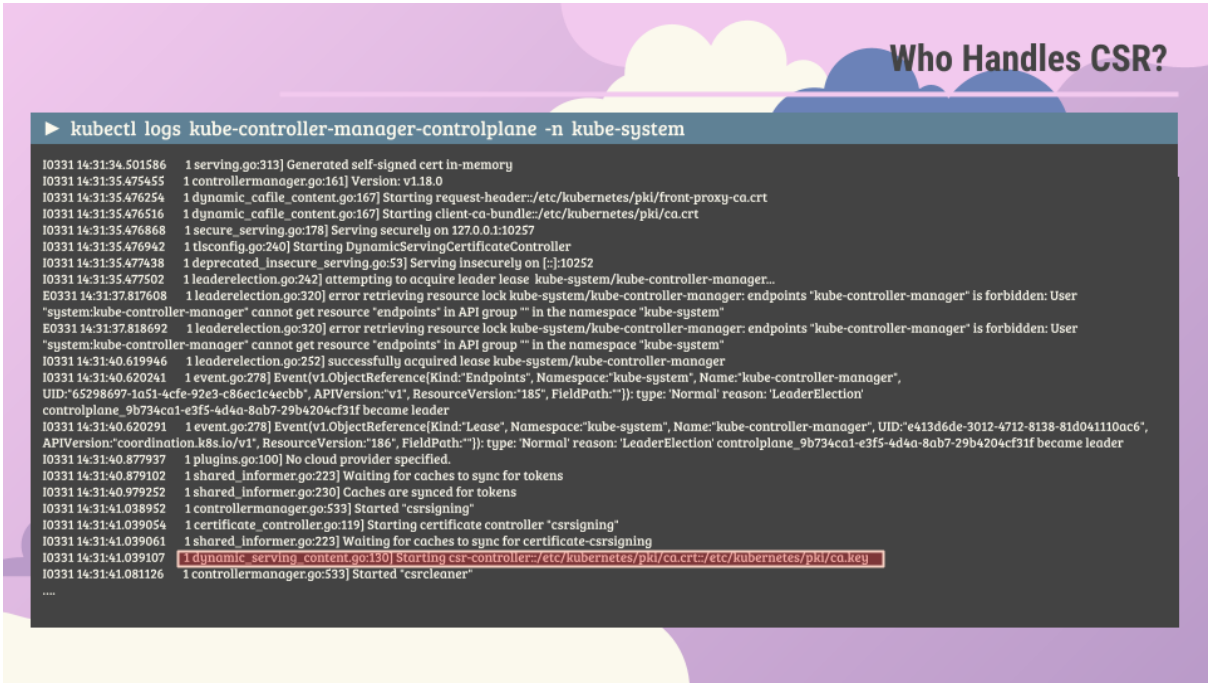
Если мы посмотрим на плоскость управления Kubernetes, мы увидим сервер kube-api, планировщик, контроллер-менеджер, сервер ETCD.

Какой из этих компонентов фактически отвечает за все операции, связанные с сертификатом?

Ну ETCD - это хранилище, API-сервер координатор и почтальон, планировщик - вещь в себе, у него очень узкие задачи. Остается controller-manager, который, как мы говорили, реализует интеллект Kubernetes.

Действительно, все операции, связанные с сертификатом, выполняются контроллер-менеджером.

Если мы внимательно посмотрим на controller-manager, то увидим, что в нем есть контроллеры, называемые csr-approving, csr-signed, csr-cleaner. Они-то и отвечают за выполнение этих конкретных задач.



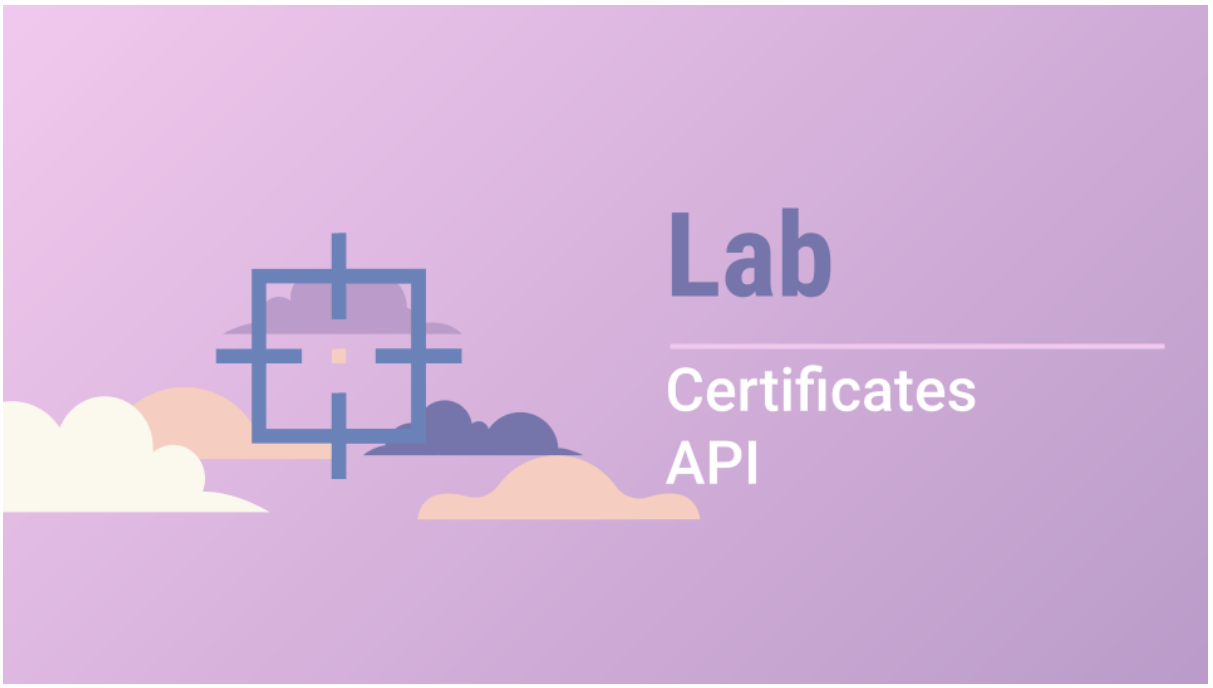
Мы знаем, что если кому-то нужно подписывать сертификаты, им понадобится корневой сертификат и закрытый ключ CA-сервера.

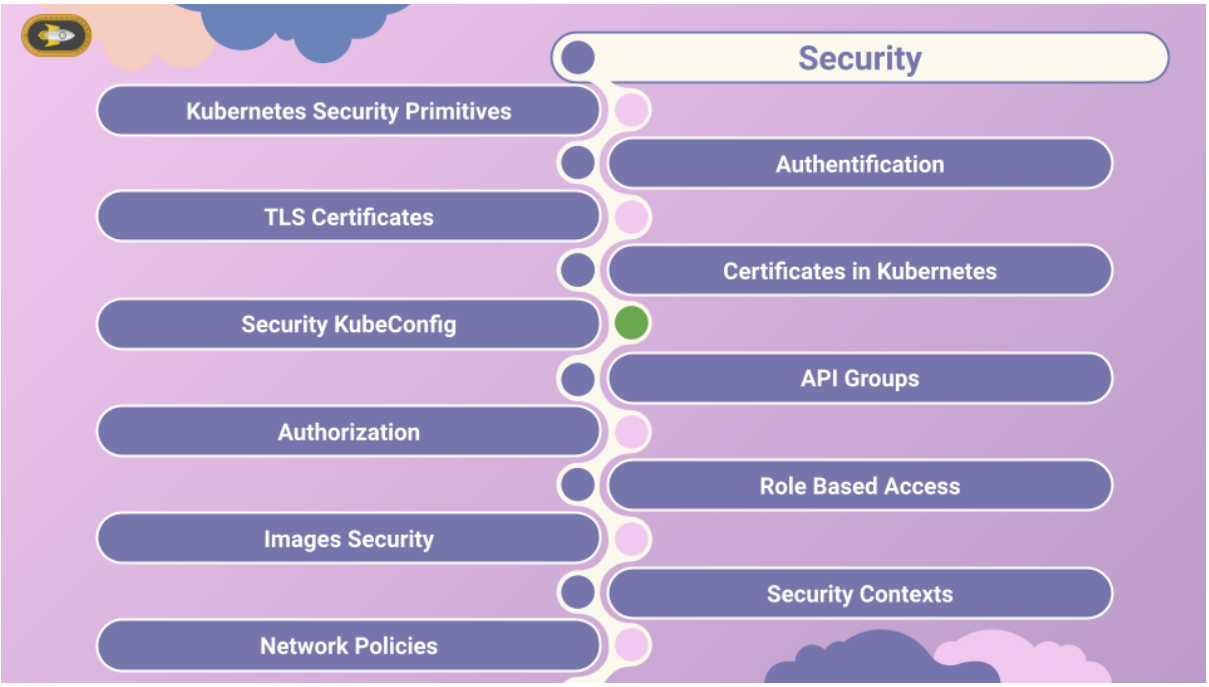
В конфигурации службы controller-manager есть опции, в которых указывается пути расположения корневых сертификатов. А заглянув в логи controller-manager, ты можешь увидеть следы работы этих контроллеров.

Это все в этой лекции.

Поиграй с API сертификатов на практике.

Увидимся на следующей лекции.





Привет и добро пожаловать на лекцию. Сегодня мы поговорим о KubeConfig в Kubernetes.

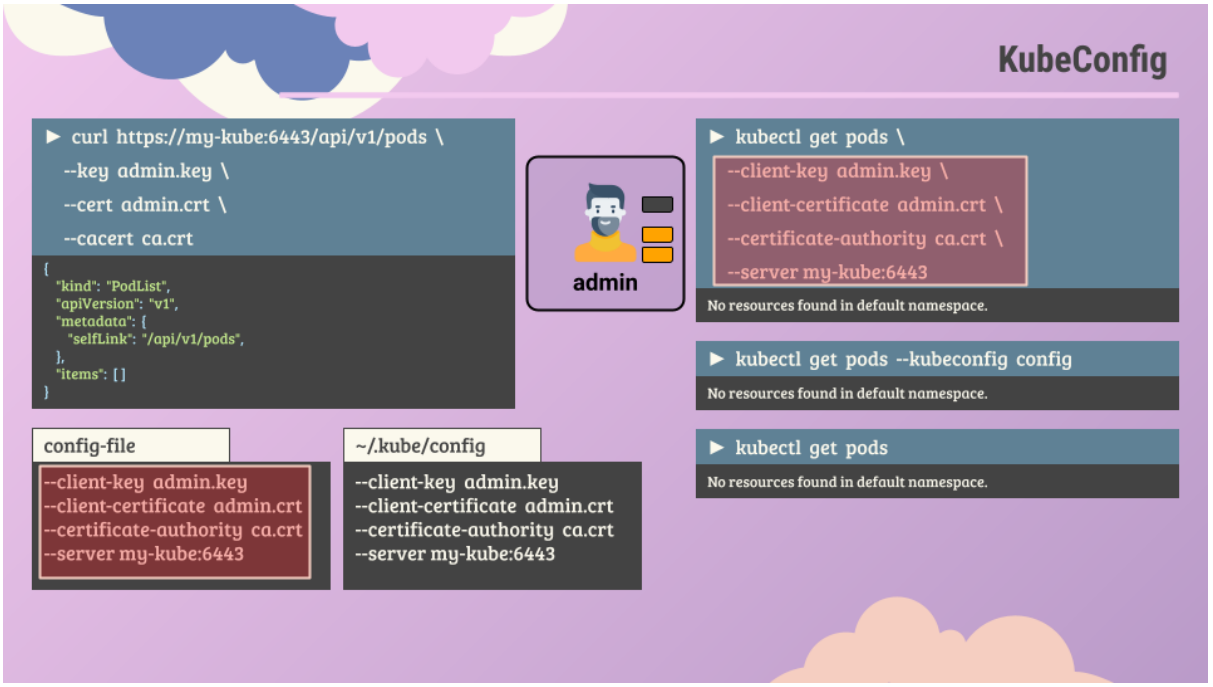
Пока мы видели, как сгенерировать сертификат для пользователя. Мы видели, как клиент использует файл сертификата и ключ для запроса в Kubernetes Rest API для получения списка PODs с помощью утилиты curl.

В этом случае мой кластер называется my-kube, я отправил запрос из curl на адрес kube-apiserver, передав ему в параметрах команды свою пару ключ-сертификат и также добавил сертификат CA. С их помощью API-сервер аутентифицировал меня и выдал результат в JSON-формате.

Как это сделать с использованием команды kubectl? Мы можем указать ту же информацию, используя специальные опции, указав ключ клиента, сертификат клиента и сертификат CA для утилиты kubectl.

Очевидно, что вводить их каждый раз - утомительное занятие. Таким образом, в Kubernetes есть механизм, с помощью которого мы можем переместить всю эту информацию в файл конфигурации с именем kube-config. Затем укажем этот файл как параметр --kubeconfig в своей команде для kubectl.

По умолчанию инструмент kubectl ищет файл с именем config в папке `.kube` домашнего каталога пользователя. Поэтому, если ты создашь там файл kubeconfig, тебе не придется явно указывать путь к файлу в команде kubectl.
По этой причине мы обычно не указываем никаких параметров для команд kubectl.



Файл kubeconfig имеет определенный формат.
Давай разберем его. Конфигурационный файл состоит из 3-х разделов. Clusters, Users и Contexts. Кластеры - это различные кластеры Kubernetes, к которым нам нужен доступ.

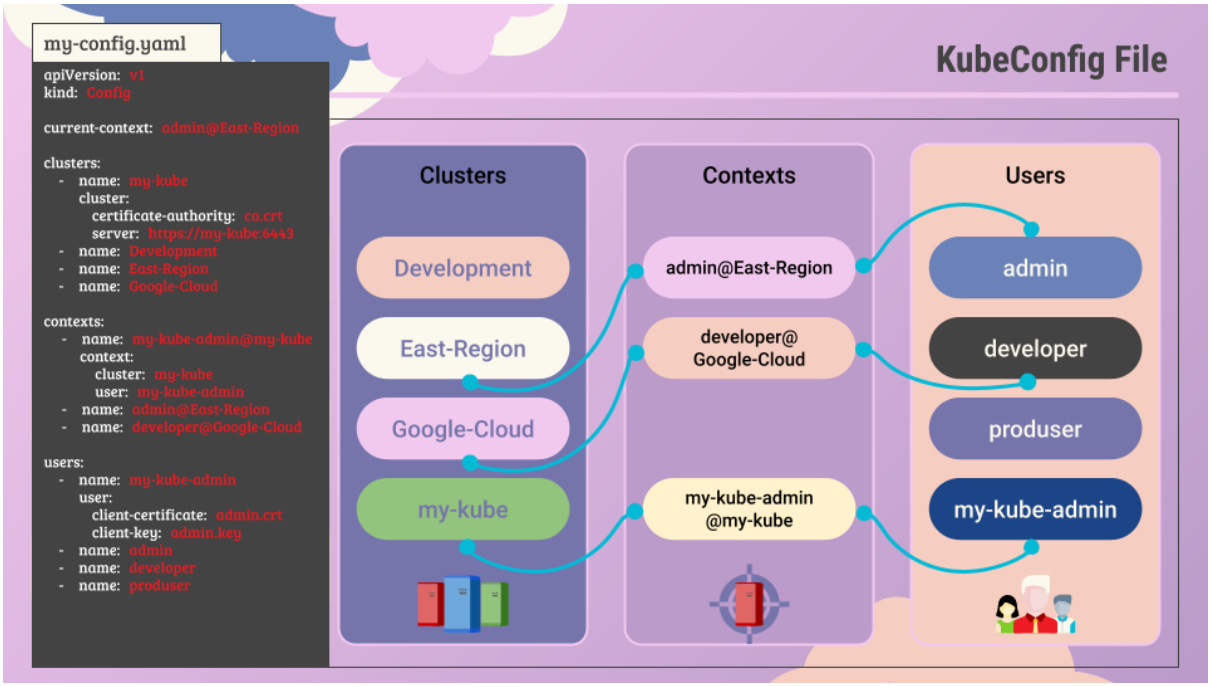
Скажем, у нас есть несколько кластеров для среды разработки, среды тестирования или продуктовой среды, или же это кластера для разных организаций, или находятся у разных облачных провайдеров. Вариантов масса.

Во всех кластерах, которые мы тут указали есть разные пользователи, которые имеют туда доступ. Так чтобы kubectl мог обратиться к этим кластерам, нам нужно будет записать всех пользователей в файл.
Например, это будут admin, developer, produser. Эти пользователи могут иметь разные привилегии в разных кластерах.

Наконец, контексты объединяют их вместе. Контекст определяет, какая учетная запись пользователя будет использоваться для доступа к какому кластеру.
Например, мы можем создать контекст с именем admin для восточного региона. Контекст будет использовать учетную запись администратора для доступа к кластеру `East-Region`.

А чтобы протестировать развертывание приложения, мы бы могли использовать данные пользователя developer, которого создали для работы в кластере в Google-Cloud.

Помни, что здесь мы не создаем новых пользователей и не настраиваем какой-либо доступ или авторизацию пользователей в кластере.



В этом процессе мы используем существующих пользователей с их существующими привилегиями и лишь определяем, какого пользователя использовать для доступа к определенному кластеру.

Используя это, нам не нужно указывать сертификаты пользователя и адрес сервера в каждой выполняемой команде kubectl.

Итак, как это ложиться на наш пример.

Опции, относящиеся к аутентификации пользователя, я положу ближе к users, а настройки сервера, положу к clusters. Объединю эти две половинки новым контекстом.

Т.о. спецификация сервера из нашей команды входит в раздел кластера, ключи и сертификаты пользователя с правами администратора входят в раздел пользователей. Контекст, указывает на использование пользователя mykube-admin для доступа к кластеру my-kube.

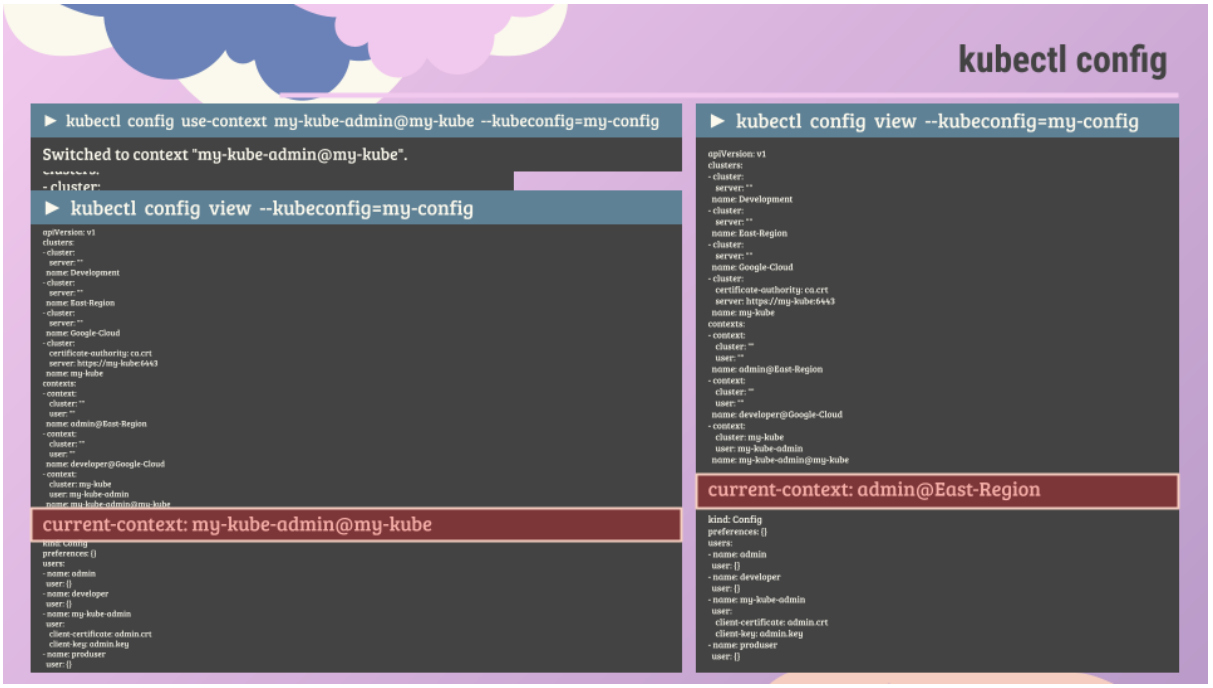
Давай теперь посмотрим на настоящий файл KubeConfig.

KubeConfig пишется в формате YAML.

Его apiVersion v1, kind Config. И, как мы уже обсуждали, в нем 3 раздела.
Один для кластеров - clusters, один для контекстов - contexts и один для пользователей - users.

Каждый из них представляет собой list.

Таким образом, ты можешь указать несколько кластеров, пользователей или контекстов в одном файле.



Под clusters мы добавляем новый элемент, это будет наш кластер my-kube. Поле name будет my-kube, поле cluster у нас dictionary, там укажем адрес сервера в поле server и корневой сертификат CA кластера.

Потом мы добавляем запись в раздел пользователей, чтобы указать сведения о моем my-kube-admin. Здесь укажем расположение сертификата и ключа этого пользователя.

Итак, мы определили кластер и пользователя, который постучиться в двери. Теперь нужно сказать, в двери какого кластера ему стучаться.

Мы создаем запись в разделе contexts, чтобы связать вместе кластер my-kube и пользователя my-kube-admin. Назовем контекст my-kube-admin@my-kube. Далее в поле context выставим поля cluster в my-kube, а user - в my-kube-admin.

Следуя такой же процедуре, мы создадим контексты для оставшихся трех кластеров и их пользователей. Сделаем это так, чтобы файл отражал связи, нарисованные на схеме.

Помни, что нам не нужно создавать какой-либо объект, как мы обычно делаем для других объектов Kubernetes. Этот файл остается как есть на твоей локальной машине и читается командой kubectl. Она берет оттуда нужные значения и использует их при установлении соединения с кластером.

Теперь, как kubectl узнает, какой контекст выбрать? Мы определили здесь три контекста, с которого ему начинать?

Мы можем указать используемый по умолчанию контекст, добавив поле `current-context` в файл KubeConfig. В этом поле укажи имя используемого контекста.

В этом случае kubectl всегда будет использовать контекст admin@East-Region. Т.е. по умолчанию kubectl будет ходить в кластер East-Region, представляясь как admin.

В kubectl есть команда `config` для просмотра и изменения файлов KubeConfig. Для просмотра текущего используемого файла запусти команду kubectl config view.

В ней будут перечислены кластеры, контексты и пользователи, а также установка текущего контекста.

Как мы обсуждали ранее, если ты не укажешь, какой файл KubeConfig использовать, он будет использовать файл по умолчанию, расположенный в папке .kube в домашнем каталоге пользователя.

В качестве альтернативы ты можешь указать файл KubeConfig, передав параметр `--kubeconfig` в командную строку как ты здесь видишь.
Или мы просто переместим наш файл конфигурации в свой домашний каталог, чтобы он стал нашим файлом KubeConfig по умолчанию.

Итак, как нам обновить свой текущий контекст? Предположим, мы использовали пользователя admin для доступа к East-Region. Как изменить контекст, чтобы использовать my-kube-admin для доступа к кластеру my-kube?

Запусти команду kubectl config use-context, чтобы изменить текущий контекст. Мы поменяем контекст на my-kube-admin@my-kube, как ты видишь в команде. Как видишь kubectl просто отредактировал поле в нашем файле `my-config`.

Да, изменения, внесенные командой kubectl config, действительно отражаются в файле. Ты можешь внести другие изменения в файл KubeConfig или удалить элементы в нем используя другие опции команды kubectl config.
Попробуй это в свободное время.

Ок, а что насчет пространств имен? Например, каждый кластер может быть сконфигурирован с несколькими namespaces внутри него. Можем ли мы настроить контекст, чтобы попадать в конкретное пространство имен?
Да! В раздел контекста в файле KubeConfig можно определить дополнительное поле `namespace`, где мы можем указать в какое именно пространство имен нам следует обращаться.

Таким образом, когда ты переключишься на этот контекст, то автоматически окажешься в этом определенном пространстве имен. Очень удобно.

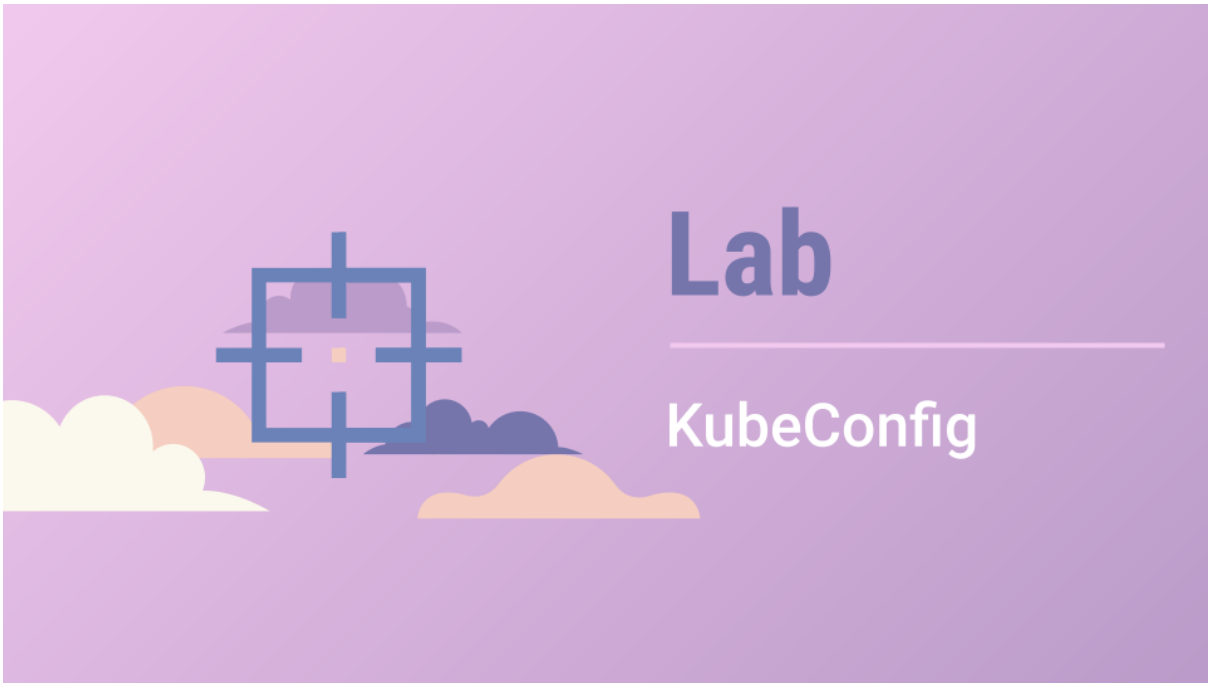
Наконец, о сертификатах. Мы указывали пути к файлам сертификатов, упомянутых в KubeConfig, следующим образом.

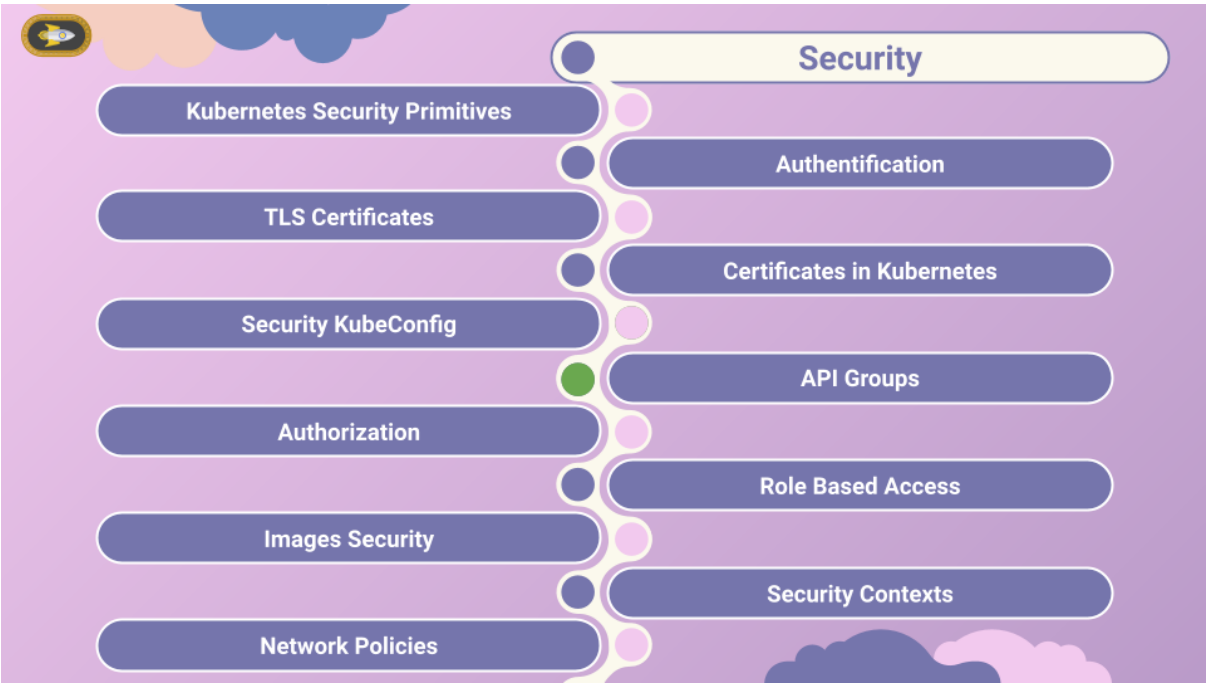
Но все же, лучше использовать полный путь вот так. Также есть замечательная возможность указать данные сертификата в этом же файле.

Давай сделаем это для root-сертификата кластера в кластере East-Region. Справа находится содержимое файла ca.crt. Вместо использования поля `certificate-authority` с путем к файлу мы используем поле `certificate-authority-data` и предоставим содержимое самого сертификата. Но только не файл как есть. Преобразуем его содержимое в формат с кодировкой base64, а затем положим его в кодированом виде.

Точно так же, если потребуется его декодировать, используй опцию `--decode` команды base64.

Ну вот и все в этой лекции. Перейди в раздел практики и попробуй поустранять неполадки с файлами KubeConfig. Жду тебя на следующей лекции!


Lab
KubeConfig



Привет и добро пожаловать.

Прежде чем мы перейдем к авторизации, необходимо разобраться в группах API в Kubernetes.

Но сначала, что такое Kubernetes API?

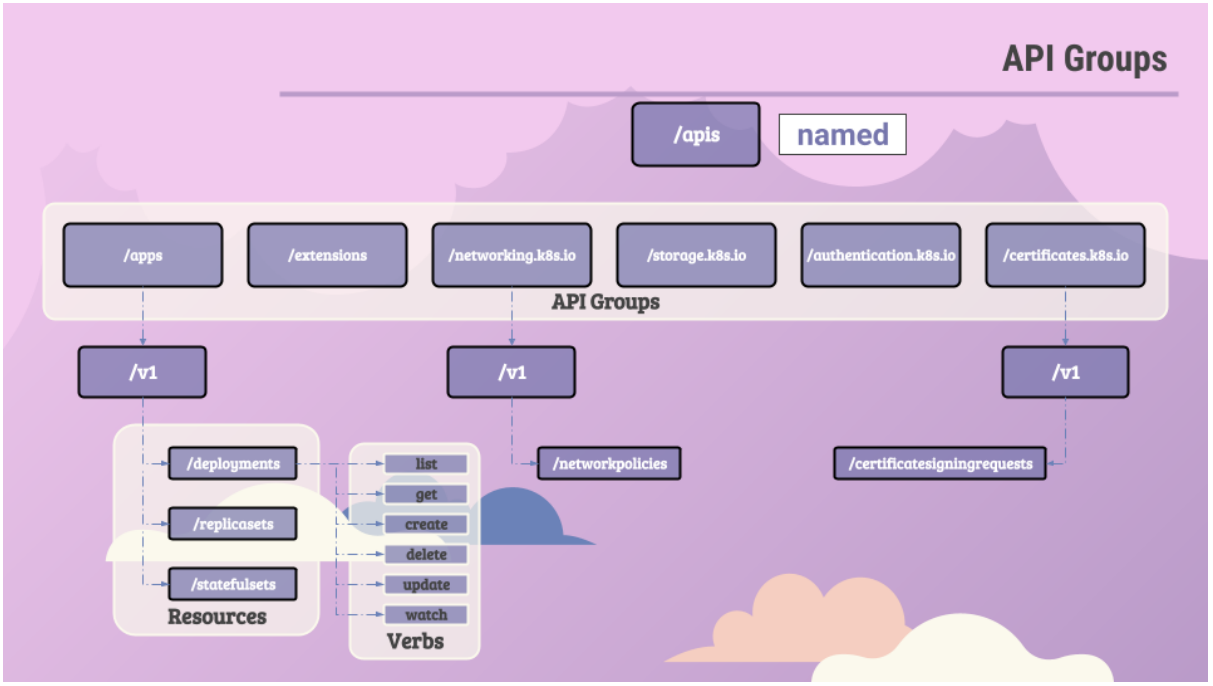
Мы узнали о сервере kube-API. Какие бы операции мы ни выполняли с кластером, мы так или иначе взаимодействовали с API-сервером. Либо через утилиту kubectl, либо напрямую.

Скажем, мы хотим проверить версию. Можно получить доступ к серверу API по адресу мастер-узла, за которым следует порт, который по умолчанию равен 6443, и сам API, в данном случае `version`.

Это вернет версию. Точно так же, чтобы получить список PODs, мы должны обратиться к URL-адресу api/v1/pods. Как иногда говорят дернуть сервер на API `pods`. Это вернет список PODs.

В этой лекции мы сосредоточимся на этих путях API. Итак `/version` и `/api`. Это группы. Kubernetes API сгруппирован в несколько таких групп в зависимости от их назначения. Например, metrics, healthz, api, apis, version, logs etc.

API version предназначен для просмотра версии кластера, как мы только что увидели. PI metrics и healthz используются для мониторинга работоспособности кластера. LOgs для интеграции со сторонними приложениями для ведения логов.
Их на самом деле много, а мы давай сфокусируемся на API, отвечающих за функциональность кластера.



Эти API делятся на две категории. Core-group и named-group. Группа ядра - это то место, где существуют все основные функции. Такие как namespaces, PODs, replication controller, events, endpoints, nodes, bindings, persistent volumes, persistent volume claims, configmaps, secrets, services и т. д.

API именованных групп более организован, и в будущем все новые функции стали размещаться в этих названных группах. В нем есть группы для приложений, расширений, сети, хранилища, аутентификации, сертификатов авторизации и т. д. Здесь показаны лишь некоторые из них. В приложениях у нас есть deployments, replicasets, statefulsets. В сети есть networkpolicies. У сертификатов есть sertificatesingingrequest, о которых мы говорили ранее в этом разделе.

Итак, те, что вверху, являются API-groups, а те, что внизу, - resources в этих API-groups.

У каждого ресурса в этом есть набор связанных с ним действий. Это то, что мы можем делать с этими ресурсами. Например, перечислить deployments (list), получить информацию об одном из них (get), создать deployment, удалить его, обновить просмотреть и т. д. Это называется глаголами - verbs. Документация Kubernetes подробно рассказывает об этом на страницах API reference. Там находится автоматически сгенерированная информация о каждой API группе и ее объектах.

Но мы можем просмотреть это и в своем кластере Kubernetes. Обратись к своему kube-apiserver по порту 6443 без указания пути, и он покажет тебе доступные группы API.

Это действительно длинный список, его можно отфильтровать слову `name` для получения named-groups.

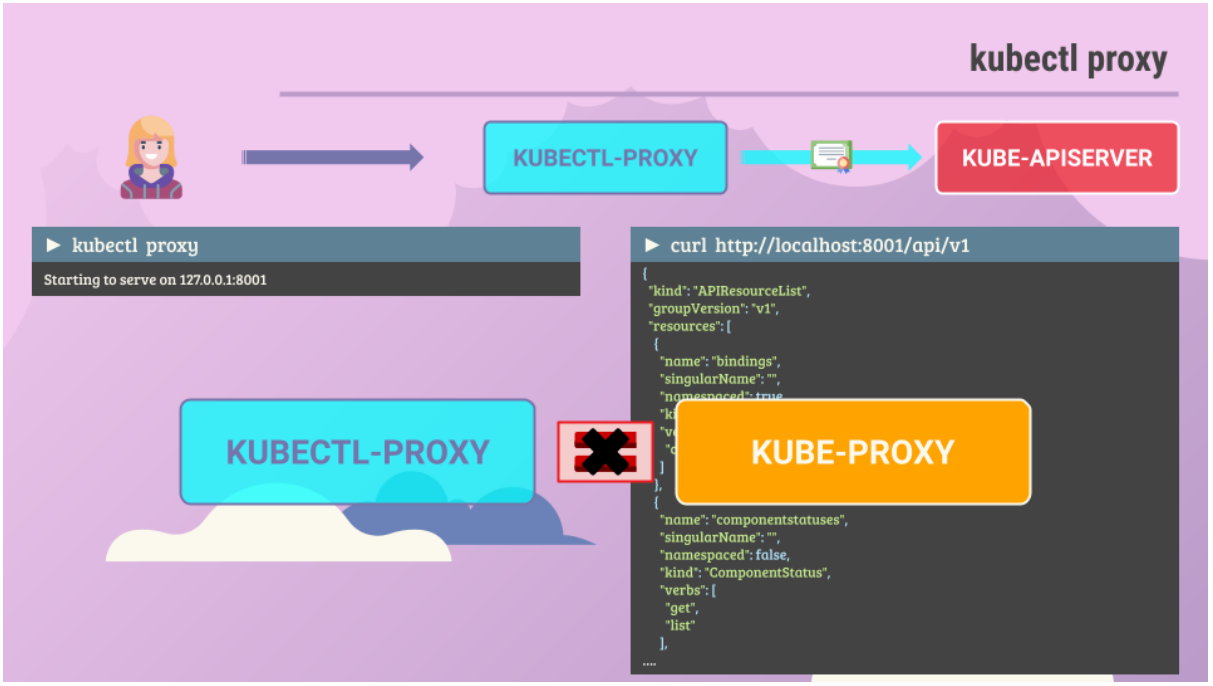
Тут есть небольшой момент о таком доступе к кластерному API.

Если ты обращаешься к API напрямую через curl, как показано здесь, то тебе не будет разрешен доступ, за исключением определенных API, таких как version, поскольку ты не указал какой-либо механизм аутентификации.
Таким образом, тебе нужно пройти аутентификацию в API, используя файлы сертификатов, передав их в командной строке следующим образом.
Еще можно извлечь токен безопасности такой командой, а далее аутентифицироваться с его помощью.

Альтернативный вариант - запустить прокси-клиент kubectl. Команда kubectl proxy запускает прокси-службу локально на порту 8001 и использует учетные данные и сертификаты из твоего файла kubeconfig для доступа к кластеру. Таким образом, тебе не нужно будет указывать их в команде curl.

Теперь ты можешь получить доступ к прокси-службе kubectl через порт 8001, а этот прокси будет использовать учетные данные из файла kubeconfig для перенаправления твоего запроса в kube-apiserver.

Мы обратились в `api/v1` и получили объект APIResourceList для группы v1 в формате JSON. Он тоже не маленький и здесь не поместился полностью.



Ок, еще важный момент для новичков. 2 термина, которые звучат почти одинаково. kube-proxy и kubectl proxy - это не одно и то же.

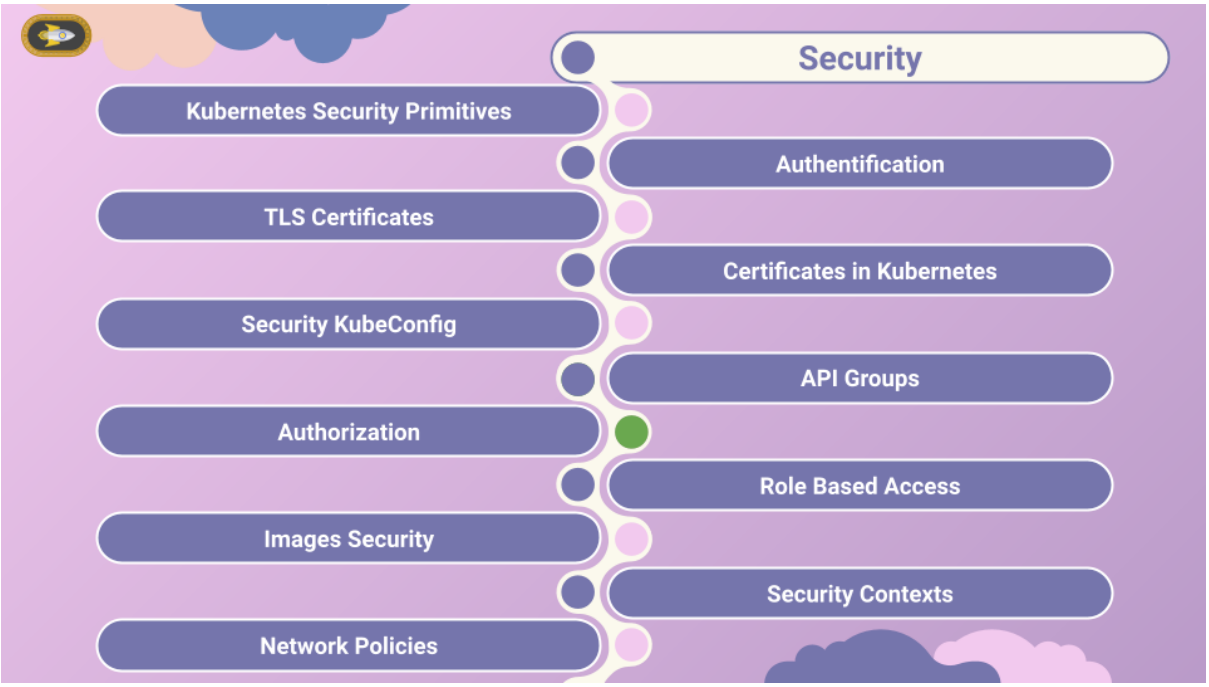
Ранее в этом курсе мы обсуждали kube-proxy. Он используется для обеспечения связи между компонентами и службами на разных узлах кластера. Мы еще обсудим kube-proxy более подробно в разделе networking позже в этом курсе.
В то время как kubectl proxy - это сервис типа HTTP-прокси, который может создать утилита kubectl для доступа к серверу kube-api.

Итак, давай подытожим. Все ресурсы в Kubernetes сгруппированы в разные группы API.

На верхнем уровне у нас есть core-group /api и named-group /apis. В названной группе API у нас есть разные разделы - группы, которые отвечают за разные области функционала Kubernetes. В них есть разные ресурсы, а каждый ресурс имеет набор связанных с ним действий, известных как verbs.

в следующем разделе об авторизации мы узнаем, как их можно использовать, чтобы разрешить или запретить пользователям некоторые действия.

Ну вот и все в этой лекции. Увидимся в следующей.



Привет и добро пожаловать на лекцию об авторизации в Kubernetes.

До этого момента мы говорили об аутентификации. Мы видели, как кто-то может получить доступ к кластеру. Мы увидели разные способы, которыми этот кто-то, будь он человек или машина, может получить доступ к кластеру.

После того, как доступ получен, что он может сделать?

Только то, что именно ему разрешено. И это то, что определяет авторизация.

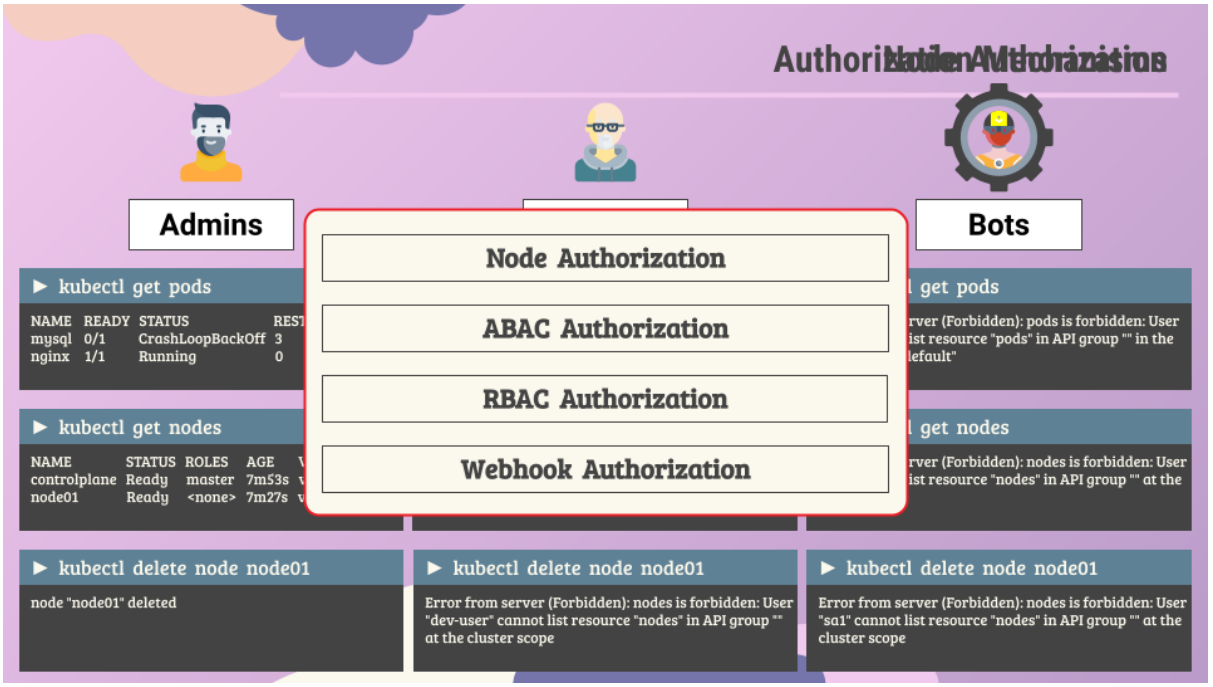
Прежде всего, зачем нам авторизация в нашем кластере? Ведь в качестве администратора, мы можем выполнять в нем всевозможные операции, такие как просмотр различных объектов - nodes, PODs, deployments. Мы можем создать или удалить эти объекты. Или мы даже можем добавить, удалить или заменить какие-то части Kubernetes. Или удалить образующие кластер узлы.

Как администратор, мы можем выполнить любую операцию.

Но вскоре у нас будут и другие, получающие доступ к кластеру, пользователи. Например, другие админы, разработчики, тестировщики или другие приложения, вроде приложений для мониторинга или приложения непрерывной доставки, такие как gitlab т. д.

Для мы будем создавать учетные записи для доступа к кластеру путем создания имен пользователей и паролей, токенов, сертификатов или serviceaccounts, как мы видели в предыдущих лекциях.

Но не разумно давать им всем такой же уровень доступа, как у нас.



Например, мы не хотим, чтобы разработчики имели доступ для изменения конфигурации нашего кластера, например добавления или удаления нод, или конфигураций хранилища, или сетей.

Мы можем разрешить им просматривать это, но не изменять. Хотя они могут иметь доступ для развертывания своих приложений.

То же самое касается сервис-аккаунтов. Мы хотим предоставить минимальный уровень привилегий внешнему приложению, чтобы хватало только для выполнения необходимых операций.

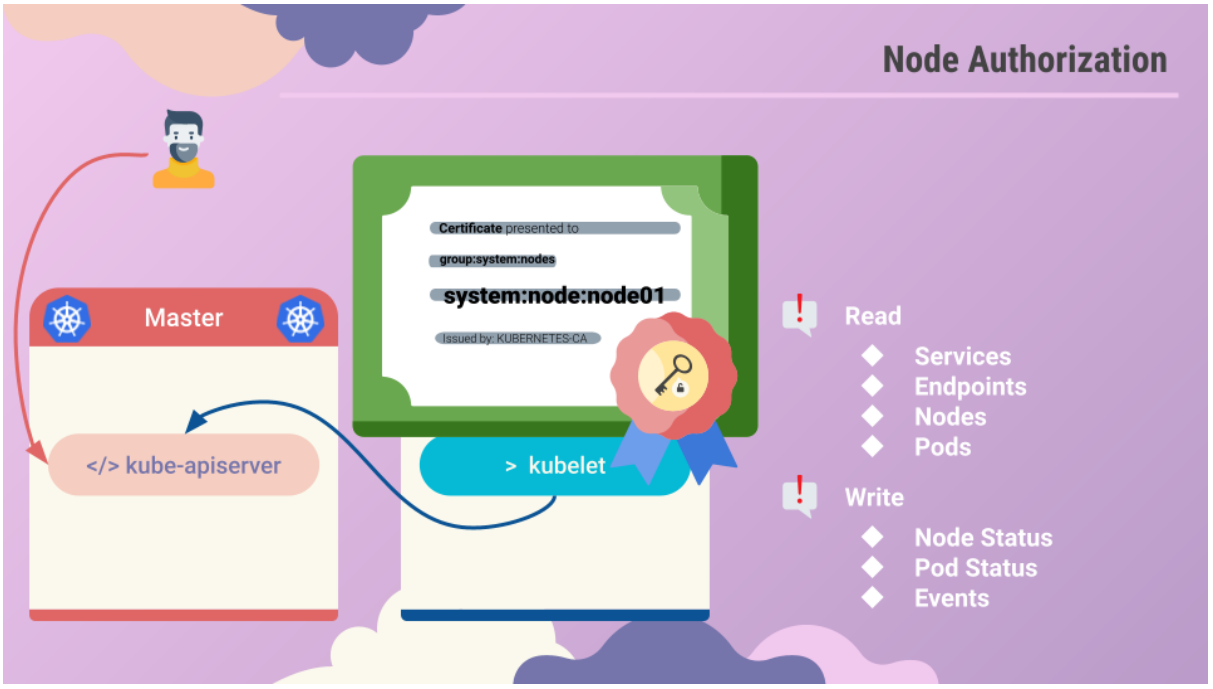
Когда мы разделяем наш кластер между разными организациями или командами, логически деля его с помощью namespaces, мы хотим ограничить доступ пользователей только их пространствами имен. В этом нам поможет авторизация внутри кластера.

Существуют различные механизмы авторизации, поддерживаемые Kubernetes, такие как node authorization, attribute based authorization, role-based authorization и web-hook.

Давай пройдемся по ним.

Мы знаем, что к API-серверу обращаются такие пользователи, как мы, для целей внешнего управления кластером.
Также с нод кластера к нему обращаются kubelets, но для целей внутреннего управления кластером.

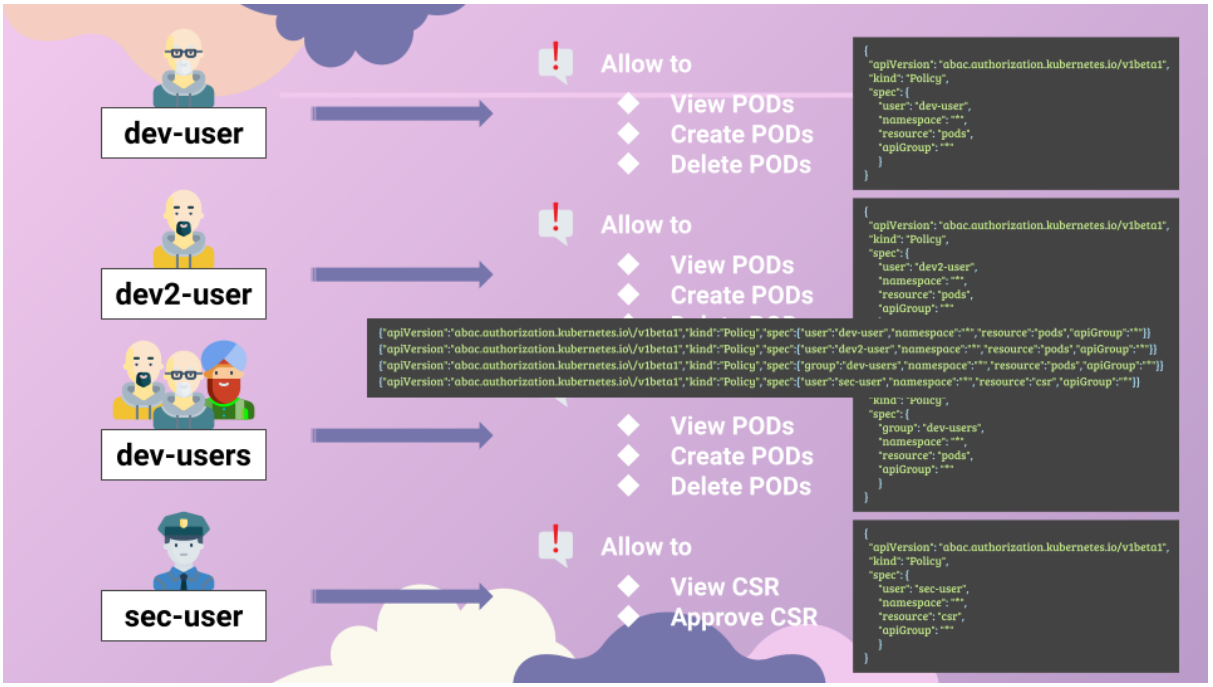
Kubelet, обращается к серверу API для чтения информации об PODs, Nodes, Services, Endpoints, а также рапортует kube-api о статусе своей ноды, событиях на ней и т.д.



Эти запросы обрабатываются специальным авторизатором, известным как node authorizer.

Ранее в лекциях, когда обсуждали сертификаты, мы упоминали, что kubelet должен быть частью группы `system:nodes` и иметь имя с префиксом `system:node`.

Таким образом, любые запросы, поступающие от пользователя с `system:node` в имени и состоящем в группе системных узлов, будут авторизованы через node authorizer и получат эти привилегии. Т.е. получат набор не всех привилегий, а те, что необходимы kubelet для внутреннего менеджмента кластера.

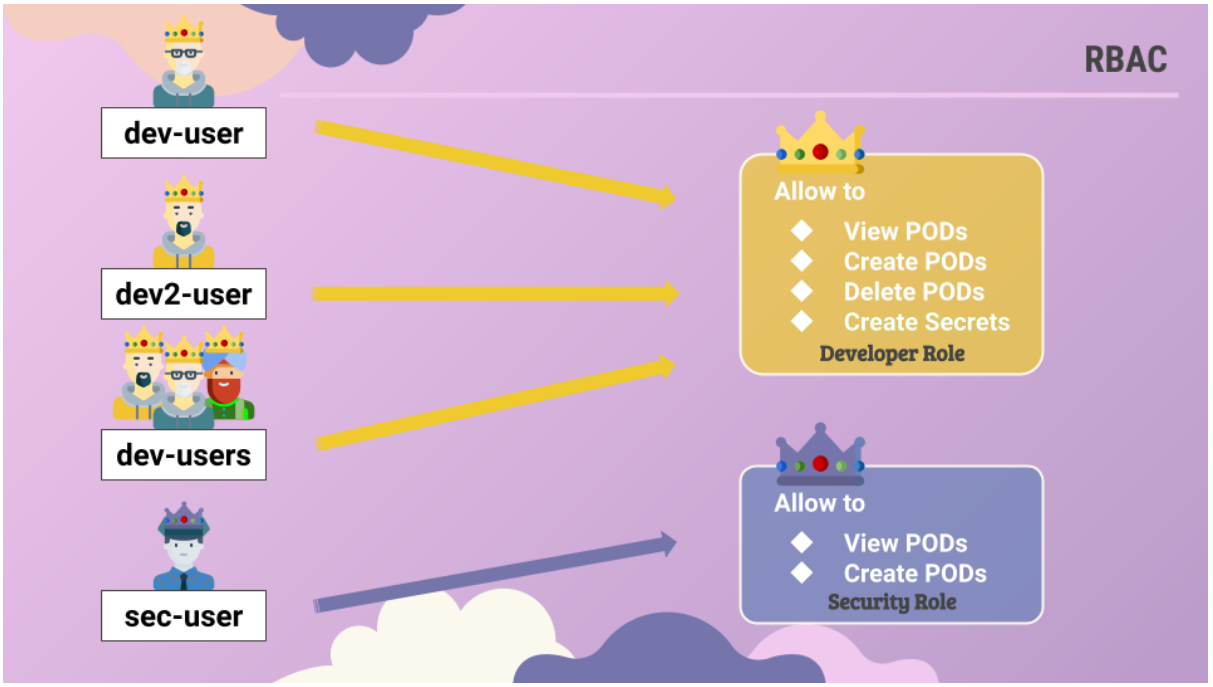


Итак, это о внутреннем доступе. Давай поговорим о внешнем доступе к API.
Например, авторизация на основе атрибутов пользователя ABAC.
Attribute based authorization - это когда мы связываем пользователя или группу пользователей с набором разрешений.

В этом случае мы говорим, что пользователь dev-user может просматривать, создавать и удалять PODs.
Для этого мы создаем policy-файл с набором политик, определенных в формате json.
Далее передаем этот файл на наш мастер и там подключаем его к серверу API.

Точно так же, мы создаем определение политик для каждого пользователя или группы и добавляем в этот файл.
Имей в виду, что политики должны быть быть особенным образом оформлены в файле по одному объекту на линию.
Теперь каждый раз, когда нам нужно добавить или изменить параметры безопасности, нам придется вручную редактировать этот файл политики и перезапускать сервер.
Таким образом, будет сложно управлять конфигурациями при применении ABAC.

Далее мы рассмотрим управление доступом на основе ролей - RBAC.



Управление доступом на основе ролей значительно упрощает менеджмент разрешений.

При использовании RBAC, вместо того, чтобы напрямую связывать пользователя или группу с набором разрешений, мы определяем ее роль для этого набора.

В этом случае роль для разработчиков - Developer Role.

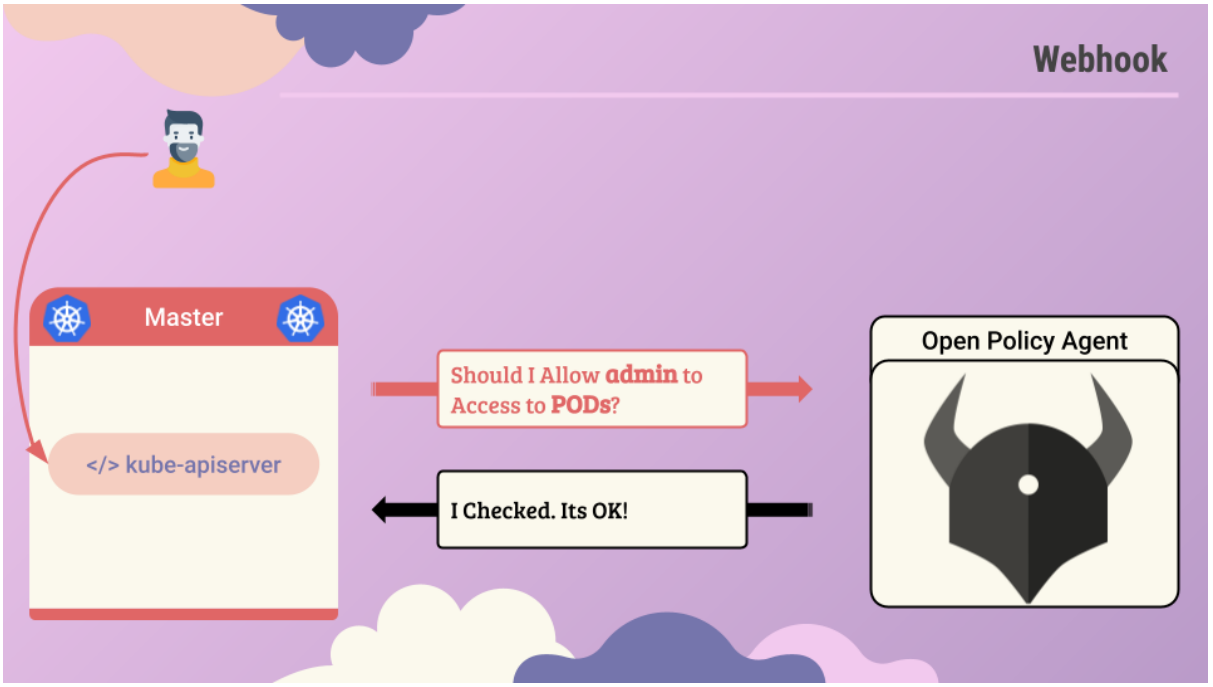
Создаем роль с набором разрешений, необходимых для разработчиков. Затем привязываем к этой роли всех разработчиков, пользователей или группы - неважно.

Точно так же создадим роль для нашего безопасника, с нужным ему набором разрешений. Затем свяжем этого пользователя с этой ролью.

В дальнейшем, всякий раз, когда необходимо внести изменения в доступ пользователей, мы просто изменяем роль, и это сразу же отражается на всех разработчиках.

RBAC обеспечивает более стандартизированный подход к управлению доступом внутри kubernetes-кластера и в следующей лекции мы рассмотрим доступ на основе ролей более подробно.

А пока давай перейдем к другим механизмам авторизации.



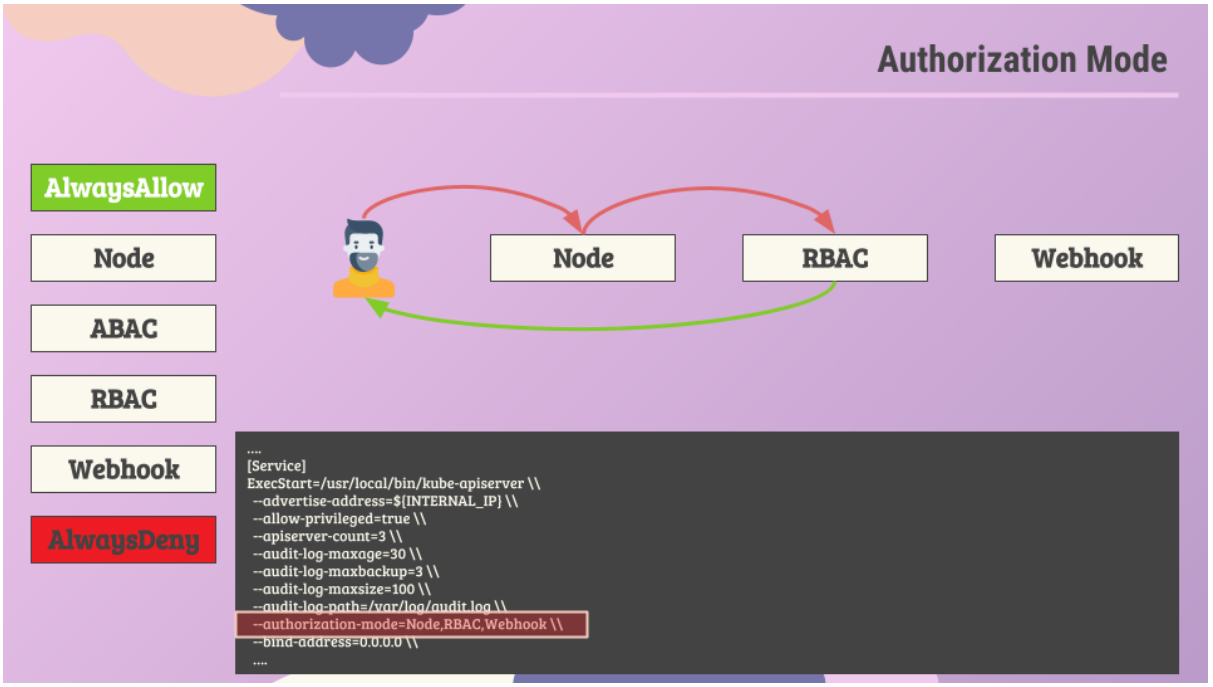
Что, если мы хотим передать все механизмы авторизации кому-то на сторону, на аутсорс, скажем, мы хотим управлять авторизацией извне, а не через встроенные механизмы, которые мы только что обсудили?

Например, использовать Open Policy Agent - это сторонний инструмент, который помогает управлять доступом и авторизацией. Мы можем сделать так, чтобы Kubernetes выполнял вызов в API Open Policy Agent с информацией о пользователе и что он затребовал от kube-api. Получив запрос, уже Open Policy Agent будет решать, следует ли разрешить действие пользователю или нет. И на основании этого ответа API-сервер предоставит пользователю доступ.

Ок, также есть еще два режима, в дополнение к тому, что мы только что видели, это всегда разрешать - AlwaysAllow и всегда запрещать - AlwaysDeny.

Как ясно из их названий они всегда разрешают все запросы, без выполнения каких-либо проверок авторизации, или всегда отклоняют все запросы. Они обычно используются при тестировании.

Итак, где нам настроить эти режимы? Какие из них активны по умолчанию? Можем ли мы иметь больше одного за раз? А если да, то как работает авторизация, если настроено несколько?



Режимы авторизации устанавливаются с помощью параметра --authorization-mode в конфигурации процесса kube-apiserver.

Если мы не укажем этот параметр, по умолчанию он будет AlwaysAllow.

Можно предоставить список нескольких режимов, которые мы хотим использовать, указав через запятую. В этом случае я установил в Node,RBAC,Webhook

Если у нас настроено несколько режимов, то запрос авторизуется с использованием каждого из них в указанном порядке.

Например, когда пользователь отправляет запрос, он сначала обрабатывается node authorizer.

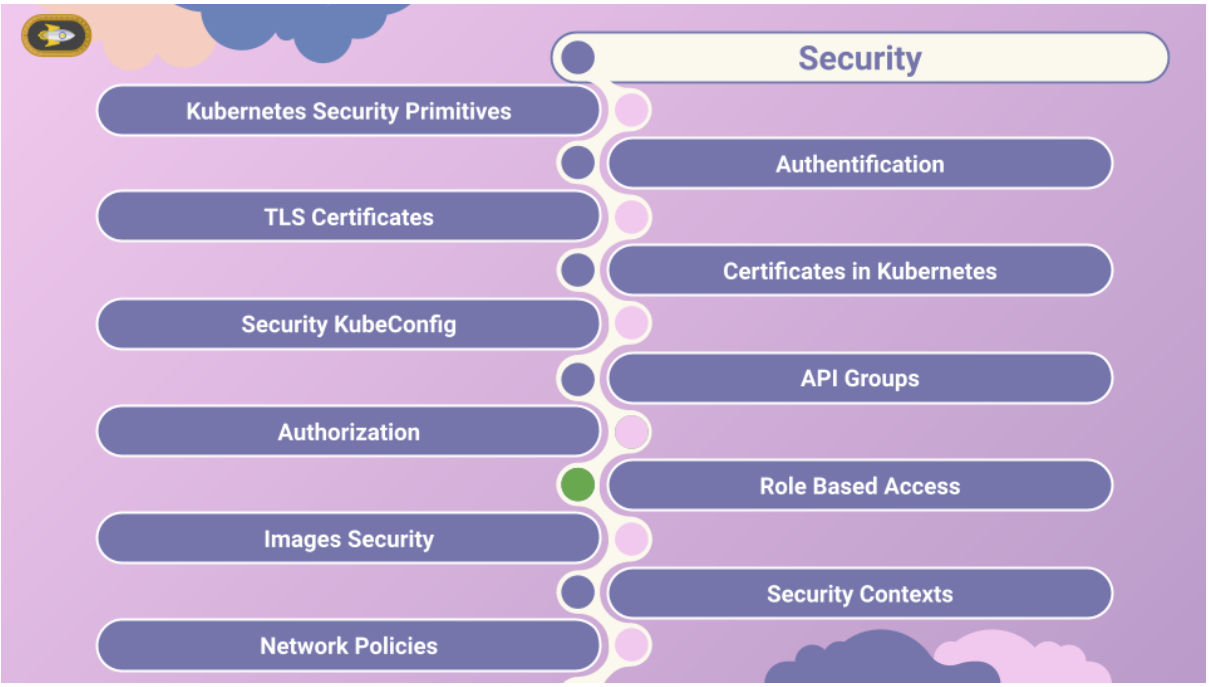
Авторизатор узла обрабатывает только запросы от нод, поэтому он не даст авторизации пользователю. Всякий раз, когда предыдущий режим авторизации отклоняет запрос, он направляется следующему в цепочке.

Т.е. далее авторизатор RBAC выполняет свои проверки и разрешения.

Авторизация прав пользователя завершена, и пользователю предоставляется доступ к запрошенному объекту.

Таким образом, каждый раз, когда один из режимов авторизации отклоняет запрос, он переходит к следующему. Как только запрос одобрен, больше никаких проверок не производится, и пользователю предоставляется разрешение.

Ну вот и все в этой лекции. В следующих лекциях мы подробнее поговорим об управлении доступом на основе ролей.



Привет и добро пожаловать на лекцию. В этой лекции мы более подробно рассмотрим управление доступом на основе ролей.

Обычно, RBAC не сразу приживается в головах новичков, это нормально. С практикой, все встает на места.

Итак, как нам определить роль с правами как показано слева? По философии Kubernetes нам нужно создать объект роли.

Мы создаем файл определения роли, в нем:

- apiVersion rbac.authorization.K8s.io/v1
- kind - Role

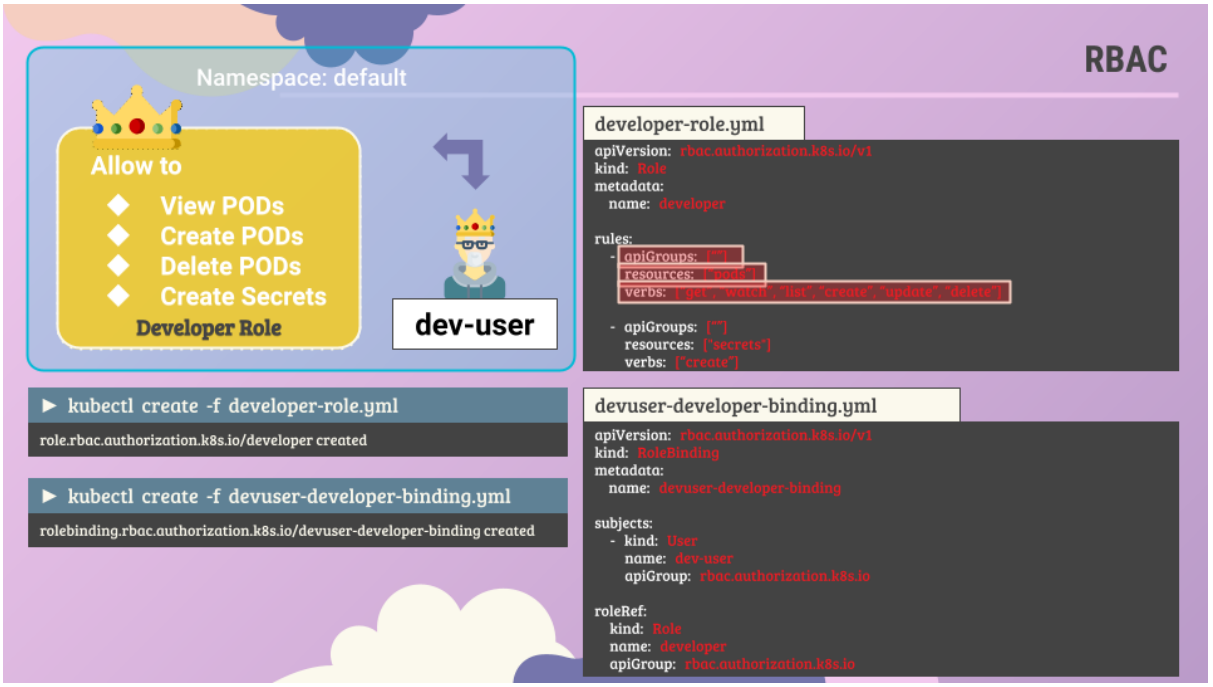
Т.к. мы делаем роль для разработчика, указываем в metadata -> name -> developer

- Далее укажем правила - rules.

Каждое правило состоит из трех разделов: apiGroups, resources и verbs.

Да, это то, о чем мы говорили на одной из предыдущих лекций про API groups.

Мы можем оставить раздел apiGroups пустым, если речь идет о core group. Для всех остальных нужно указать название группы.
В ресурсы запишем, к какому типу мы даем разрешения. Мы хотим предоставить разработчикам доступ к нашим PODs.



Они могут выполнять следующие действия, определяемые verbs: “get”, “watch”, “list”, “create”, “update”, “delete”.
Чтобы позволить разработчикам создавать secrets, мы добавляем еще одно правило для их создания.

Как видишь, мы можем добавить несколько правил для одной роли.

Создадим роль с помощью команды kubectl create, указав файл определения.

Следующий шагом будет привязка пользователя к этой роли.

Для этого мы создаем еще один объект под названием RoleBinding. Объект с типом RoleBinding связывает объект пользователя с ролью.

Пользователь у нас `dev-user`, назовем этот объект `devuser-developer-binding`

apiVersion rbac.authorization.K8s.io/v1

Далее у нас два раздела: subjects и roleRef.

В subjects мы указываем данные пользователя.
В разделе roleRef мы предоставляем подробную информацию о созданной нами роли.

Создадим rolebinding с помощью команды kubectl create.

Также обратите внимание, что роли и привязки ролей подпадают под область действия namespaces.

Таким образом dev-user получает доступ к PODs и Secrets в дефолтном пространстве имен.

Если тебе требуется настроить доступ dev-user в другом пространстве имен, укажи namespace в metadata файла определения.

Для просмотра созданных ролей запусти команду kubectl get roles.

Чтобы получить список привязок ролей, выполни kubectl get rolebindings.

Чтобы узнать подробности о role, воспользуйся командой kubectl describe role указав имя роли developer.
Как видишь, здесь детальные сведения о ресурсах и разрешениях для каждого ресурса. Аналогично для просмотра сведений о rolebindings запусти команду kubectl describe rolebindings.
Здесь мы можем увидеть подробную информацию о существующей привязке ролей.

Что, если ты, как пользователь, хочешь узнать, есть ли у тебя доступ к определенному ресурсу в кластере?

Ты можешь использовать команду kubectl auth can-i и проверить, разрешено ли тебе, скажем, создать deployments. Или удалять узлы.

Если ты являешься администратором, то можно даже выдать себя за другого пользователя, чтобы проверить его разрешение.

Предположим, тебе поручено создать необходимый набор разрешений для пользователя на определенный набор операций.

Ты это сделал, но хочешь проверить, работает ли это так, как задумано.

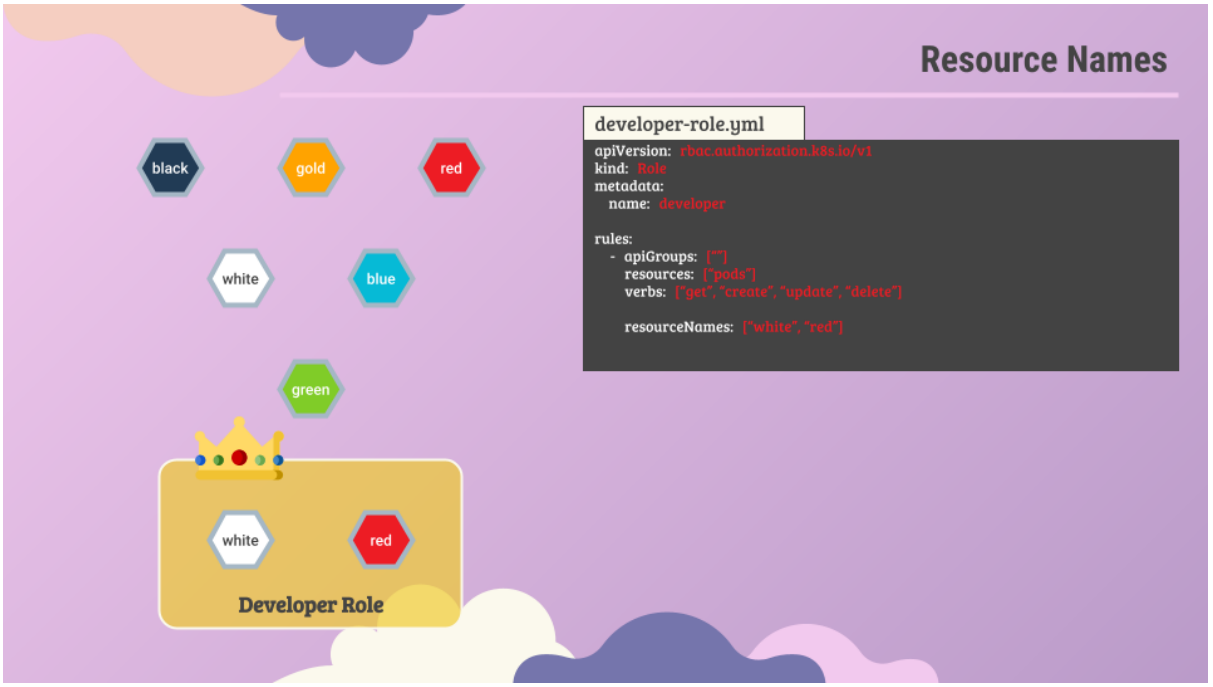
Тебе не нужно проходить аутентификацию как целевой пользователь, чтобы проверить это.

Вместо этого можно запустить ранее использованную команду с опцией `--as` и указать пользователя. Подобно тому, как я сделал здесь.

Поскольку мы не предоставляли разработчикам разрешения на создание deployments, она возвращает no.

Однако dev-user имеет доступ к созданию PODs.

Ты также можешь указать пространство имен в команде следующим образом. Dev-user не имеет разрешения на создание PODs в пространстве имен prod.



Небольшое примечание об именах ресурсов.

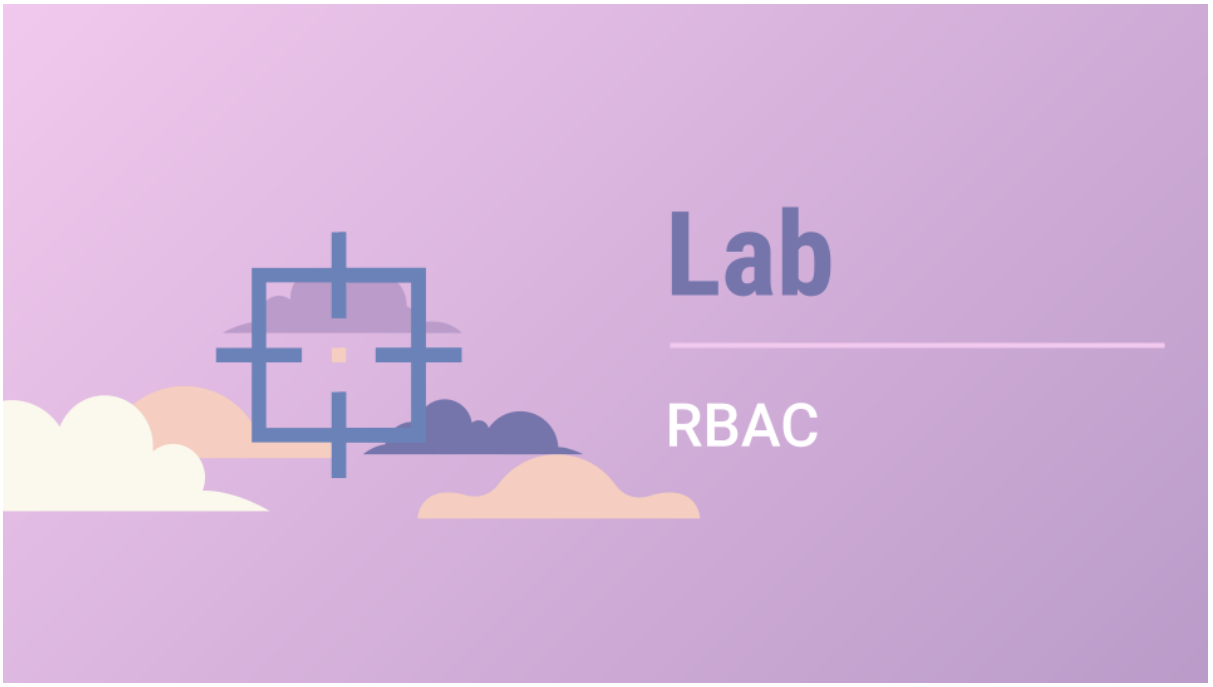
Мы только что увидели, как можем предоставить пользователям доступ к таким ресурсам, как PODs в пространстве имен.

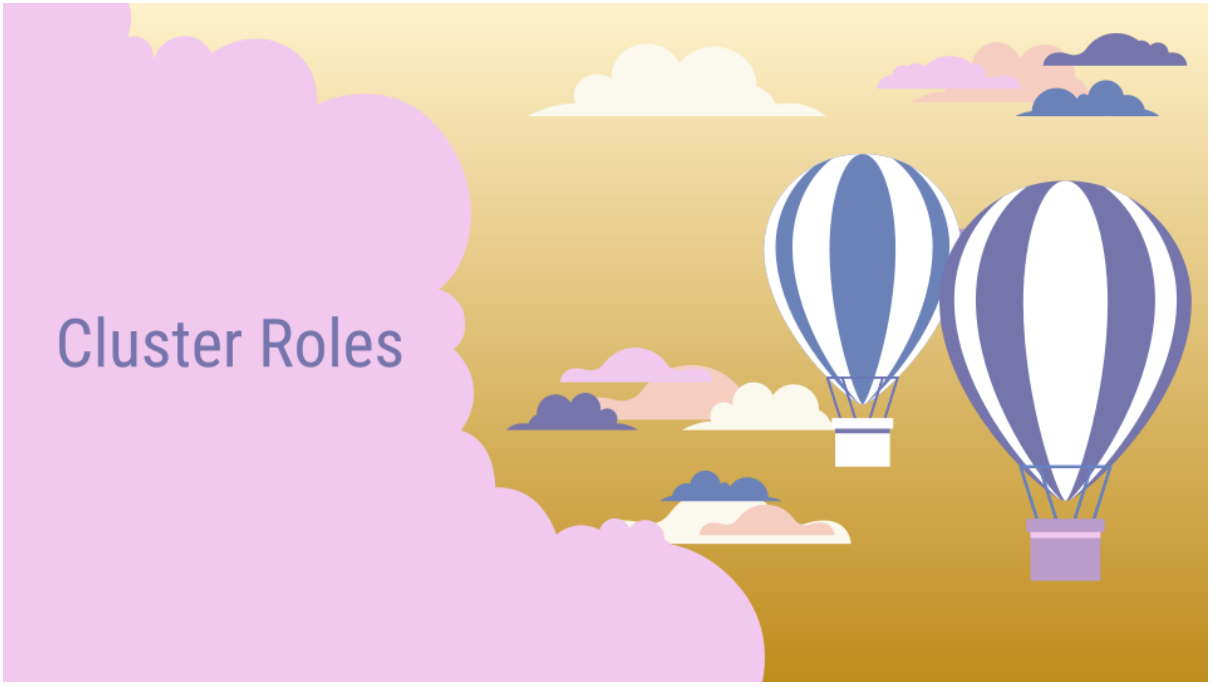
Но мы можем настроить это более гранулярно, опустившись с уровня namesapce на уровень имен ресурсов.

Например, у нас есть шесть PODs в пространстве имен. Мы хотим предоставить пользователю доступ к PODs, но не ко всем PODs.

Мы можем дать доступ только к PODs с именем `white` и `red`, добавив в правило поле resourceNames.

Ну вот и все в этой лекции. Перейди в раздел практики и потренируй ролевой контроль доступа.


Lab
RBAC



Привет и добро пожаловать на лекцию.

В предыдущей лекции мы обсуждали roles и role bindings. В этой лекции мы поговорим о кластерных ролях и их привязках.

Когда мы говорили roles и role bindings, я упомянул, что роли и привязки ролей имеют параметр namespace, что означает, что они создаются в конкретных пространствах имен.

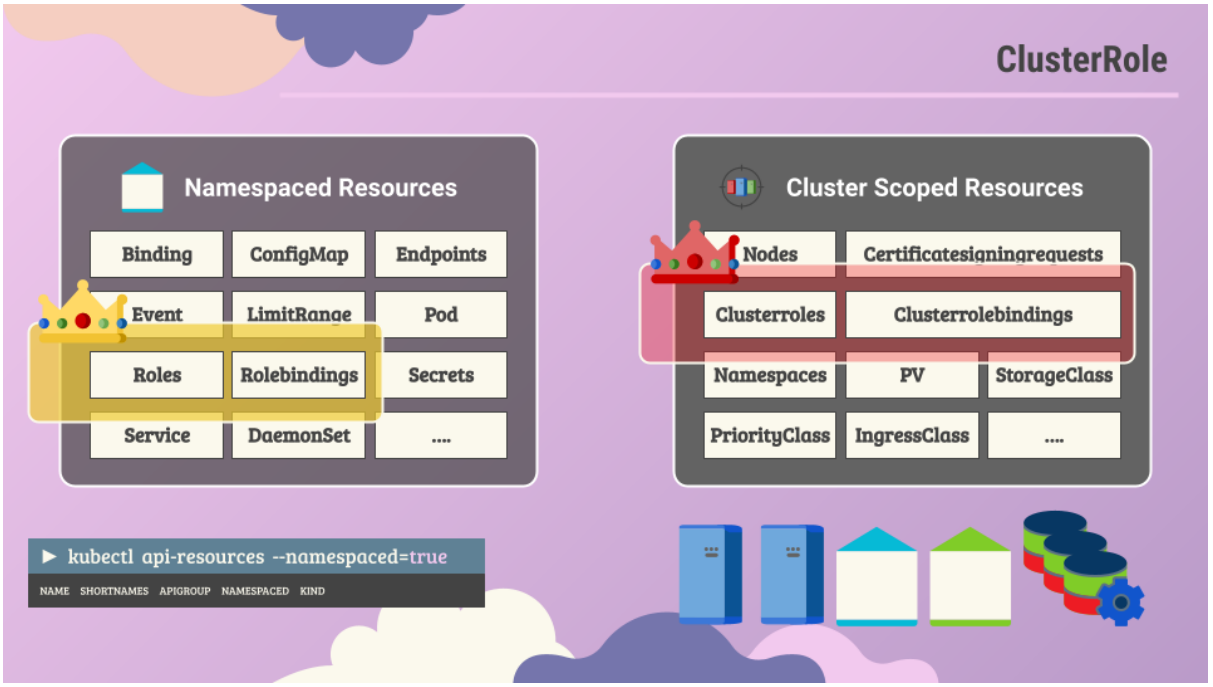
Если мы не укажем этот namespace, они будут созданы в пространстве имен default и будут управлять доступом только в нем. В одной из предыдущих лекций, мы обсуждали пространства имен и то, как они помогают в группировке или изоляции ресурсов, таких как PODs, deployments и services.

Но как насчет других ресурсов, например нод? Можно ли сгруппировать или изолировать узлы в namespace?

Например, можем ли мы заявить, что node01 отныне является частью пространства имен dev.

Нет, эти ресурсы находятся на уровне или в масштабе кластера (cluster wide или cluster scope resources).
Они не могут быть связаны с каким-то определенным пространством имен. Таким образом, ресурсы делятся на категории либо namespaced, либо cluster scoped.

На протяжении этого курса мы видели много Namespaced Resources. Такие как PODs, replicasets, jobs, deployments, services, secrets.
Также сюда относятся и те, с которыми мы познакомились в прошлой лекции - roles и role bindings.



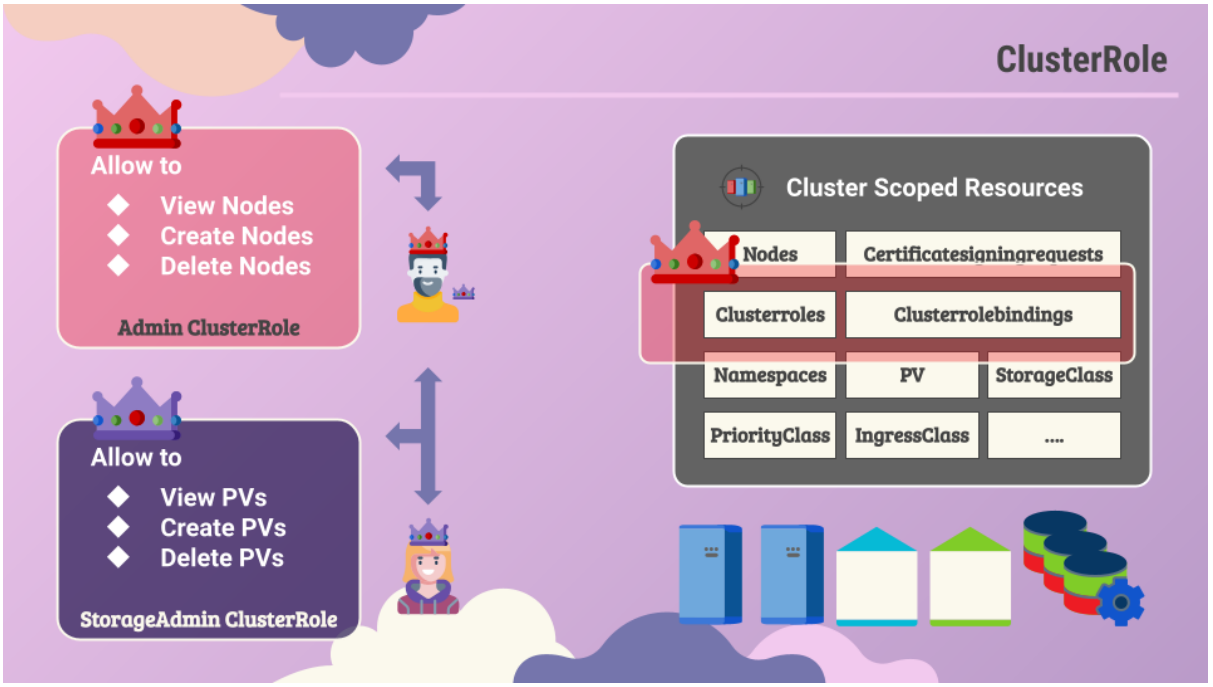
Эти ресурсы создаются в пространстве имен, которое мы указываем при их создании. Если не укажем пространство имен, они будут созданы в default namespace.

И для их просмотра, удаления или обновления нам всегда нужно указать правильное пространство имен.

Cluster scoped ресурсы - это те ресурсы, для которых мы не указываем namespace при создании. Это ресурсы вроде nodes, persistent volumes, clusterroles, clusterrole bindings, certificate signing requests.

Также и сами объекты пространств имен не могут быть namespaced.

Обрати внимание, что это не полный список ресурсов, чтобы увидеть полный список namespaced-ресурсов и ресурсов, не относящихся к пространствам имен, запусти команду kubectl api-resources с установленной опцией namespaced.



В предыдущей лекции мы увидели, как авторизовать пользователя для доступа к namespaced-ресурсам. Для этого мы использовали роли и привязки ролей.

Но как нам разрешить пользователям использовать ресурсы всего кластера, такие как узлы или постоянные тома?

Это то место, где стоит применить clusterroles и clusterrolebindings.
Кластерные роли аналогичны простым ролям, за исключением того, что они предназначены для cluster scoped ресурсов.

Например, роль Admin может быть создана, чтобы предоставить администратору кластера разрешения на просмотр, создание или удаление узлов в кластере.

Точно так же, роль StorageAdmin может быть создана для авторизации администратора хранилища и авторизировать пользователя для оперирования постоянными томами.

Для этого создадим файл определения с kind: ClusterRole, укажем правила, как мы уже делали это для roles.

Resources будут nodes и добавим нужные полномочия.
Затем создадим clusterrole, применив манифест.

Следующим шагом является привязка пользователя к этой кластерной роли.
Для этого мы создаем еще один объект ClusterRoleBinding, он связывает пользователя с кластерной ролью, аналогично как RoleBinding делает с обычной ролью.

Мы назовем ее cluster-admin-binding. Kind: ClusterRoleBinding. Под subjects мы указываем сведения о пользователе, это cluster-admin-user, и кластерную роль cluster-admin в разделе roleRef.

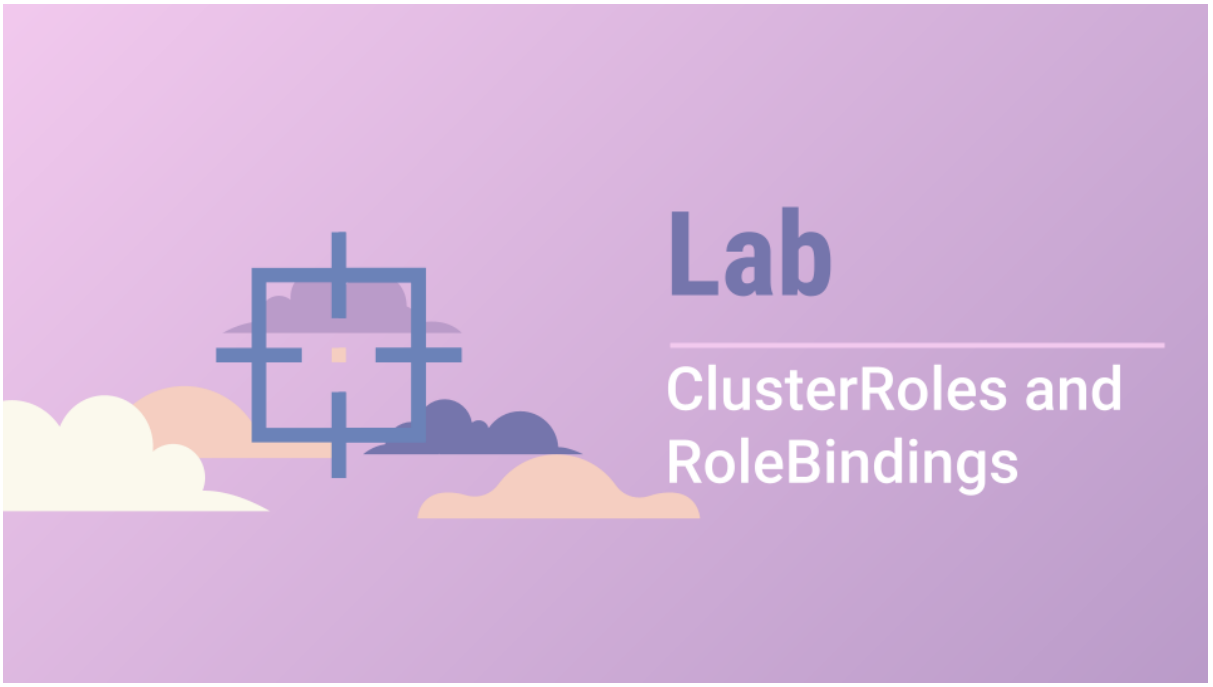
Создаем ClusterRoleBinding с помощью команды kubectl create.

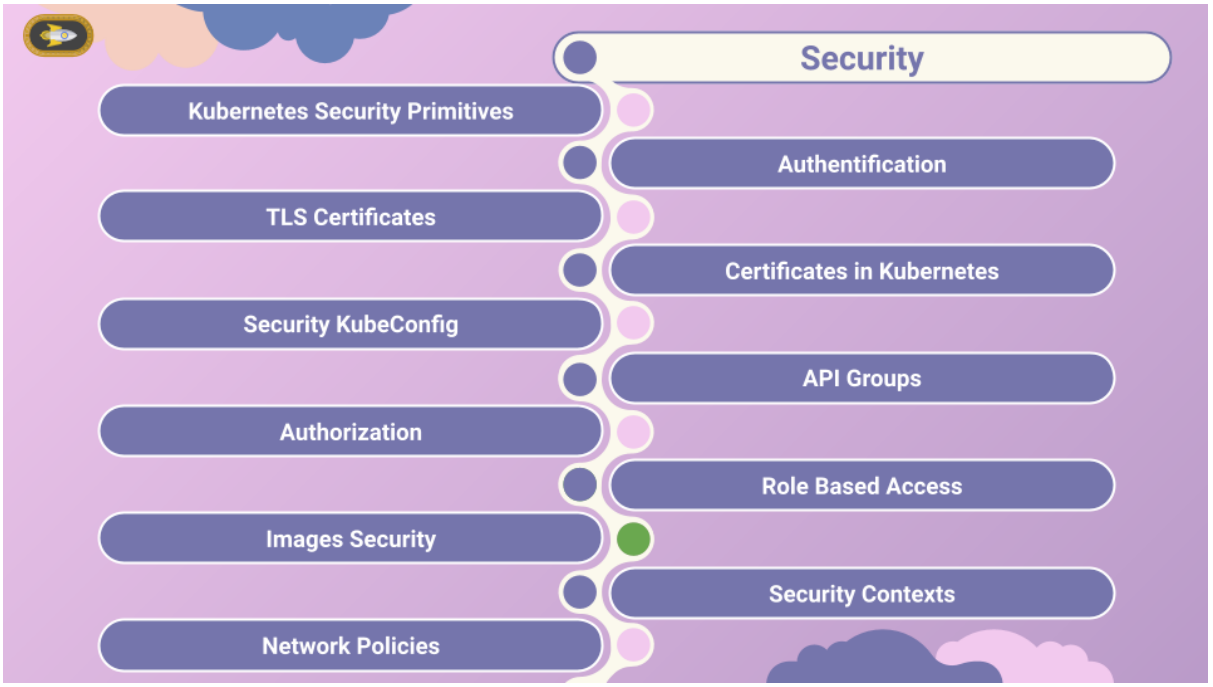
Один момент, прежде чем закончим.
Мы сказали, кластерные роли и их привязки используются для ресурсов уровня кластера. Но это не жесткое правило, ты можешь создать ClusterRole и для namespaced-ресурсов.

Если ты это сделаешь, пользователь получит доступ к этим ресурсам во всех пространствах имен.
Ранее, когда мы создавали роль для авторизации пользователя для доступа к PODs, у пользователя был доступ к PODs в определенном namespace.
Использовав кластерную роль мы разрешаем пользователю доступ ко всем PODs в кластере, в том числе и к системным. Это мощная штука, имей это ввиду.

При первой настройке кластера в Kubernetes по умолчанию создано несколько ClusterRoles.

Ты узнаешь о них в предстоящем практическом тесте. Переходи к нему, а я жду тебя в следующей лекции.





Привет и добро пожаловать. В этой лекции мы поговорим о защите образов.

Мы начнем с основ именования образов, а затем поговорим о безопасных реджистри и о том, как настроить свои PODs для использования образов из безопасных репозиториев.

На протяжении курса мы разворачивали несколько разных типов PODs, в которых размещались различные типы приложений, такие как веб-приложения, базы данных, и кэш Redis и т. д.

Начнем с простого файла определения POD. Здесь мы использовали образ nginx для развертывания контейнера nginx. Давай подробнее рассмотрим этот образ.

Имя - nginx. Но что это за образ и откуда он взят? Его имя соответствует соглашению об именах образов Docker.

NGINX - название образа или репозитория.

Когда мы говорим nginx, на самом деле это nginx / nginx.

Первая часть обозначает имя пользователя или учетной записи.

Например, если ты сам создал учетную запись в DockerHub, тогда учетная запись пользователя, которую ты выбрал, является первой частью.

Если мы не укажем здесь имя учетной записи, предполагается, что оно совпадает с именем репозитория.

В данном случае это nginx.
Если ты создашь свою собственную учетную запись и разместишь свои собственные репозитории или образы, тогда, чтобы до них добраться, используй аналогичный шаблон.

Теперь, где эти образы хранятся и откуда скачиваются? Поскольку мы явно не указали место расположения образов, предполагается, что они находится в Docker-реджистри по умолчанию - это dockerhub.

Его доменное имя - docker.io.

Все образы хранятся в реестре. Каждый раз, когда мы создаем новый образ или обновляем существующий, мы пушим его в этот реджистри. И каждый раз, когда кто-либо развертывает это приложение, оно извлекается из этого реджистри.

Есть также много других популярных реестров.
Реестр от Google находится по адресу gcr.io, где хранится множество образов, связанных с Kubernetes. Это все общедоступные образы, которые каждый может загрузить и получить к ним доступ.

Если у тебя есть собственные приложения, которые не должны быть общедоступными, развертывание внутреннего частного реджистри может быть хорошим решением.

Многие поставщики облачных услуг, такие как AWS, Azure или GCP, по умолчанию предоставляют частный реестр для облачной учетной записи.

В любом из этих решений, будь то в Dockerhub, в реестре Google или в твоем собственном реджистри, мы можем сделать репозиторий частным, чтобы к нему можно было получить доступ с помощью набора учетных данных.

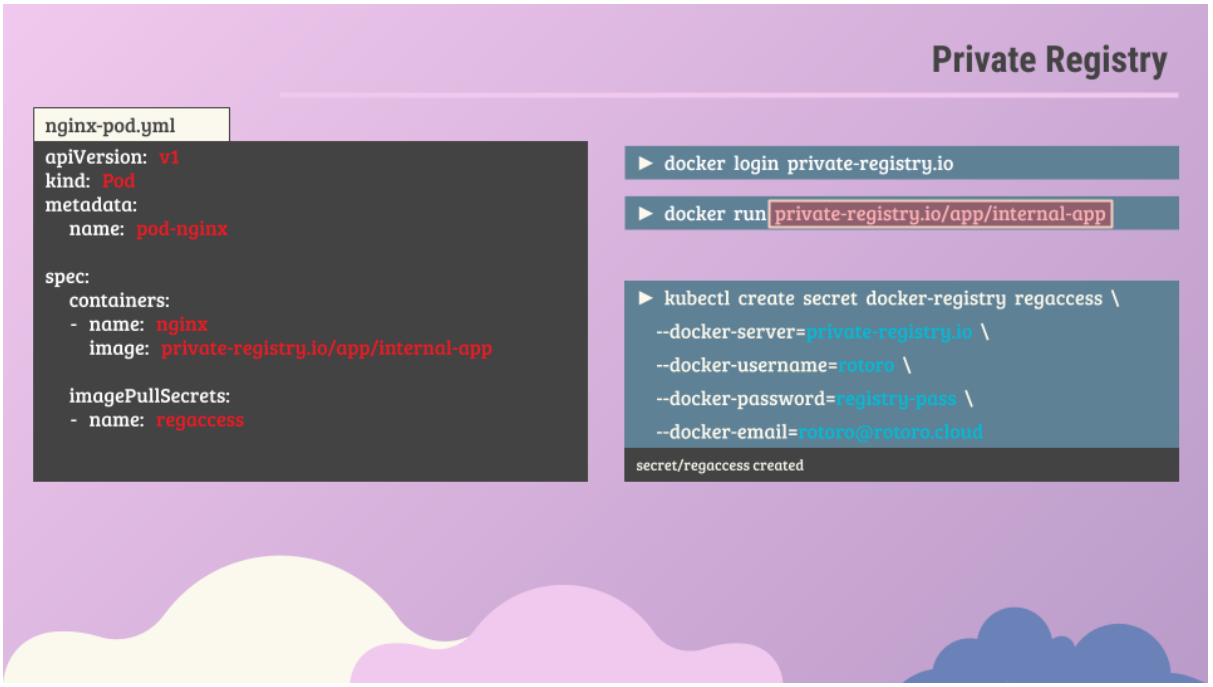
С точки зрения Docker, чтобы запустить контейнер с использованием приватного образа, сначала пользователь должен войти в свой частный реестр.

Используя команду docker login, введи свои учетные данные.

Теперь твои данные для входа сохранены на жестком диске и они будут использованы при запуске образа из private registry.

Вернемся к нашему файлу определения POD. Как использовать вместо общедоступного наш приватный реджистри?

Мы заменяем имя образа полным путем на тот, который указывает на наш частный реджистри.



Но как реализовать аутентификацию в части docker login?
И как Kubernetes получит учетные данные для доступа к этому частному реестру?

Как мы знаем, в Kubernetes образы извлекаются и запускаются средой выполнения контейнеров - container runtime. Поэтому нужно передать учетные данные или Docker или другому рантайму, расположенному на узле.

Сначала мы создаем объект secret с учетными данными в нем. Тип secret будет docker-registry, а название regaccess. Создадим его этой командой.

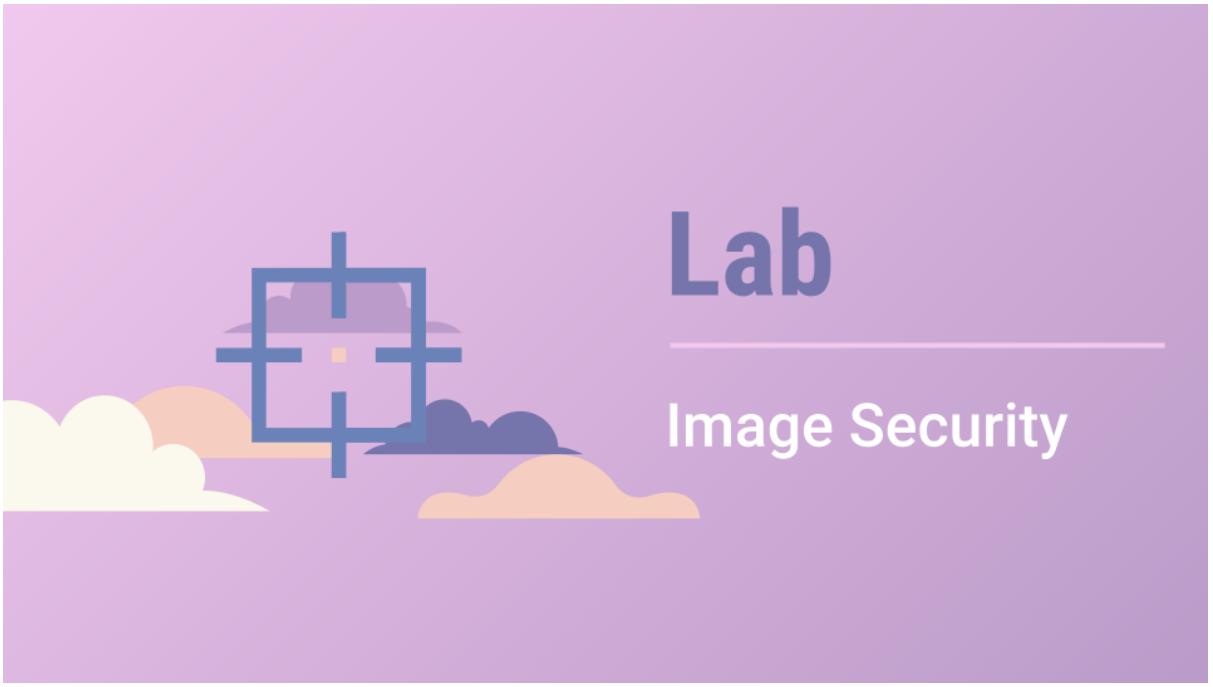
`docker-registry` - это встроенный тип secret, созданный для хранения учетных данных Docker.

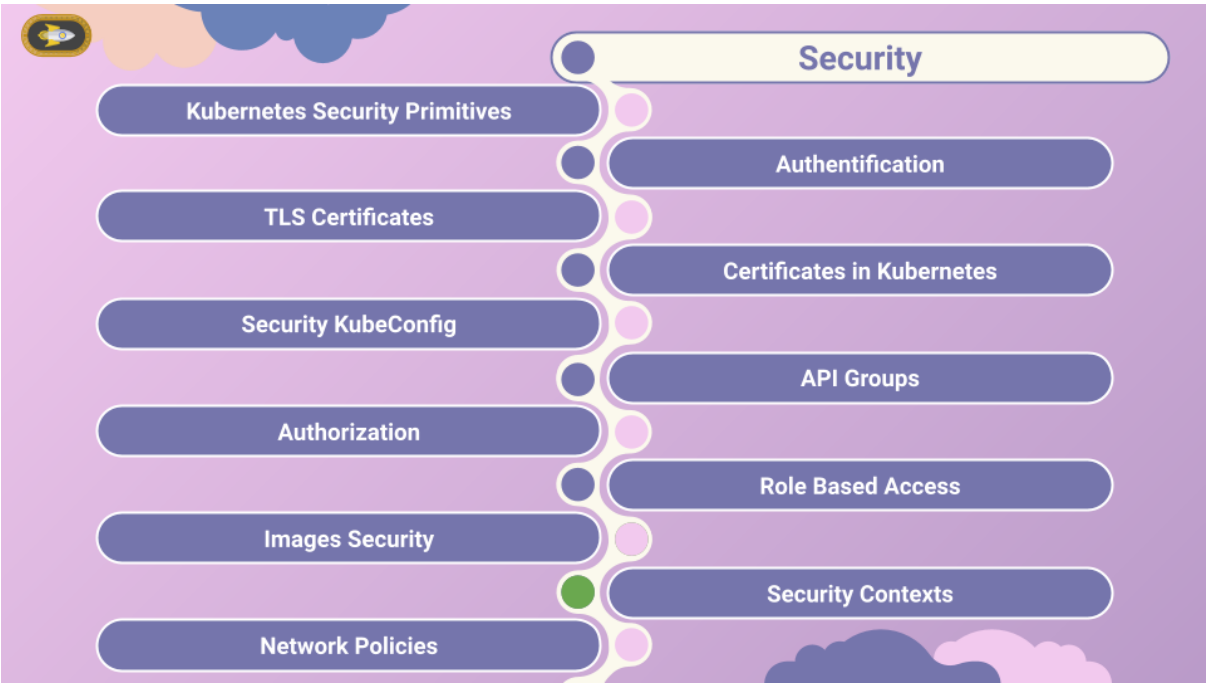
Затем мы указываем имя сервера реджистри, имя пользователя для доступа, пароль и адрес электронной почты пользователя.

После того, как secret создан, мы указываем ссылку на него внутри нашего файла определения в разделе imagePullSecrets.

Когда Kubernetes создаст POD, kubelet на узле, куда POD назначится использует учетные данные из secret для пуллинга образов из приватного реджистри.

Вот и все, в этой лекции. Перейди в раздел упражнений и попробуй себя в работе с образами.


Lab
Image Security



Привет и добро пожаловать на эту лекцию о Security Contexts в Kubernetes.

Когда мы запускаем докер-контейнер, у нас есть возможность назначить некоторый набор опций безопасности, например таких, как идентификатор пользователя, с который будет исполняться контейнер или можно добавить или удалить linux capabilities для контейнера.

Это также можно настроить и в Kubernetes. Как мы уже давно знаем, в Kubernetes контейнеры инкапсулируются в POD. Мы можем настроить параметры безопасности на уровне контейнера или на уровне POD.

Если мы настраиваем их на уровне POD, настройки будут перенесены на все контейнеры внутри POD.

Также можно представить их как в POD, так и в контейнер. В таком случае настройки в контейнере переопределят настройки в POD.

Начнем писать файла определения POD.

Этот POD запускает образ alpine и держит его живым с помощью команды sleep 500.

Чтобы настроить контекст безопасности в POD, в раздел спецификации нужно вставить поле с названием `SecurityContext`, а под ним установим идентификатор пользователя в параметре `runAsUser`.

В случае контейнера, чтобы установить ту же конфигурацию на его уровне, перемести весь раздел в массив, описывающий требуемый контейнер, как ты видишь здесь.

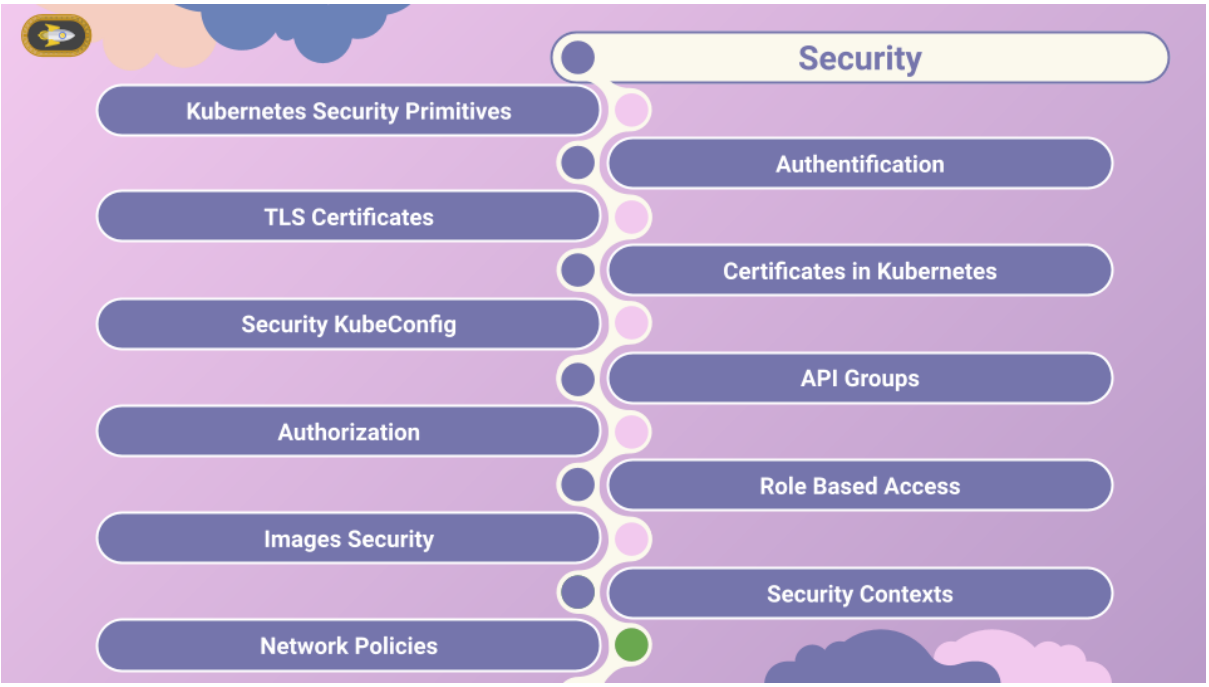
Для задания capabilities, используй параметр под полем SecurityContext и укажи список этих возможностей для добавления в контейнер.

Еще момент - capabilities можно задать только на уровне контейнеров.

Ну вот и все, что касается контекста безопасности в Kubernetes.

Перейди в раздел упражнений и потренируйся просматривать настройки и устранять неполадки, связанные с контекстами безопасности в Kubernetes.

Увидимся на следующей лекции.



Привет и добро пожаловать на эту лекцию о сетевых политиках - network policies.

Давай сначала разберемся с основами сети и безопасности.

Мне очень жаль, если для тебя это слишком просто, но у каждого студента свой уровень, и я хочу убедиться, что мы все на одной волне. Мы просто уделим этому пару минут.

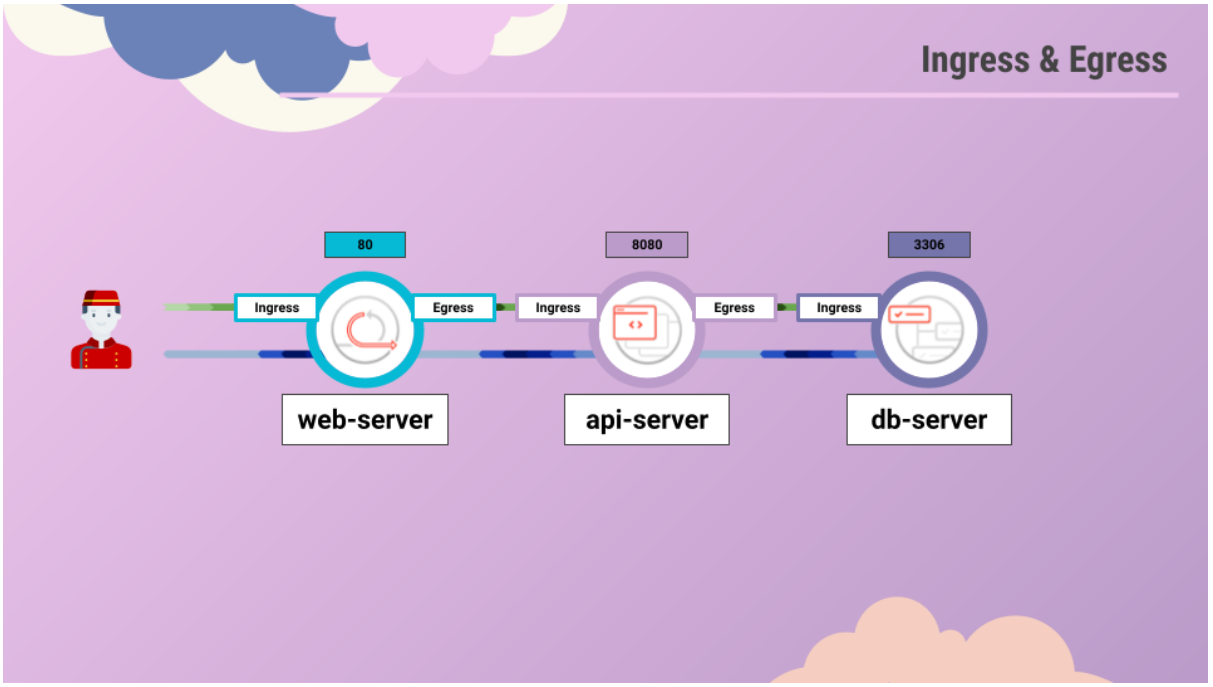
Начнем с простого примера трафика, проходящего через веб-приложение и сервер базы данных.

У нас здесь веб-сервер, обслуживающий интерфейс для пользователей, сервер приложений, обслуживающий внутренний API, и сервер базы данных.
Пользователь отправил запрос на веб-сервер через порт 80, веб-сервер переслал его на сервер API через порт на порт 8080 в backend.

Сервер API обратился к базе данных через порт на 3306, получил данные из нее, а затем сформировал ответ и отправил его обратно пользователю.

Очень простой сетап. Здесь у нас два типа трафика: входящий (ingress) и исходящий (egress).

Например, для веб-сервера входящий трафик от пользователей - это ingress трафик, а исходящий запрос к серверу приложений - это egress трафик, я подписал его табличками.



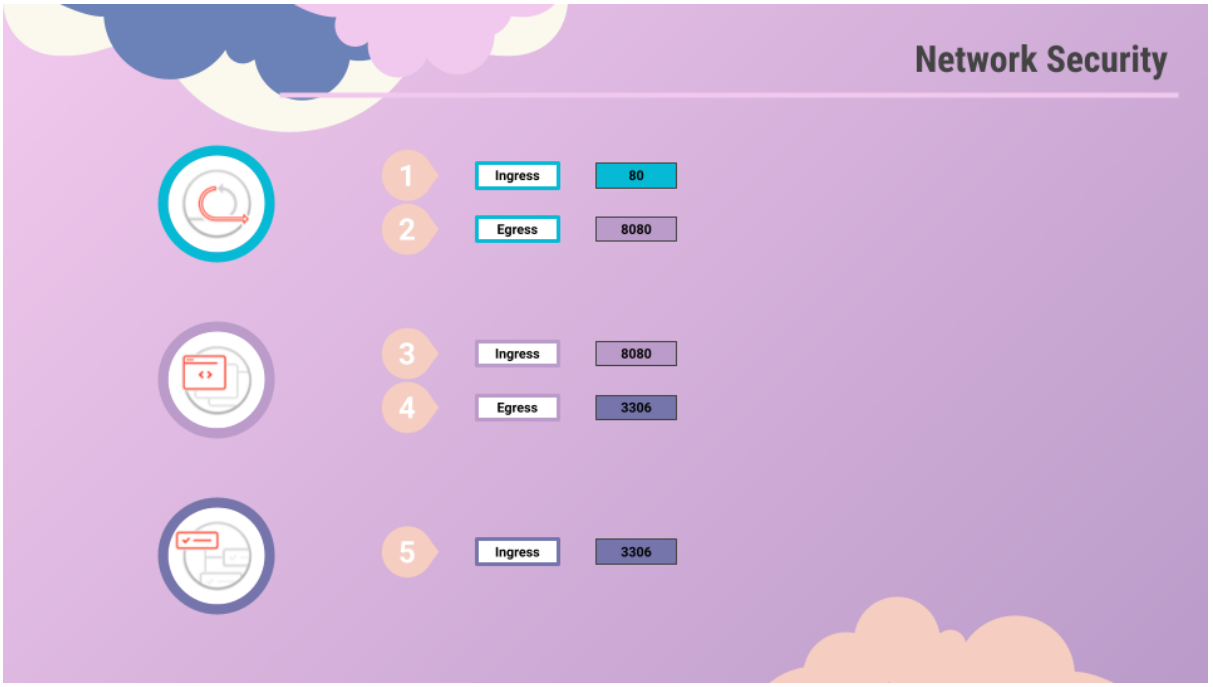
Когда мы определяем ingress или egress, помни, что мы отталкиваемся от того места, в котором возник трафик.
Ответ пользователю, обозначенный синей линией, на самом деле не имеет значения, поскольку он появился позже.

Точно так же в случае внутреннего сервера API. Он получает входящий трафик от веб-сервера на порт 8080 и имеет исходящий трафик на порт 3306 сервера базы данных.
С точки зрения сервера базы данных, он получает входящий трафик на 3306 от сервера API.

Если бы мы попробуем скомпоновать все эти таблички, то получим список правил, необходимых для того, чтобы наша система работала.

У нас появилось правило ingress, которое разрешило прием HTTP-трафика на порт 80 веб-сервером. Далее egress-правило, разрешающее трафик с веб-сервера на порт 8080 сервера API.

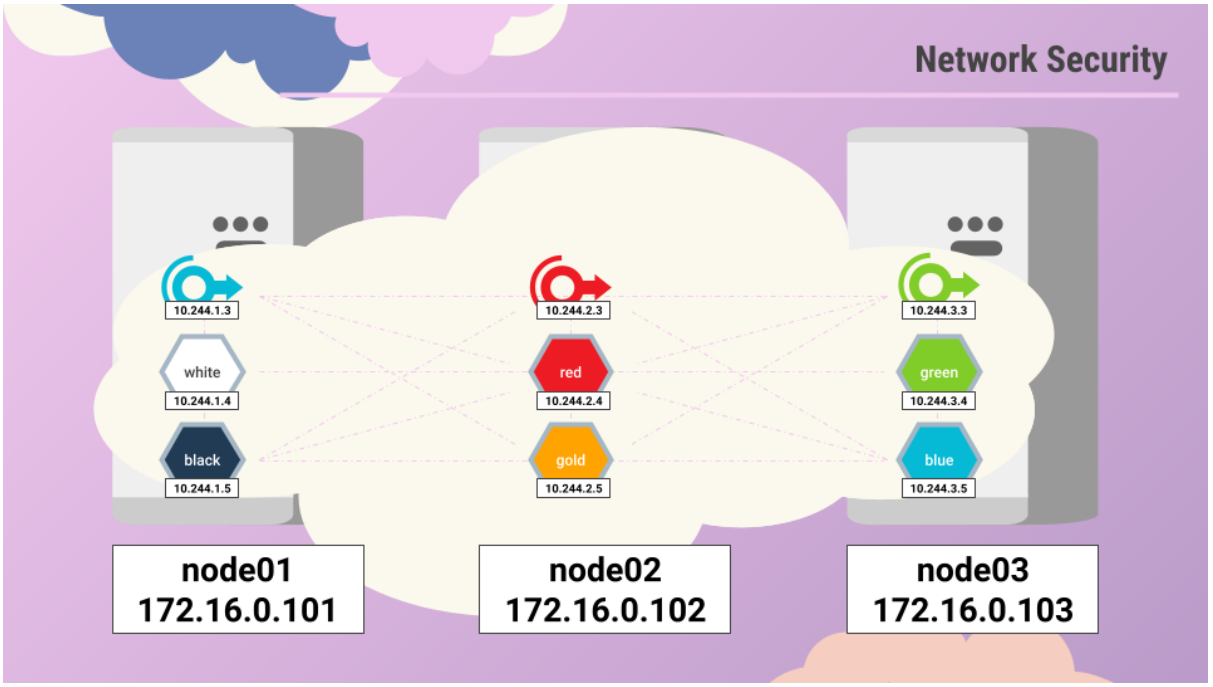
Еще ingress-правило для приема на порт 8080 уже на сервере API и правило egress, разрешающее отправлять трафик на порт 3306 на сервер базы данных.
Наконец, правило для ingress для сервер базы данных, чтобы он принимал трафик на порт 3306 с API-сервера.



Вот это были основы потоков трафика и их правил. Теперь давай посмотрим на сетевую безопасность в Kubernetes.

Итак, у нас есть кластер с набором узлов, на которых размещается некоторое количество PODs и services. У каждого узла есть IP-адрес, как и у каждого POD или службы.

Одним из предварительных условий для создания сети в Kubernetes является то, что какое бы решение по организации сети не было выбрано, PODs должны иметь возможность взаимодействовать друг с другом без необходимости настраивать какие-либо дополнительные параметры, такие как маршруты, туннели и т.д.

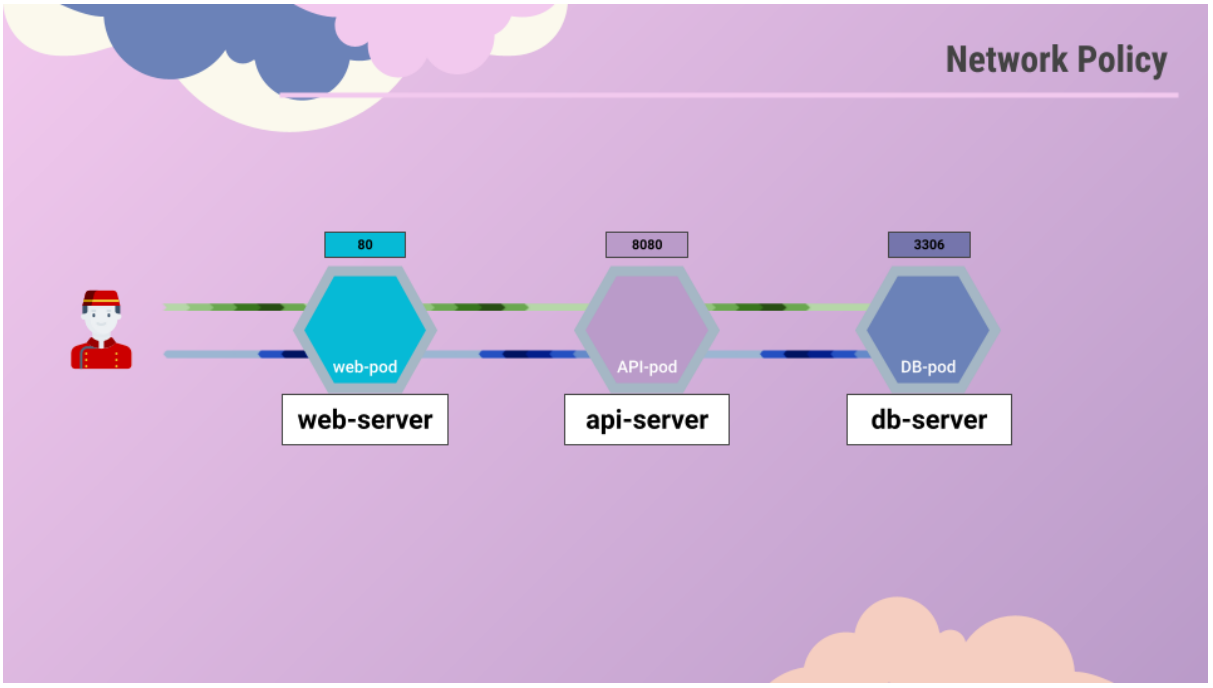


Например, в этом сетевом решении все PODs находятся в виртуальной частной сети, которая реализована на всех узлах кластера Kubernetes. Она как бы развернута над локальной сетью нод.

И все внутренние сущности по умолчанию могут связываться друг с другом, используя IP-адреса и имена PODs или services, настроенные для этой цели.

Kubernetes по умолчанию настроен с правилом `Все разрешено` - All Allow, которое разрешает трафик из любого POD в любой другой POD или service.

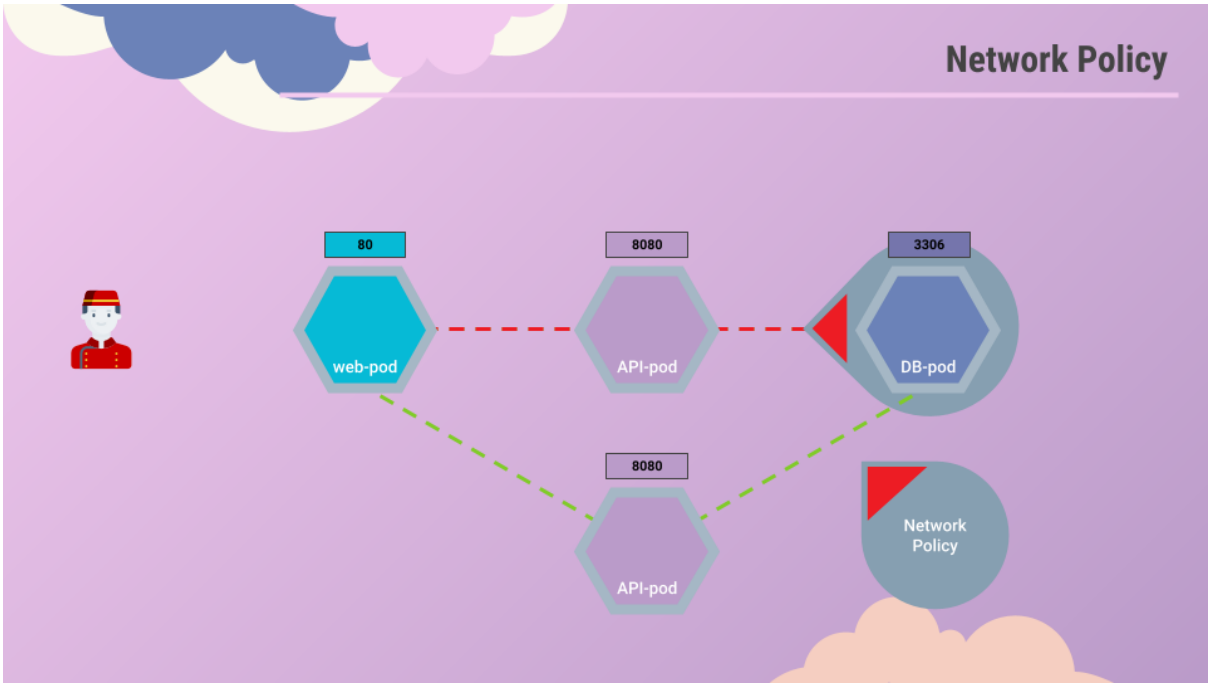
Давай теперь вернемся к нашему предыдущему обсуждению и посмотрим, как оно соотносится с Kubernetes.



Для каждого компонента в приложении мы развертываем POD. Один для frontend-сервера, еще для сервера API и один для базы данных.
Мы создаем службы для обеспечения связи между POD, а также с конечным пользователем.

Основываясь на том, что мы обсуждали немного раньше, по умолчанию все три PODs могут взаимодействовать друг с другом в кластере.

Что делать, если мы не хотим, чтобы веб-сервер мог напрямую связываться с сервером базы данных? Скажем, наша команда безопасников поставила нам такое условие.

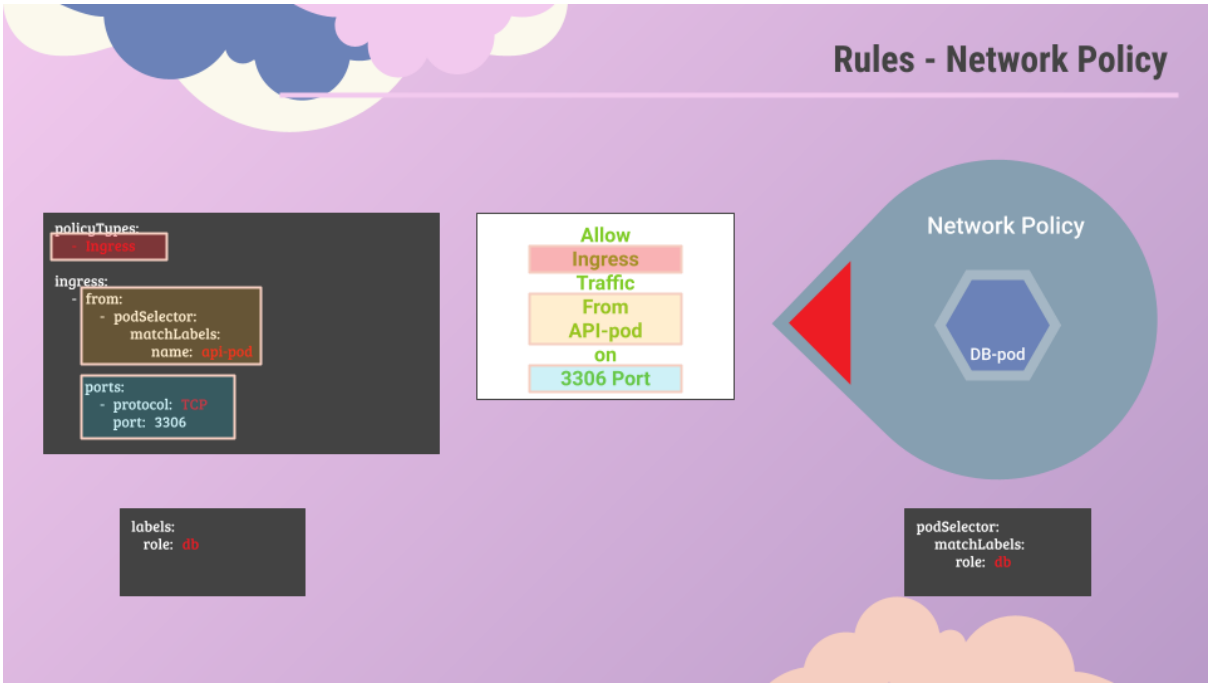


Вот где отлично себя проявит механизм Kubernetes, известный как network policies. Мы должны реализовать сетевую политику, чтобы разрешить трафик на сервер db только с сервера api.

NetworkPolicy - это еще один объект Kubernetes, относящийся к пространству имен, так же, как и PODs, ReplicaSets или Services.

Мы связываем сетевую политику с одним или несколькими PODs.

Определим правила в сетевой политике. В этом случае я бы сказал, разрешить только входящий трафик из api-pod на порт 3306.



После создания этой policy, она начнет блокировать весь другой трафик к POD, а пропускать только тот, который будет соответствовать указанному правилу.

Еще раз, это повлияет только на POD, к которому применяется эта network policiy.

Ок, а как применить или связать сетевую политику с POD?

Мы используем тот же метод, который использовался ранее для связывания ReplicaSets или Services с PODs. Labels и Selectors.

Мы маркируем POD и используем те же метки в поле matchLabels в сетевой политике. Затем указываем тип правила в параметре policyTypes, который укажет, как применять правила - для разрешения входящего или исходящего трафика. Ну или и то, и другое вместе.

В нашем случае мы хотим разрешить только входящий трафик к db-pod.

Итак, мы добавляем поле Ingress.
Теперь определим, откуда можно. Мы решили, что только из POD API-сервера. Укажем в поле from в podSelector название нужного POD, а именно api-pod. Как видишь, здесь мы снова используем метки и селекторы.

И, наконец, порт, на котором разрешен трафик, - 3306.

Давай сложим все это вместе.

Мы начинаем с пустого файла определения объекта и, как обычно, у нас есть apiVersion, kind, metadata и spec.

ApiVersion - networking.k8s.io/v1, kind - NetworkPolicy. Назовем политику db-policy.
Затем в раздел спецификации мы сначала положим podSelector, чтобы применить эту политику к POD с БД.

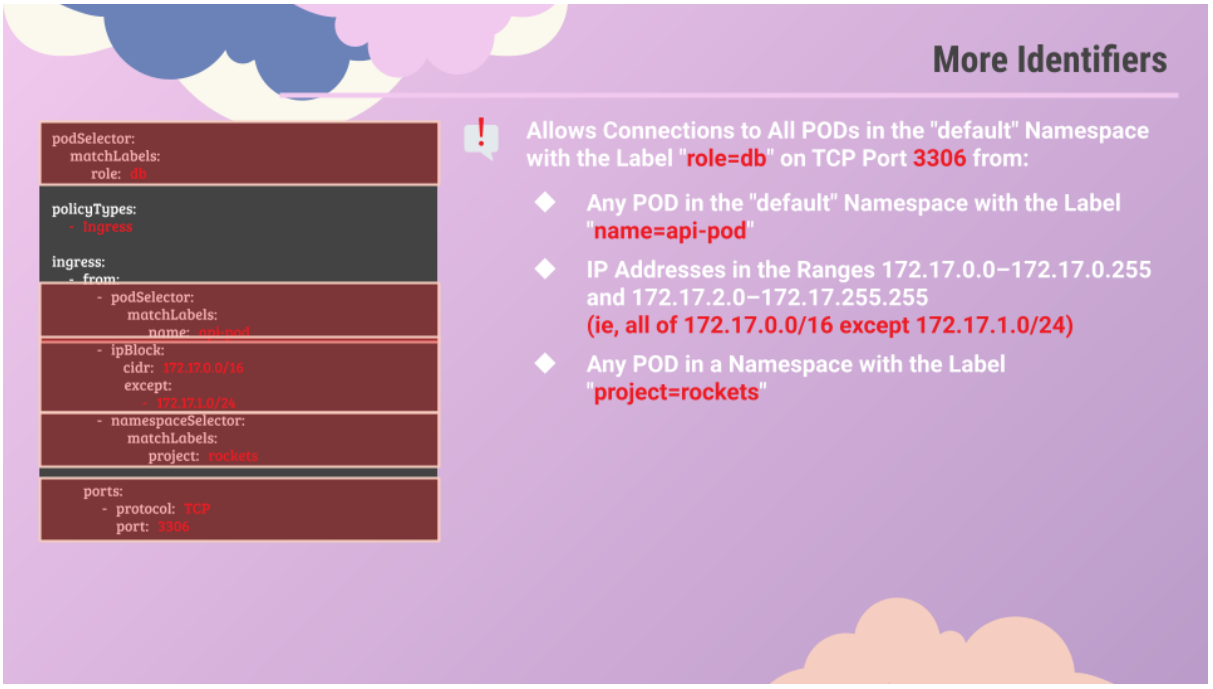
Затем мы переместим правило, которое создали и поместим на тот же уровень, что и podSelector.

Все. У нас готова первая network policy.

Запустим команду kubectl create, чтобы создать политику.

Мы можем также гибко сконфигурировать свои политики, используя дополнительные виды селекторов, такие как ipBlock, который может разрешить поступление трафика от определенных адресов. Также внутри этого правила можно внести ограничения с помощью поля except, которое исключает часть диапазона адресов.

Третий селектор namespaceSelector используется для указания меток namespace.



Здесь я показал пример, как мы последовательно выставили правила, которые дают возможность получать доступ для POD с определенной меткой, для диапазона адресов, за исключением подсети из него, а также для всех PODs из определенного пространства имен. Мы подробнее поговорим об особенностях дизайна в следующей лекции.

Помни, что сетевые политики применяются сетевым плагином, который непосредственно не является частью проекта Kubernetes.
И не все сетевые решения поддерживают сетевые политики.

Некоторые, из тех, что поддерживают: kube-router, Calico, Romana и Weave-net.

Если же мы используем Flannel в качестве сетевого решения, то он не поддерживает сетевые политики и не собирается это внедрять. Всегда обращайся к документации сетевого решения, чтобы узнать о поддержке сетевых политик.

Также помни, что даже в кластере, настроенном с помощью решения, не поддерживающего сетевые политики, мы все равно можем создать network policies, но они просто не будут применяться. А мы не получим никаких сообщений об ошибке, что сетевое решение не что-то не поддерживает.

Еще пара моментов, прежде чем мы закончим.

Network policy пока не дружит с TLS, если у тебя идет шифрование при транспортировке используй service mesh или ingress controller для внедрения своих правил.

Также не используй этот механизм для направления трафика через единый шлюз.

Все эти ограничения описаны в документации, полистай её.

Что ж, на этом лекция закончилась. Почитай документацию, а в следующей лекции мы рассмотрим подходы к дизайну правил. Увидимся!

Привет и добро пожаловать. В этой лекции мы более подробно рассмотрим сетевые политики.

Здесь у нас есть те же части веб, API и база данных, о которых мы говорили в предыдущей лекции.

Итак, сначала давай проясним наши требования.

Наша цель - защитить POD базы данных, чтобы он не разрешал доступ из любых других PODs, кроме API, и чтобы был открыт только порт 3306.

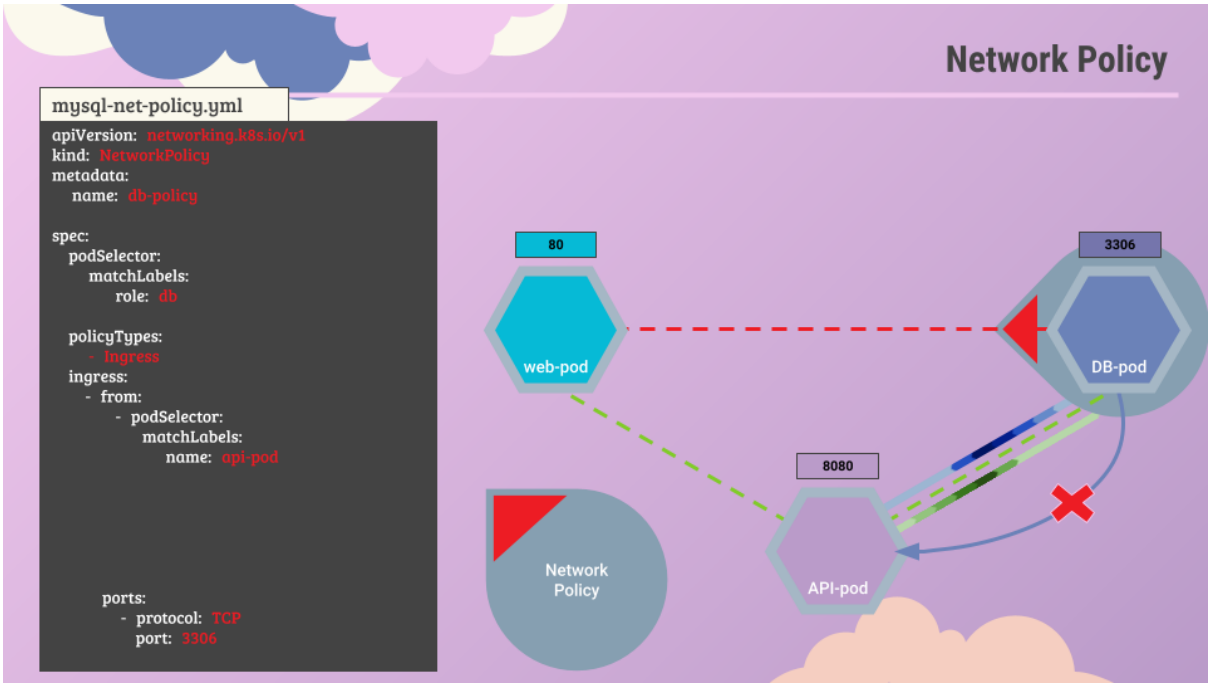
Еще предположим, что нас не беспокоят web-pod и API-pod. Там разрешен любой трафик на вход и на выход из любого места.

Наша задача - защитить эту часть с POD базы данных, разрешив трафик только из API. Так что думаем с перспективы DB-pod, и не думая об остальных.

Таким образом, нам не нужно беспокоиться о поддержке веб-POD, поскольку с него не будет разрешен трафик на базу данных.
Так что давай избавимся от него.

Мы также можем забыть о порте в POD с API, к которому подключается веб-сервер, поскольку нас это тоже не волнует.

Как мы уже обсуждали, Kubernetes по умолчанию разрешает весь трафик из всех PODs во все места назначения, поэтому в качестве первого шага мы хотим заблокировать все, что входит и выходит из POD базы данных.
Итак, мы создаем network policy.



Назовем его db-policy.
И первый шаг - связать эту сетевую политику с POD, который мы хотим защитить. Делаем это с помощью меток и селекторов.

Добавим поле podSelector, а в нем параметр matchLabels. В нем разместим такую же метку, как и на db-pod, а именно role: db.
Это свяжет нашу policy с POD базы данных и заблокирует весь трафик.

Однако нам нужно сохранить возможность получать трафик на порт 3306 с API-pod, и отправлять его обратно.

Вот как мы будем рассуждать дальше:

Во-первых, нам нужно выяснить, какой тип политик должен быть определен в этом объекте NetworkPolicy, чтобы он подходил под наши требования.

Как помнишь, есть два типа политик, и мы обсуждали их в предыдущей лекции.

Есть ingress и egress.

Итак, нужны ли нам здесь правила на вход или на выход? А может и то, и другое?

Напомню, что мы смотрим на это с точки зрения db-pod, и мы хотим разрешить входящий трафик из API-pod.

Что значит входящий? Правильно, это ingress.

POD API выполняет запросы к базе данных, а POD базы данных возвращает результаты.

А что насчет результатов? Нам нужно отдельное правило, чтобы результаты возвращались в POD с API?

Нет, потому что, как только мы разрешим входящий трафик, ответ на этот трафик разрешается автоматически.

Для этого не нужно отдельного правила.

Итак, в этом случае все, что нам нужно, - это Ingress правило, разрешающее трафик из API-pod в db-pod. Это позволит API подключаться к базе данных и выполнять запросы, а также получать результат запросов.

Поэтому при принятии решения о том, какое правило следует создать, нам нужно заботиться лишь о направлении, из которого исходит запрос, который здесь обозначен зеленой анимированой линией.

И нам не нужно беспокоиться об ответе, который обозначен синей линией.

Однако это правило не означает, что POD базы данных сам сможет инициировать подключение к API-pod и выполнить там вызовы API.

Например, db-POD пытается выполнить вызов API из API-pod. Это будет запрещено, потому что теперь это будет egress трафик, исходящий из POD базы данных.

Для него потребуется определить конкретное egress-правило трафика.

Так что я надеюсь, что ты уловил разницу между ними и ясно отличаешь ingress от egress. Я просто хотел убедиться, что тебе ясно, какой тип политики следует выбрать в соответствии с имеющимся у тебя требованиями.

Как я говорил, одна network policy может иметь тип ingress или тип egress, или и то, и другое в тех случаях, когда сторона хочет разрешить входящие соединения, а также сама выполнять внешние вызовы.

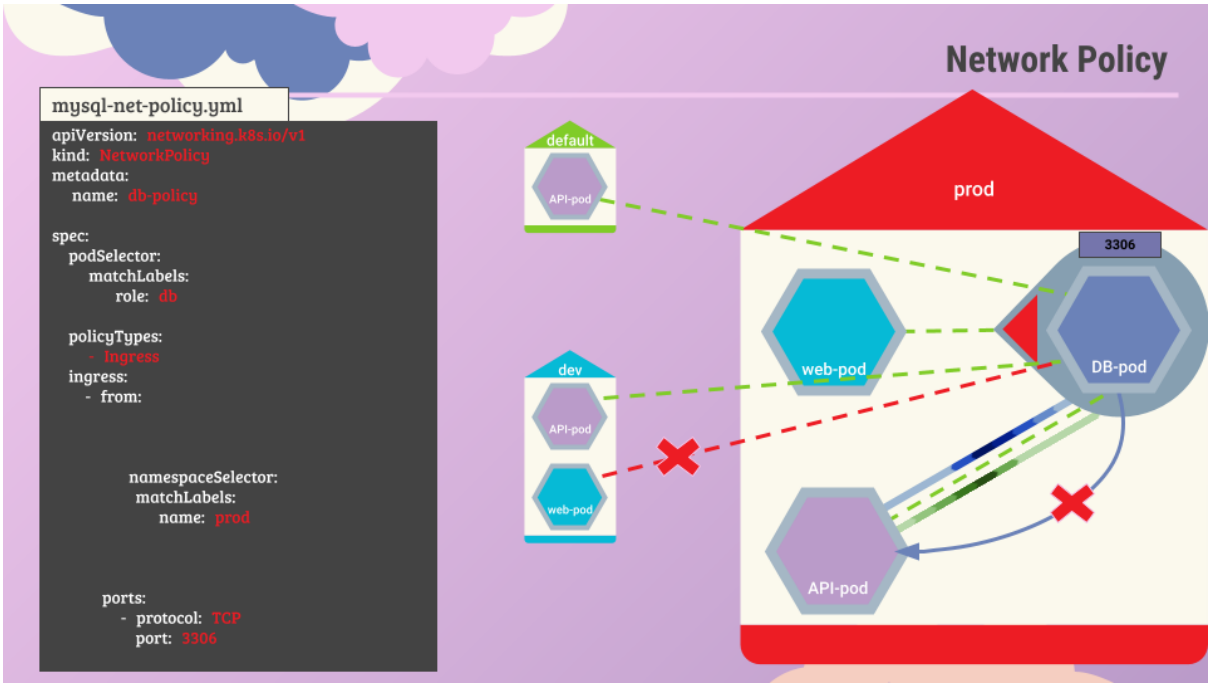
Итак, в данный момент нашему примеру требуется только один тип ingress.
Теперь, когда мы определились с типом политики, следующим шагом будет определение особенностей этой политики.

Мы создали раздел под названием Ingress, в котором можем указать несколько правил.

Каждое правило имеет поле from, и ports. Как ты наверное догадался, поле from определяет источник трафика, которому разрешено проходить в POD базы данных. Здесь мы используем селектор POD и предоставим метки API-pod.

Поле ports определяет номер порта и тип протокола куда разрешается трафик.
В данном случае это порт 3306 с протоколом TCP.

Все. Это создало политику, которая блокирует весь трафик в db-pod, за исключением трафика из API-pod.



Ок, но что, если в кластере есть несколько API-pod с одинаковыми метками, но в разных namespaces.

Например, здесь у нас есть разные пространства имен для тестовой среды, разработки и продакшена. В каждой из сред подняты PODs с API-сервером с одинаковыми метками.

Текущая политика позволяет любому POD из любого пространства имен с соответствующими метками достигать POD базы данных.

Мы хотим позволить только POD из пространства prod достигать db-pod.
Так как же нам это сделать?

Для этого мы добавляем новый селектор, который называется namespaceSelector и используем его вместе со старым, который обрабатывает метки PODs.
Для namespaceSelector мы снова используем matchLabels, в которой указываем имя метки в namespace и ее значение.

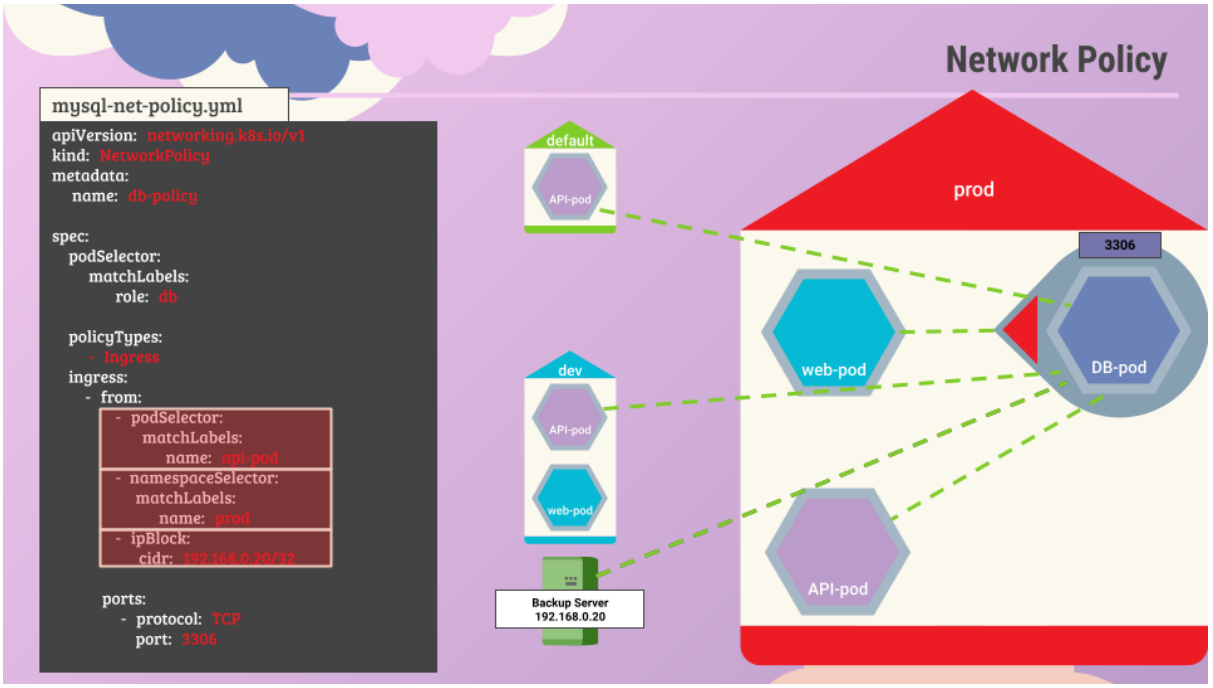
Не забудь, что тебе нужно сначала установить эту метку на пространство имен, чтобы это работало.

Вот что делает селектор пространства имен: он помогает определить, из какого namespace трафик может достигать POD с базой данных.

А что, если у нас есть только namespaceSelector, без podSelector, как я показал здесь.

В этом случае всем PODs в указанном пространстве имен будет разрешено достигнуть db-pod, например такому как web-pod, который был у нас ранее.

Но PODs, находящиеся за пределами этого namespace, не смогут добраться до db.



Давай взглянем на другой вариант использования. Допустим, у нас есть backup-сервер где-то за пределами kubernetes-кластера, и мы хотим разрешить серверу подключаться к POD базы данных.

Поскольку этот сервер резервного копирования развернут не в этом кластере, механизмы namespaceSelector и podSelector не будут работать. Они могут работать только с составляющими кластера.

Однако мы знаем IP-адрес backup-сервера, и он равен 192.168.0.20. Мы могли бы настроить сетевую политику, чтобы разрешить трафик, приходящий с определенных IP-адресов.

Для этого мы добавляем новую запись в под свойство from, с названием ipBlock. IP-блок позволяет указать диапазон IP-адресов, с которых можно разрешить трафику попадать в POD.

Итак, это три отдельных селектора под параметром from в ingress. И это также применимо к egress, я покажу уже скоро.

Ок, у нас есть podSelector, позволяющий выбирать PODs по labels, у нас есть namespaceSelector для выбора пространств имен по их меткам, и у нас есть селектор ipBlock, для выбора диапазонов IP-адресов.

Их можно разделить на разные отдельные правила или использовать вместе как части одного.

В этом примере в массиве from у нас есть два элемента. Это два правила. В первом правиле есть селектор POD и селектор namespace, они работают вместе, а во втором правиле только селектор IP-блока.

Таким образом, для разрешения трафика от источников допускается выполнение любого из этих критериев - или селекторы или нужный IP. Т.е. это логика ИЛИ.

Однако в рамках первого правила у нас два критерия: PODs должны быть из определенного namespace и с определенной меткой. И эти оба условия должный быть соблюдены, чтобы правило разрешило соединение.
Т.е. здесь у нас логика И.

А что, если мы разделим их, добавив тире перед namespaceSelector, как я сделал здесь?

Теперь это будут два отдельных правила, т.е. всего их будет уже три.

И это будет означать, что разрешен трафик, соответствующий первому правилу, то есть из любого POD, соответствующего в части метки из любого пространства имен.

Второе правило разрешит ingress всем PODs из prod namespace. Ну а третье останется как было и будет пускать в POD с сервера резервного копирования, поскольку у нас там установлена спецификация ipBlock.

Т.о., теперь у нас есть три отдельных правила, и мы сильно расширили возможности подачи ingress-трафика в db-POD.
Как видишь, такое небольшое изменение может иметь такое большое влияние. Поэтому важно понимать, как работают эти правила, чтобы составлять их в соответствии со своими требованиями.

Ок, теперь давай избавимся от всего этого и вернемся к базовому набору правил и посмотрим на исходящий поток - egress.

Допустим, вместо того, чтобы сервер резервного копирования инициировал копирование, у нас есть агент в db-pod, который периодически отправляет резервную копию в backup-хранилище.

В этом случае трафик исходит из базы данных на внешний сервер. Для этого нам нужно определить правила для egress.

Итак, сначала мы добавляем egress к типам политики, а затем добавляем новый раздел с названием egress, чтобы определить особенности политики.

Вместо `from`, как мы делали с ingress, в случае egress будет `to`. В общем-то, это единственная разница.

Точно также здесь мы можем использовать любой из селекторов, например namespaceSelector, podSelector или селектор ipBlock.

Поскольку сервер хранения для кластера является внешним, мы используем ipBlock и предоставляем адрес в CIDR-нотации.

Порт, на который будет разрешена отправка запроса, находится далее в разделе ports.

Таким образом, это правило разрешает трафик, исходящий из POD базы данных, на внешний сервер резервного копирования по указанному адресу.

Ну вот и все о network policies и rules. Отправляйся в лабораторную и сам поработай с сетевыми политиками.

Увидимся в следующей лекции.