Auth Considerations for Kubernetes

FusionAuth

Auth. Built for Devs, by Devs.

Auth Considerations for Kubernetes

Dan Moore

This book is for sale at
http://leanpub.com/authconsiderationsforkubernetes

This version was published on 2022-04-25

Contents

Introduction

Kubernetes! It seems like the whole world is excited about this new software deployment platform.

From the project docs[1], "Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem."

That's all well and good, but what kind of workloads are being moved over to this platform? There are three types of workloads moving to Kubernetes.

- Legacy monolithic apps. Here you are looking to move to Kubernetes to take advantage of the operational benefits, without necessarily rearchitecting how the applications themselves work.
- A monolithic application or applications you are evolving toward a microservices architecture.
- Existing microservices you are porting over. This might be moving from a homegrown or other orchestration framework.

In each of these cases, the applications running in those workloads probably have the concept of users. In fact, Kubernetes itself, being a platform, has the concept of users.

And where you have users, you have authentication, authorization and user management. In other words, you have auth.

The rest of this book will discuss various aspects of auth from the standpoint of Kubernetes and these workloads so commonly deployed to that platform.

[1] https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

Enjoy!

Auth and Workloads

Kubernetes is used to handle the following kinds of workloads, typically:

- Legacy monolithic apps. Here you are looking to move to Kubernetes to take advantage of the operational benefits, without necessarily rearchitecting how the applications themselves work.
- A monolithic application or applications you are evolving toward a microservices architecture.
- Existing microservices you are porting over. This might be moving from a homegrown or other standardized orchestration framework like Mesos.

How does auth interact with each of these scenarios?

In the first case, legacy apps, your authentication infrastructure probably won't change. While you can use an external, standards based auth service like FusionAuth to add single sign-on (SSO) between your applications, such changes won't be Kubernetes specific and aren't therefore covered in this book.

If you'd like to learn more about SSO, FusionAuth's SSO guide[2] covers that topic in both the general case and implementation with FusionAuth.

In this case, to minimize risk, you'll want to complete a migration to Kubernetes before considering application code changes. Don't change the engine while the car is being towed!

In the latter two situations, however, you'll want to ensure access for both users and services are controlled.

[2] https://fusionauth.io/docs/v1/tech/guides/single-sign-on

In other words, you'll need to consider how to implement authentication and authorization in the context of your Kubernetes based microservices architecture.

First, though, let's take a step back and ask what auth actually is.

What is Auth?

Auth is a portmanteau of authentication and authorization and delineates who is making requests and what actions they are allowed. Typically, auth also covers user management, which is provisioning users who can then be authenticated and authorized.

As mentioned above, with Kubernetes, plenty of complex functionality is built-in, such as scaling and resiliency. That is one major reason to use it for your applications!

But auth is not a built-in Kubernetes service. You'll need to select or build this functionality yourself. In Kubernetes, there are multiple levels of auth to consider: infrastructure and application. Let's do an overview of these concerns; later chapters will cover these in more detail.

Infrastructure Auth

The first type of auth controls access to Kubernetes managed resources. For example, controlling who can start pods or create deployments.

This functionality is well defined and you can use a variety of different mechanisms[3] to enforce access control. Controlling who can view, delete and modify infrastructure is pretty important.

[3] https://kubernetes.io/docs/reference/access-authn-authz/authentication/

If you are delegating to a standalone auth service for this functionality using OpenID Connect (OIDC), be careful about running it in the Kubernetes cluster to which it controls access. You don't want to have the service be unavailable and then be unable to manage your cluster.

Options to avoid this fate include:

- Running a separate Kubernetes cluster for this critical infrastructure.
- Having a fallback auth strategy, such as usernames and passwords.
- Running an auth provider outside of Kubernetes, but internal to the network the cluster is operating on.
- Using a SaaS solution for your auth server. Be aware of any network access limits you may face. Some SaaS providers may be able to run inside your network.

This topic will be covered in more detail in a later chapter.

Application Auth

The other type of auth is at a different level of abstraction: the application layer. That is, how do your applications know who a request is from and what the user or entity has permission to do? Almost all applications have the concept of a user and permissions which control what they have access to.

Let's consider two scenarios mentioned above: evolving an application toward microservices or porting an application already split up into microservices. A greenfield microservices application has the same auth considerations as the ported app.

When evolving a monolithic application towards microservices, you'll need to split the application up into bounded contexts. User

management and auth is a common context and the definitions of the domain objects are well understood by most developers. Therefore auth can be a good option to extract first.

The steps to take would be:

- Choose an auth server and deploy it to Kubernetes.
- Plan the groups and roles your application needs.
- Set up your application to delegate to the auth server. Typically you can use an open-source library to handle the authentication process.
- Modify your application to ensure it can use the results of the authentication process to properly control access using roles or permissions. You may need to tweak how you handle sessions.
- Evaluate your user data and determine if you want to perform a big bang or slow migration of that data.
- Perform a test migration to a staging auth server instance. You can also point your application's staging environment to test that the application integration is correct.
- Migrate your user data and update your application production environment with the new code that delegates authentication to the auth server.

Now you are done with migrating auth to Kubernetes and can begin moving other contexts to Kubernetes as well.

When building a greenfield microservices application, the concerns are slightly different. While you'll want to deploy an auth server to Kubernetes to manage application users, you'll also want the auth system to serve as a secure token service (STS). Using an STS lets you build a zero trust token based strategy. I'll cover that more below.

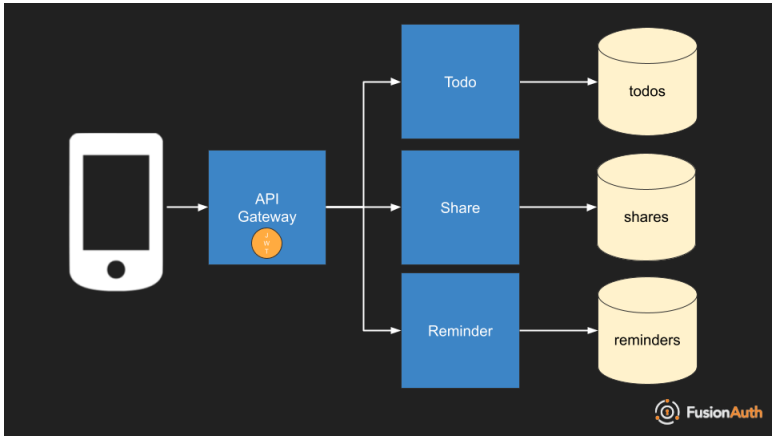
Auth At the Gateway

Microservices are typically fronted by an API gateway or an ingress. This can be done in front of the Kubernetes cluster, using a service mesh ingress gateway or by an NGINX pod running within the cluster.

This gateway also needs to authenticate and authorize requests.

There are a couple of potions. It can use tokens, timebound credentials typically generated by an OAuth grant. Tokens are flexible and secure and can be generated by a standalone auth server. API gateways often support validation of the token, especially if it is a JSON Web Token (JWT), which is a common format.

Below, the API gateway is checking a client token before any access to services behind that gateway is allowed.



An example of service to service communication.

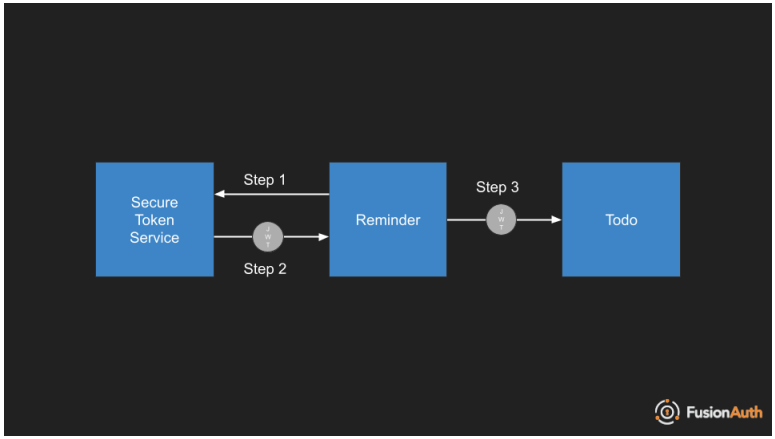
API keys held by a client are an alternative and are conceptually more straightforward. Some API gateways have built-in support for throttling, managing and rotating of API keys.

Auth Between Microservices

Within a microservices system there are multiple ways to authenticate and authorize requests:

- MTLS or client certificates: where a service has a client certificate and presents it as part of each request. This works well for service to service communication, especially when managed by a service mesh like Istio or Linkerd, but client certificates are typically not tied to individual users.
- API keys: similar to client certificates, these arbitrary strings that each service possesses uniquely identify them. If you use these, make sure you plan for rotation. Since these strings aren't cryptographically verifiable, they aren't as secure as client certificates, but they are easier to implement.
- Tokens: these short lived credentials can contain information about the bearer. Often JWTs are used as these tokens. They are typically cryptographically signed so that the contents can be trusted.

An STS can be used to create tokens uniquely tied to one service. Below, the reminder service is performing a client credentials grant to get a token for the Todo service.



An example of service to service communication.

While you can implement more than one of these, tokens are the most flexible. They can be granted to clients outside of your cluster and checked at the API gateway, as mentioned above.

Tokens may contain authorization information such as roles or permissions. They can also contain any other useful data that can be represented as JSON. Tokens can also be minted by an STS for each user request or even for each microservices request.

Auth Considerations

When you are thinking about application auth in Kubernetes managed microservices, consider these scenarios:

- User driven auth requests of microservices: a user request has been received by an application in your cluster. You need to determine who the user is and what actions they can perform.
- Service to service requests: When scheduled jobs run, they may require more than one microservice. You want to know that microservice A is authorized to access microservice B.

- Requests from one microservice on behalf of a user to another. This is a hybrid of the previous two situations. A request comes in and one microservice handles it but needs to access other services on behalf of the user. This hybrid is useful but optional.

You have options for token handling when processing each of these types of requests:

- Pass the token provided by a user to each microservice. The API gateway does no processing.
- Mint a new token at the API gateway or right after the request passes through it. This new token can have a subset of the information provided by a user, a different expiration time and be signed by a different key.
- Convert a token provided by a client into form parameters or a header to remove all token processing from microservices.
- Require each request between microservices to be authenticated by a valid, microservice specific token, possibly including user specific details from an upstream provided token.

Which of these makes sense depends on what information is provided in the token the API gateway processes, how much information each microservice needs, the risk of an access token being compromised, and whether you want to distribute your authorization logic.

Finally, you could build a service to mint these tokens yourself, but there are a robust set of auth providers that are Kubernetes friendly. These handle authentication and embedding authorization information into a token. So, a better choice is to find an auth provider which works well within Kubernetes in the scenarios you have. There are many to choose from, both commercial and open source.

Selecting an Auth Provider

What are criteria for such an auth provider? While the nuts and bolts depend on what kind of application you are migrating over or running, there are common attributes.

- Deployable as a container. If you are running in Kubernetes, any auth required by applications should be containerized. You typically won't want your auth provider to be external to your cluster, especially if you are using it to mint tokens for use within your microservices system.
- Fast. Depending on your scenario, you could be getting tokens or keys from this auth provider every time a request is made from one microservice to another, so pick one that is performant. Even if you are extracting auth from a monolithic application, you still want it to be fast because users want to get to the features you are building, not spend time logging in.
- Scalable. You want to make sure that whatever auth server you are using can scale horizontally, since a key bottleneck for auth is in the CPU-intensive password hashing operation. FusionAuth scales horizontally to dozens of pods which handle thousands of requests per second.
- Standards compliant. Make sure your auth provider supports OAuth2, OIDC and the relevant token RFCs such as RFC 7519[4].
- API first, with support for tools you use to manage your cluster. You'll want to manage the configuration of this auth server the same way you manage your cluster, whether that is with terraform, pulumi or golang code.
- Supports mTLS. This is a nicety; you can add this on with an NGINX sidecar[5], but if you are using client certificates, you'll want to ensure the auth server supports them as well.

[4] https://datatracker.ietf.org/doc/html/rfc7519
[5] https://zhimin-wen.medium.com/https-sidecar-f26ad139dafb

Team Ramifications

Once an auth server is set up and available, developers won't have to worry much about user security and authentication or authorization. Instead of writing login pages and ensuring tokens are minted correctly, developers can offload auth to the external service. They'll be able to focus on writing business logic.

For devops folks and operators, an auth server offers a standard way to implement authentication, authorization and user management. They will need to decide whether or not to use an auth server at the infrastructure level, though.

Both developers and devops practitioners should collaborate on the correct types of credentials to use within and at the boundaries of applications running within Kubernetes: token, API key, client certificates or some combination of all three.

Summary

Kubernetes allows easier development of software by automating parts of the software lifecycle, including deployments, rollbacks and scaling. However, it doesn't provide all the common infrastructure.

Authentication and authorization are components that require a Kubernetes friendly third party solution and some integration work. However, by carving auth out as a separate service and using tokens as short lived credentials, you'll allow developers to focus on writing business logic.

Types of Kubernetes Auth

When considering auth inside your Kubernetes cluster, it's good to think of three different types. Each has different needs and requirements and implementation choices.

The first is infrastructure. The second is service-to-service. And the third is authentication and authorization on a given request. Let's look at each in turn.

Infrastructure

This is the control plane layer. This determines who can do what to your Kubernetes cluster. This includes such tasks as:

- updating a deployment
- deleting a node
- adding a secret

This is the authentication and authorization discussed in the Kubernetes documentation[6].

As outlined there:

> API requests are tied to either a normal user or a service account, or are treated as anonymous requests. This means every process inside or outside the cluster, from a human user typing kubectl on a workstation,

[6] https://kubernetes.io/docs/reference/access-authn-authz/authentication/

> to kubelets on nodes, to members of the control plane, must authenticate when making requests to the API server, or be treated as an anonymous user.

So any API request that needs to read from, add to, or modify Kubernetes configuration must be authenticated (unless the request is available to anonymous users). The Kubernetes documentation outlines all supported options for both service accounts and user accounts, so this chapter won't cover them in detail.

A wide variety of options are supported, including

- Custom headers using an authenticating proxy
- OIDC tokens generated by an external IdP
- Client certificates
- Webhooks which receive a token and can validate access

There's also an impersonation option, which allows users to "take on" the identity of other users, to test access or otherwise troubleshoot.

Once users are authenticated, information about them, such as username, group, and resources requested, is available to Kubernetes authorizers[7]. These are again well documented, but support the following methods of determining access to particular resources:

- ABAC: where policies are combined and evaluated
- RBAC: where roles associated with the user control access
- Webhooks, which fire to a known destination; the response is what determines access

If relying on external sources to determine user resource access, such as an OIDC server or webhooks, you'll want to make sure you have another means of authentication independent of that external

[7] https://kubernetes.io/docs/reference/access-authn-authz/authorization/

source. This allows you to modify the configuration of your cluster when those external resources are unavailable.

This is covered in more detail in the next chapter.

Application Auth

When you have containers running on Kubernetes, there are another two types of auth entirely different from the infrastructure auth outlined above.

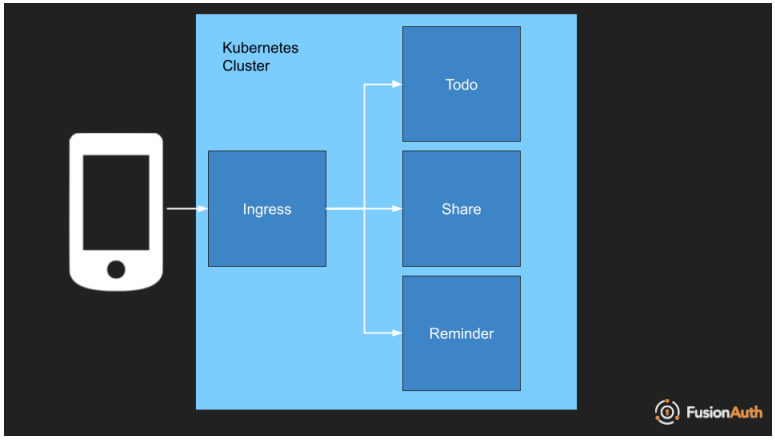


Diagram of todo application in kubernetes.

For instance, if you are running a todo application like the one diagrammed above, you need to make sure a user Alice has access to Alice's todos and a user Bob has access to Bob's todos, but neither Alice nor Bob should have access to the other's data.

This authentication and authorization happens within the todos application and it has nothing to do with the access decisions around nodes and Kubernetes API.

There are two common types of authentication within the application:

- service to service communication
- user initiated requests

Service to Service Communication

You want to lock down communication between the constituent parts of your application. You have a few options, but a choice is mutual TLS. Mutual TLS uses client x.509 certificates and the TLS protocol to authenticate and authorize different parts of your system.

Suppose the reminder service from the application above needs information from the todo service. There's a new feature being built. The reminder service will send an email to every user who has a todo with a due date falling in the next 24 hours. Therefore the reminder service needs to query the todo service.

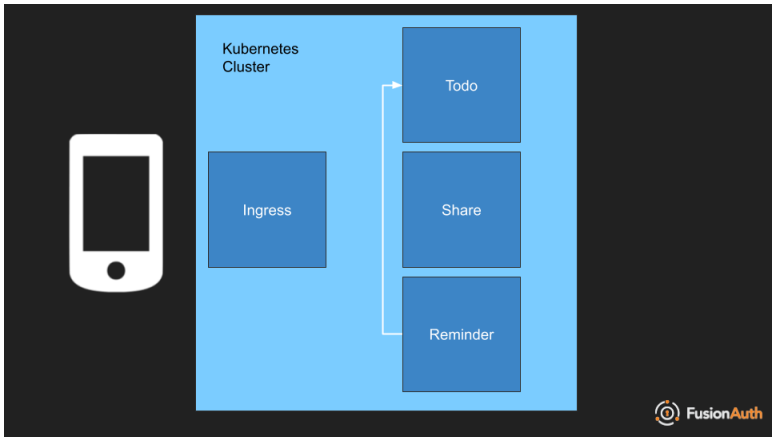


Diagram of service to service communication application in kubernetes.

When building this feature, you'll want to ensure:

- that the reminder service can access the data it needs

- that the todo service isn't open to any unauthorized access

Because the reminder service is making the request, mutual TLS is a good solution. Each service can have a certificate and they can mutually verify them. This can be done manually, but a far simpler solution is to use a service mesh such as Istio or Linkerd, because they'll take care of the certificate provisioning, the ambassador pods in between your services and the renewals.

If you are using Istio, enable strict mutual TLS authentication using this configuration:

```
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: "default"
spec:
  mtls:
    mode: STRICT
```

You'd apply it using a command like this:

```
kubectl apply -n istio-system -f - <<EOF
```

You can read more about mutual TLS authentication in Istio[8].

Since every service in Istio is transparently associated with a client certificate, once mutual TLS is enabled, you can enforce authorization rules.

Continuing with the example above, the reminder service can call the todo service, but not the reverse. Here's an example configuration. The first enables the reminder service to call the todo service, but only with the `GET` HTTP method:

[8] https://istio.io/latest/docs/tasks/security/authentication/mtls-migration/

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: "todos-viewer-allow"
  namespace: default
spec:
  selector:
    matchLabels:
      app: todos
  action: ALLOW
  rules:
  - from:
    - source:
        principals: ["cluster.local/ns/default/sa/reminde\
r"]
    to:
    - operation:
        methods: ["GET"]
```

And the following policy denies the todo service access to the reminder service.

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: "todos-viewer-allow"
  namespace: default
spec:
  selector:
    matchLabels:
      app: reminder
  action: DENY
  rules:
  - from:
    - source:
        principals: ["cluster.local/ns/default/sa/todos"]
```

This setup provides service level authentication and enforces access control.

However, you don't have to use a service mesh for service to service communication. You could provide every service with a unique static identifier which was rotated regularly (an API key) and implement the exact same type of service to service authentication. You could also use the OAuth client credentials grant and treat each service or endpoint as a separate resource.

There are many ways to solve this issue, but at the root, each recognizes the service as the requesting entity.

However, what happens when a user is involved? Let's look at that next.

Auth for Requests

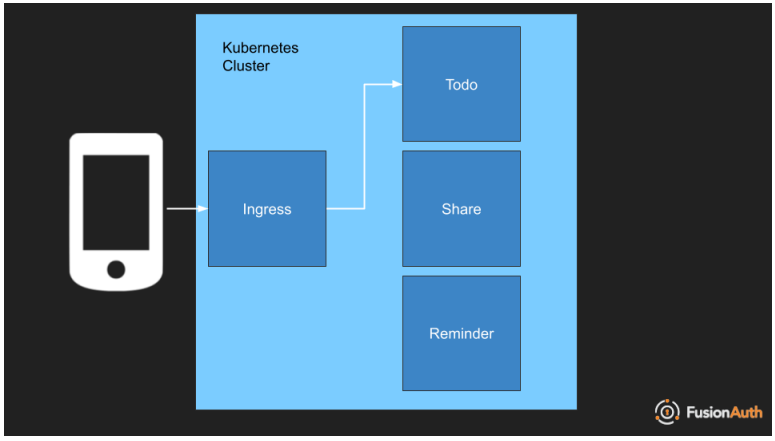


Diagram of user request.

When a request for a todo comes in, it is associated, as mentioned above, with a particular user such as Alice or Bob. This is an additional layer of authentication and authorization which client

certificates or the other methods mentioned previously can't help with. In this case you want to reach for tokens.

Tokens are typically provided by the requesting client and are the result of something like an OAuth grant. They are very often JSON Web Tokens and contain a payload which looks similar to this:

```
{
  "aud": "85a03867-dccf-4882-adde-1a79aeec50df",
  "exp": 1644884185,
  "iat": 1644880585,
  "iss": "acme.com",
  "sub": "00000000-0000-0000-0000-000000000001",
  "jti": "3dd6434d-79a9-4d15-98b5-7b51dbb2cd31",
  "authenticationType": "PASSWORD",
  "email": "admin@fusionauth.io",
  "email_verified": true,
  "applicationId": "85a03867-dccf-4882-adde-1a79aeec50df",
  "roles": [
    "ceo"
  ]
}
```

Again, depending on your implementation, you may be able to configure a service mesh to examine claims in the token, such as the `roles` or `sub` claims. The former controls what roles a user has, while the latter is the identifier for a user. You can also use an ambassador container to examine these claims, or do so inside your microservices.

Implementation options and more about tokens as an auth solution for Kubernetes applications are discussed in detail in a future chapter..

What's important to know is that the auth information is included in the token, and can be shared between the various services to

ensure they only offer up data or functionality that is appropriate for this user.

On Behalf Of Requests

However, there is an interesting subset of user requests. There can be cases where you want to make a request of a service on behalf of a user. To do so, modify the token from the request with additional information about the service making a request.

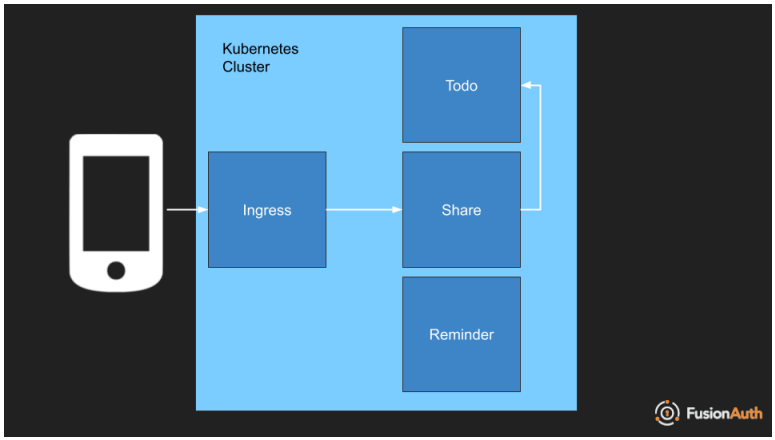


Diagram of on behalf of request.

For example, in the todos application, one feature would be todo sharing: Alice might share a todo with Bob. In this case, when Bob requests his shared todos, the share microservice will need to call the todo service. But the request must include information specifying it is doing so on behalf of Bob, not itself.

This can either be done via a standardized OAuth grant[9], if your token minting library or application supports it. Alternatively, you could re-mint the token or have a secondary layer of authentication by passing the token as well as an identifier of the share service.

[9] https://datatracker.ietf.org/doc/html/rfc8693/

Conclusion

This chapter covered three different types of Kubernetes auth, each protecting a different resource or method of communication:

- infrastructure
- service-to-service
- request/response

Each of these is important in ensuring that your application is appropriately secured.

Implementing RBAC in Kubernetes with FusionAuth

Role-based access control (RBAC) is an authorization protocol that restricts system access based on the roles of a user within an organization. In RBAC, permissions are assigned to roles and roles are assigned to users so that no user is directly assigned a permission. A user is then granted the permissions available to the role(s) they have been assigned.

RBAC can also be applied in the context of Kubernetes. If you want to restrict access to the various resources (pods, services, etc.) in your Kubernetes cluster, you can employ RBAC. The Kubernetes API server supports RBAC natively[10] by making use of Roles and ClusterRoles.

In this chapter, you'll implement RBAC in Kubernetes using FusionAuth[11]. However, these instructions will work with any OIDC compliant auth server.

Doing so can help protect the infrastructure layers of your Kubernetes cluster from mistaken or undesired access.

What is RBAC?

When you're working with multiple users, you need some kind of authorization system in place that restricts or grants users access

[10] https://kubernetes.io/docs/reference/access-authn-authz/rbac/
[11] https://fusionauth.io

to different parts of the system. A developer in your organization should not be able to access the billing section, and similarly, an accountant shouldn't have access to your project's source code.

If you have a small number of users, it's tempting to attach the permissions directly to the user. You might create a table in your database and store the user-permission assignments there. However, as the number of users grows, you'll quickly start to see repetitions. For instance, you might have several users who require the same set of administrative permissions. Keeping track of all these permissions can become challenging and tedious.

RBAC addresses this problem by assigning permissions to *roles*. As an example, suppose you have "Admin" and "Viewer" roles for a project. The Admin role has access to all the resources in the project and the ability to create, update, or delete any resource. The Viewer role, however, can only view resources and cannot modify or create resources. With RBAC, you can assign the Admin role to users who need administrative access, and the Viewer role to users who only need read-level access. Now, you don't have repetitions and it's easier to update permissions for a role.

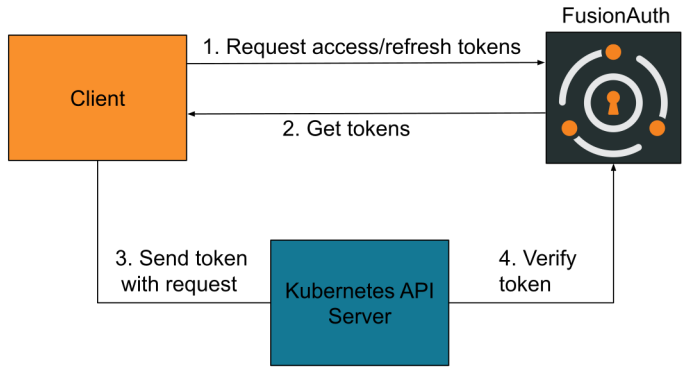
A Kubernetes cluster has many different resources that need to work in sync with each other in order to properly function. Using RBAC, you can employ a proper authorization policy for your cluster. For example, you can guarantee that a developer of one project cannot accidentally make changes to another project, nor delete a pod or service and bring the entire system down.

This chapter uses the OpenID Connect[12] (OIDC) protocol to communicate with FusionAuth to implement RBAC in Kubernetes.

When a client like `kubectl` wants to connect to the Kubernetes API, it will use the Authorization Code grant to get an access and refresh token from FusionAuth. This token will be used to authenticate with the Kubernetes API. Then the API server will talk to FusionAuth to verify that the client-provided token is valid.

[12] https://openid.net/connect/

Finally, it will use the roles of the authenticated user to validate whether the client is allowed to perform the operation or not.



An architecture diagram of K8s when using OIDC for RBAC.

Implementing RBAC in Kubernetes

Before you get started, there are a few prerequisites. First, you'll need a Kubernetes cluster with `kubectl` configured. This chapter uses minikube[13] and some minikube-specific features. However, it's easy to adapt the steps for your Kubernetes cluster of choice.

If you are using minikube, you'll also need to install VirtualBox[14] since you'll be using the VirtualBox driver[15] for minikube.

cfssl and cfssljson[16] will also need to be installed to generate SSL certificates for the FusionAuth installation.

Finally, Helm[17] is required for installing the required software packages, such as FusionAuth, ElasticSearch and PostgreSQL.

[13] https://minikube.sigs.k8s.io/
[14] https://www.virtualbox.org/wiki/Downloads
[15] https://minikube.sigs.k8s.io/docs/drivers/virtualbox/
[16] https://github.com/cloudflare/cfssl
[17] https://helm.sh/

Deploying FusionAuth to Kubernetes

The first step of implementing RBAC in Kubernetes is to deploy FusionAuth to Kubernetes.

To begin, start the minikube cluster. You'll use the VirtualBox driver to make modifying the virtual machine (VM) easier in future steps.

```
minikube start --cpus 4 --memory 5g --driver=virtualbox
```

Deploy PostgreSQL to the cluster using Helm:

```
helm repo add bitnami https://charts.bitnami.com/bitnami
helm install pg-minikube bitnami/postgresql \
  --set postgresqlPassword=password
```

This command deploys PostgreSQL to the minikube cluster and creates a user named `postgres` with password `password`. To update the password, you can change the `postgresqlPassword` value.

List all the pods and make sure the PostgreSQL pod is ready:

```
kubectl get pods -n default -o wide
```

The output should be similar to the following:

```
NAME                          READY   STATUS    RESTARTS   A\
GE   IP           NODE       NOMINATED NODE   READINESS G\
ATES
pg-minikube-postgresql-0   1/1     Running   0          8\
2s   172.17.0.3   minikube   <none>           <none>
```

If it doesn't show `1/1` in the `Ready` column, wait a bit longer for it to start.

The next step is to deploy Elasticsearch[18] using Helm. First, add the repo:

[18] https://www.elastic.co

```
helm repo add elastic https://helm.elastic.co
```

Download the recommended configuration using curl:

```
curl -O https://raw.githubusercontent.com/elastic/Helm-ch\
arts/master/elasticsearch/examples/minikube/values.yaml
```

Finally, install Elasticsearch:

```
helm install es-minikube elastic/elasticsearch -f values.\
yaml
```

Just like you did previously, list the pods:

```
kubectl get pods -n default -o wide
```

You should now see three Elasticsearch pods. Make sure they are ready before continuing:

```
NAME                        READY   STATUS    RESTARTS   A\
GE     IP           NODE       NOMINATED NODE   READINESS\
 GATES
elasticsearch-master-0      1/1     Running   0          1\
18s    172.17.0.5   minikube   <none>           <none>
elasticsearch-master-1      1/1     Running   0          1\
18s    172.17.0.6   minikube   <none>           <none>
elasticsearch-master-2      1/1     Running   0          1\
18s    172.17.0.4   minikube   <none>           <none>
pg-minikube-postgresql-0    1/1     Running   0          5\
m48s   172.17.0.3   minikube   <none>           <none>
```

Finally, it's time to deploy FusionAuth. First, add the Helm repo:

```
helm repo add fusionauth https://fusionauth.github.io/cha\
rts
```

Download the configuration file:

```
curl -o values.yaml https://raw.githubusercontent.com/Fus\
ionAuth/charts/master/chart/values.yaml
```

Now, edit `values.yaml` and set the following values:

1. **database.host**: `pg-minikube-postgresql.default.svc.cluster.local`
2. **database.user** and **database.root.user**: `postgres`
3. **database.password** and **database.root.password**: your PostgreSQL password (if you copied the installation command above and ran it unmodified, this is `password`)
4. **search.host**: `elasticsearch-master.default.svc.cluster.local`

Now install FusionAuth with the following command:

```
helm install my-release fusionauth/fusionauth -f values.y\
aml
```

List all the deployments and ensure the `my-release-fusionauth` deployment is ready.

```
kubectl get deployments -o wide
```

You should see something like this:

```
NAME                    READY   UP-TO-DATE   AVAILABLE   \
AGE   CONTAINERS   IMAGES                              SEL\
ECTOR
my-release-fusionauth   1/1     1            1           \
42s   fusionauth   fusionauth/fusionauth-app:1.30.1   app\
.kubernetes.io/instance=my-release,app.kubernetes.io/name\
=fusionauth
```

Adding an SSL ingress

Next, you need an ingress[19] to expose the FusionAuth deployment so that you can access it with the browser on your host machine.

For that, enable the ingress add-on:

```
minikube addons enable ingress
```

Before you can access FusionAuth, create SSL certificates so that it can be accessed over HTTPS. This is necessary to get OIDC working.

First, create a file named `ca-config.json` with the following content:

```
{
  "signing": {
    "default": {
      "expiry": "43800h"
    },
    "profiles": {
      "server": {
        "expiry": "43800h",
        "usages": [
```

[19] https://kubernetes.io/docs/concepts/services-networking/ingress/

```
            "signing",
            "key encipherment",
            "server auth",
            "client auth"
          ]
        }
      }
    }
}
```

Create another file named `ca-csr.json` with this code:

```
{
    "CN": "fusionauthca",
    "key": {
        "algo": "rsa",
        "size": 2048
    }
}
```

Finally, another file named `fusionauth.json` needs to be created:

```
{
    "CN": "fusionauth",
    "hosts": ["fusionauth.local"],
    "key": {
        "algo": "ecdsa",
        "size": 256
    }
}
```

Run the following commands to generate the certificates:

```
mkdir -p ssl
cfssl gencert -initca "ca-csr.json" | cfssljson -bare "ss\
l/fusionauth-ca" -
cfssl gencert \
    -ca="ssl/fusionauth-ca.pem" \
    -ca-key="ssl/fusionauth-ca-key.pem" \
    -config="ca-config.json" \
    -profile=server \
    "fusionauth.json" | cfssljson -bare "ssl/fusionauth"
```

If you see any errors with this command, make sure you named all the JSON files as specified above.

Once the command finishes, you should see an `ssl` directory with the necessary SSL certificates and keyfiles.

You should see six files:

```
fusionauth-ca-key.pem
fusionauth-ca.csr
fusionauth-ca.pem
fusionauth-key.pem
fusionauth.csr
fusionauth.pem
```

The next step is to upload the certificates to the cluster:

```
 tar -c -C ssl fusionauth-ca.pem | ssh -t -q -o StrictHos\
tKeyChecking=no \
    -i "$(minikube ssh-key)" "docker@$(minikube ip)" 'sud\
o tar -x --no-same-owner -C /var/lib/minikube/certs'
```

Create a Kubernetes secret using the certificates:

```
kubectl create secret tls fusionauth-cert --cert="ssl/fus\
ionauth.pem" --key="ssl/fusionauth-key.pem"
```

Now that you have the SSL certs, it's time to create the ingress. Save the following in `ingress.yaml`:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-release-fusionauth
  labels:
    app.kubernetes.io/name: fusionauth
    app.kubernetes.io/instance: my-release-fusionauth
  annotations:
    kubernetes.io/ingress.class: nginx
spec:
  rules:
  - http:
      paths:
      - backend:
          service:
            name: my-release-fusionauth
            port:
              number: 9011
        path: /
        pathType: Prefix
    host: fusionauth.local
  tls:
    - hosts:
        - fusionauth.local
      secretName: fusionauth-cert
```

Create the ingress:

```
kubectl create -f ingress.yaml
```

Set up /etc/hosts

Next, let's give our ingress a nice domain name, accessible from our host machine.

Run the following command to get the IP of the minikube cluster:

```
minikube ip
```

The output should look something like this:

```
192.168.59.101
```

Edit `/etc/hosts`. If you are using Windows, edit the `c:\windows\system32\drivers\etc\hosts` file. Add the following entry:

```
192.168.59.101  fusionauth.local
```

Make sure to use the actual IP address of your cluster. This ensures that the hostname `fusionauth.local` resolves to the cluster IP address.

Visiting https://fusionauth.local[20] should now take you to the FusionAuth setup page. You'll find a certificate error because of the self-signed certificate, but you can safely ignore it. Accept the warning and you'll see the FusionAuth setup page.

[20] https://fusionauth.local

Configure FusionAuth

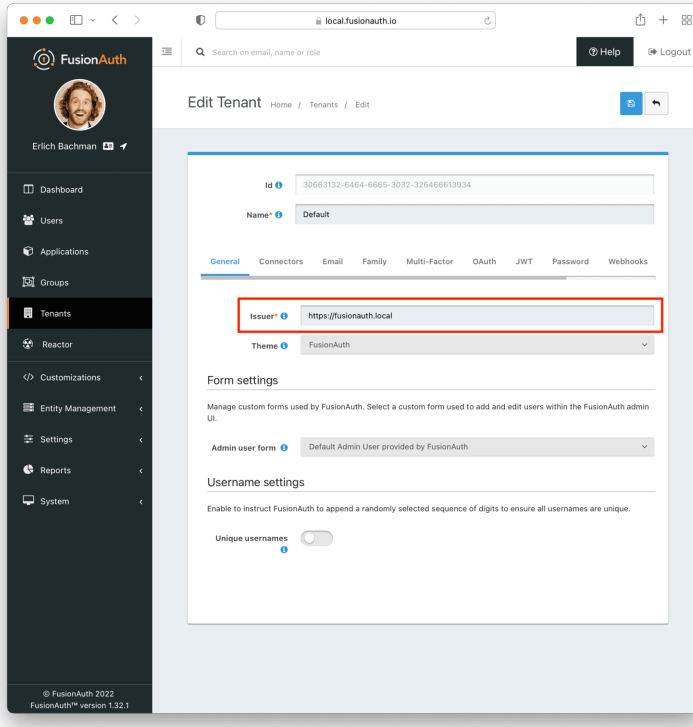
On the FusionAuth setup page, create a user account to access the dashboard. For more on the setup wizard, see this tutorial[21].

After you create the user, you'll be at the dashboard.

Before setting up OIDC with Kubernetes, you'll need to change a few default settings.

First, visit the "Tenants" page from the sidebar and edit the "Default" tenant. Change the "Issuer" field from the default value of `acme.com` to `https://fusionauth.local` and save the changes. Kubernetes will validate the Issuer field during authentication, so it's vital that this is the same as the FusionAuth domain.

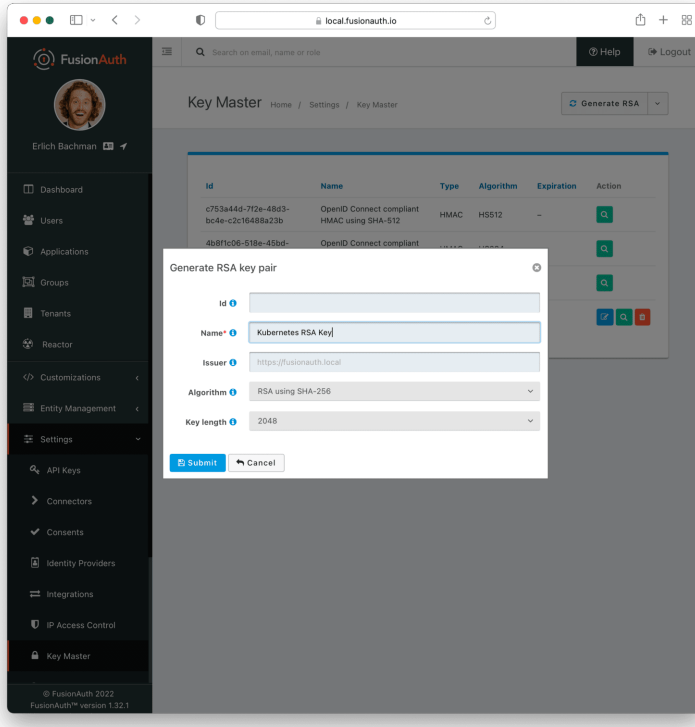
[21] https://fusionauth.io/docs/v1/tech/tutorials/setup-wizard/



Modifying the tenant settings to have a correct issuer.

The default JWT signing of FusionAuth uses the HS256 algorithm. However, the Kubernetes plugin you'll use to authenticate expects the RS256 algorithm. So let's create an RS256 key. Visit "Settings", then "Key Master" and select "Generate RSA" from the dropdown on the top-right.

Give it a name such as `Kubernetes RSA Key`, and click "Submit". The other default settings can remain the same.

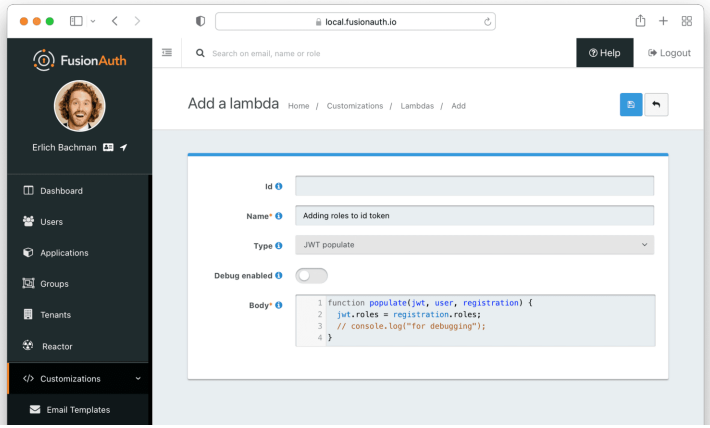


Creating an asymmetric signing key in FusionAuth.

The Kubernetes API server uses the Id token to authenticate users, so you need to add the roles claim to the ID token using a lambda[22]. Navigate to "Customizations" then "Lambdas" and create a new lambda. Give it a name, select `JWT populate` as the "Type", enter the following code in the "Body" field, then save it:

[22] https://fusionauth.io/docs/v1/tech/lambdas/

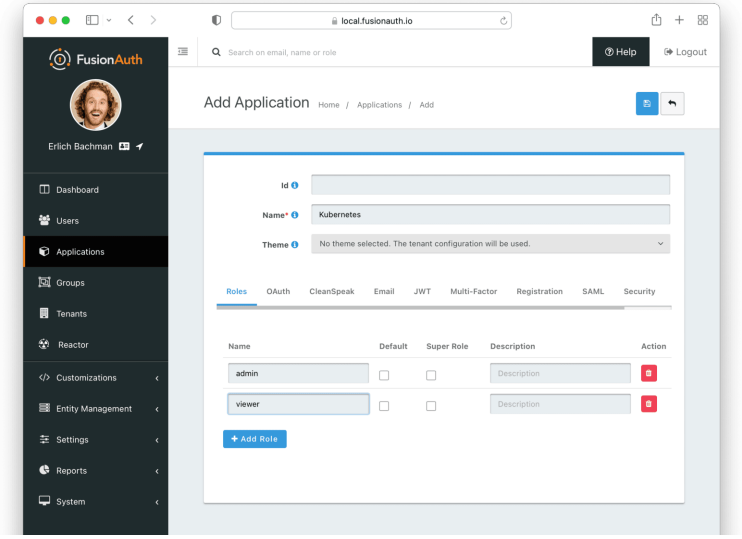
```
function populate(jwt, user, registration) {
  jwt.roles = registration.roles;
  // console.log("for debugging");
}
```



Creating a new lambda.

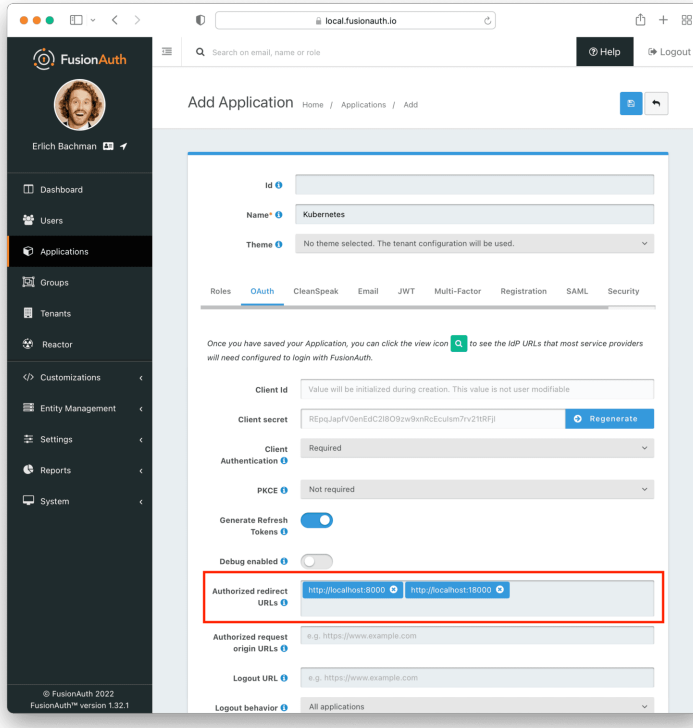
The final step in configuring FusionAuth is to create an application. Visit the "Applications" section and create a new application. Give it a descriptive name, for example `Kubernetes`.

The "Roles" tab is where you can create new roles for this application. Go ahead and create two roles, `admin` and `viewer`. Later, you'll map these roles to Kubernetes roles to enable RBAC.



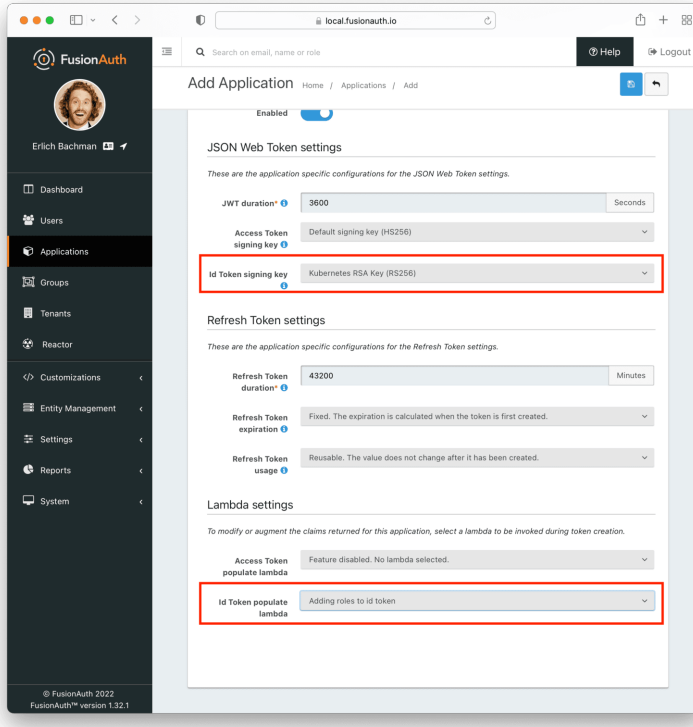
Adding roles to a new application.

In the "OAuth" tab, add `http://localhost:8000` and `http://localhost:18000` in the "Authorized redirect URLs" field. These URLs are used by the `oidc-login` Kubernetes plugin to redirect the user after authentication.



Configuring the OAuth settings for a new application.

In the "JWT" tab, first enable JWT creation by toggling the switch. In the "Id Token signing key" dropdown, select the RS256 key you created above. In the "Id Token populate lambda" dropdown, select the lambda you created earlier.

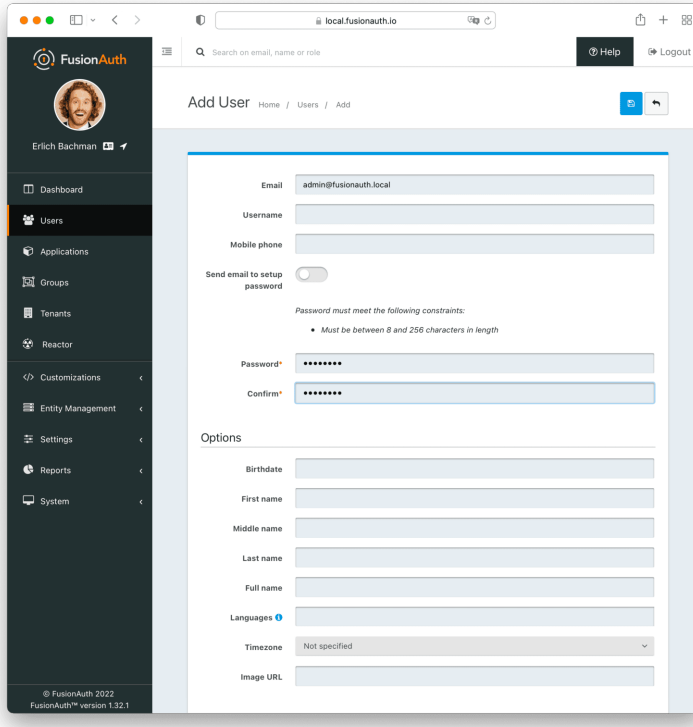


Configuring the JWT settings for a new application.

Finally, save the application.

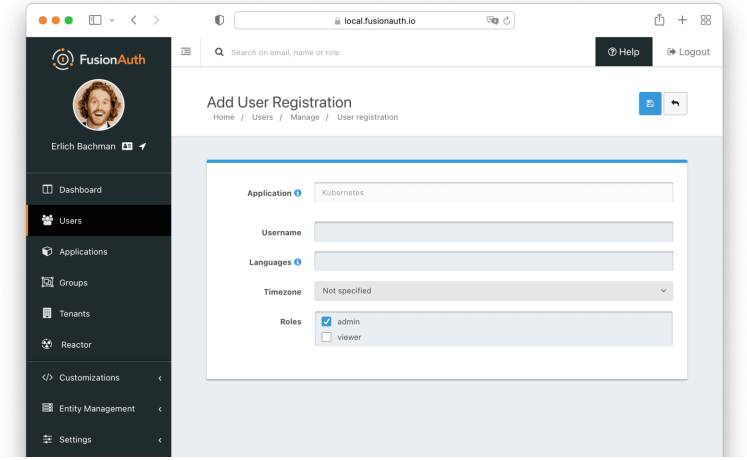
For demonstration, let's create two users. One will be given the `admin` role, and another will be assigned the `viewer` role.

Visit the "Users" page, and add a new user. Enter `admin@fusionauth.local` in the email field and disable "Send email to setup password" Enter a password for this new user and save it.



Adding an admin user.

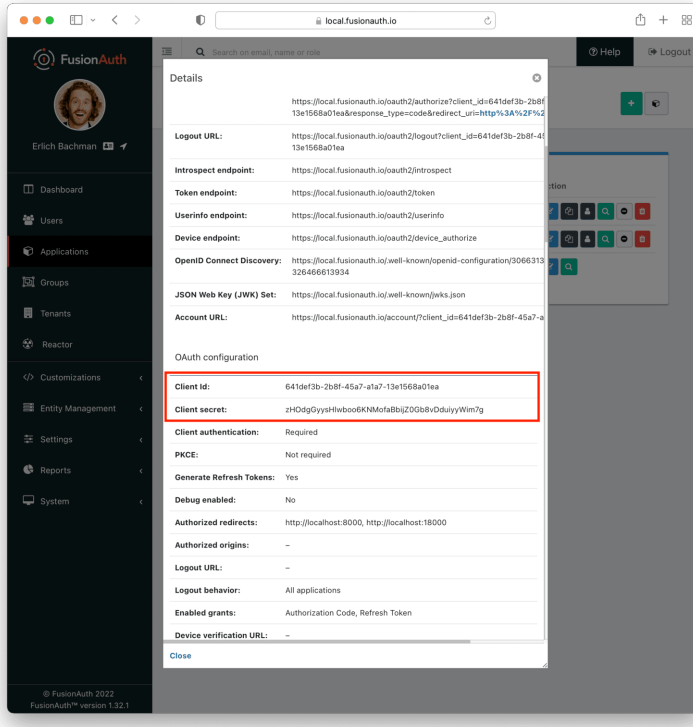
Click the "Add registration" button, select the `Kubernetes` application and the `admin` role, and save the registration.



Adding a registration for the Kubernetes application for the admin user.

Repeat the above process to create another user. Use `viewer@fusionauth.local` as the email and assign this user the viewer role.

Finally, click the view icon next to the `Kubernetes` application on the applications page and write down the Client Id and Client secret.



Getting the client Id and client secret.

Setting up the OIDC mapping in Kubernetes

To set up Kubernetes correctly, first stop the minikube cluster:

```
1 minikube stop
```

Then, turn on natdnshostresolver[23] on the minikube VM:

[23] https://www.virtualbox.org/manual/ch09.html#nat_host_resolver_proxy

```
VBoxManage modifyvm minikube --natdnshostresolver1 on
```

Restart the minikube cluster with additional API server flags. Replace `<YOUR_CLIENT_ID>` with the client Id of the application you obtained from FusionAuth above.

```
 minikube start --driver=virtualbox \
            --extra-config=apiserver.oidc-issuer-url=http\
s://fusionauth.local \
            --extra-config=apiserver.oidc-client-id=<YOUR\
_CLIENT_ID> \
            --extra-config=apiserver.oidc-username-claim=\
email \
            --extra-config=apiserver.oidc-username-prefix\
="oidc:" \
            --extra-config=apiserver.oidc-groups-claim=ro\
les \
            --extra-config=apiserver.oidc-groups-prefix="\
oidc:" \
            --extra-config=apiserver.oidc-ca-file=/var/li\
b/minikube/certs/fusionauth-ca.pem \
            --extra-config=apiserver.authorization-mode=N\
ode,RBAC
```

Using the `--extra-config` flags, the new configurations will be passed to the cluster.

Here are explanations of each of the flags used:

- `--oidc-issuer-url`: sets the OIDC issuer URL (`https://fusionauth.local`)
- `--oidc-client-id`: the application's client Id; provide the client Id copied in the previous step
- `--oidc-username-claim`: tells the API server which claim in the `id_token` is the username (in this case, it's `email`)

- `--oidc-username-prefix`: is used to set a prefix (`oidc:` in this case) to usernames coming from FusionAuth; this prevents possible collisions with already existing Kubernetes users
- `--oidc-groups-claim`: tells the API server which claim in the `id_token` is the users' groups or roles (in this case, it's the `roles` claim which is an array of strings)
- `--oidc-groups-prefix`: similar to `--oidc-username-prefix`, it sets a prefix to roles coming from FusionAuth to avoid collisions
- `--oidc-ca-file`: the certificate authority file which is used to validate the SSL certificate of the issuer
- `--authorization-mode`: enables RBAC authorization mode; the `Node` mode is also specified so that internal authorization is not hampered

Set up Kubernetes cluster roles

Let's create the ClusterRoleBindings[24] for the roles `admin` and `viewer`. These cluster roles will control what a user can do in the Kubernetes cluster and will be assumed by users that are authenticated with FusionAuth.

Save the following in `crb-admin.yaml`:

[24] https://kubernetes.io/docs/reference/access-authn-authz/rbac/#rolebinding-and-clusterrolebinding

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: fusionauth-cluster-admin
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: oidc:admin
```

As you can see from the `subjects.name` field, the `admin` role from FusionAuth is prefixed with `oidc:` and is bound to the `cluster-admin` ClusterRole in the `roleRef.name` field. This gives users in the `admin` FusionAuth-managed role cluster-admin permissions in Kubernetes.

Similarly, Create `crb-viewer.yaml` for the `viewer` role:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: fusionauth-cluster-viewers
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: view
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: oidc:viewer
```

This uses the `view` `ClusterRole` and gives the FusionAuth-managed `viewer` role *only* view access to the cluster resources. You could of

course create more cluster roles and FusionAuth roles should you need more granularity.

Next, create the two `ClusterRoleBindings`:

```
kubectl create -f crb-admin.yaml
kubectl create -f crb-viewer.yaml
```

Getting the Id token to Kubernetes

The Kubernetes API server processes the Id token to authenticate users from the OIDC providers. There are mainly two ways you can get the Id token:

1. Manually using the Authorization Code grant[25]. This is a cumbersome process, and since the API server can't perform a token refresh, you must manually refresh the token when it's expired.
2. Using a CLI or web-based helper like `kubelogin`[26]. `kubelogin` uses a web-based application to perform the authorization code flow and automatically refresh the token once it's expired.

The latter approach is what you'll be using today.

First, install Krew[27] following the instructions in the docs[28], then install the `kubelogin` plugin:

```
kubectl krew install oidc-login
```

[25] https://datatracker.ietf.org/doc/html/rfc6749#page-24
[26] https://github.com/int128/kubelogin
[27] https://krew.sigs.k8s.io/
[28] https://krew.sigs.k8s.io/docs/user-guide/setup/install/

When you run the following command, a new browser window will open and take you to the FusionAuth login page. Use the `admin@fusionauth.local` user to log in. Remember to substitute `<YOUR_CLIENT_ID>` and `<YOUR_CLIENT_SECRET>` when running the command below.

```
 kubectl oidc-login setup \
         --oidc-issuer-url=https://fusionauth.local \
         --oidc-client-id=<YOUR_CLIENT_ID> \
         --oidc-client-secret=<YOUR_CLIENT_SECRET> \
       --insecure-skip-tls-verify
```

Once you're logged in, the browser tab will automatically close. You'll see the Id token logged to the console (there's a lot of output, so you may need to search for it). It'll look something like this:

```
{
  "aud": "b2676ddc-b628-4e52-b7c0-98a45141e5f1",
  "exp": 1639072669,
  "iat": 1639069069,
  "iss": "https://fusionauth.local",
  "sub": "3efe953b-a3b5-42ab-83c9-a492ddf4d933",
  "jti": "b3b4d0ed-43f3-484b-91ee-6a55c9317287",
  "authenticationType": "PASSWORD",
  "email": "admin@fusionauth.local",
  "email_verified": true,
  "at_hash": "mDohX5aBtnehWjIiO5AtZQ",
  "c_hash": "veRYp54qq8nmUhItkELJSg",
  "scope": "openid",
  "nonce": "NSvkZLWU_AH8bBHFmzJwIH8Zf0cSYgRzwk40yA8WPLI",
  "sid": "5d06e4ef-d9bd-42fa-93c3-9521cc0addca",
  "roles": [
    "admin"
  ]
}
```

Verify that the `email` field is correct, and that the `roles` array contains the `admin` role.

Finally, set up the `kubeconfig` with the following command. Make sure to replace `<YOUR_CLIENT_ID>` and `<YOUR_CLIENT_SECRET>`.

```
kubectl config set-credentials oidc \
          --exec-api-version=client.authentication.k8s.io\
/v1beta1 \
          --exec-command=kubectl \
          --exec-arg=oidc-login \
          --exec-arg=get-token \
          --exec-arg=--oidc-issuer-url=https://fusionauth\
.local \
          --exec-arg=--oidc-client-id=<YOUR_CLIENT_ID> \
          --exec-arg=--oidc-client-secret=<YOUR_CLIENT_SE\
CRET> \
          --exec-arg=--insecure-skip-tls-verify
```

Now you can use this context to run kubectl commands. Let's go big, by deleting one of the Elasticsearch cluster nodes:

```
kubectl --user=oidc delete po elasticsearch-master-0
```

Use the `admin@fusionauth.local` account to log in.

The pod will be deleted. You can test with this command if you'd like.

```
kubectl get pods -n default -o wide
```

Don't worry about the pod! It will come back since it's part of a replica. If the pod is already back, take a look at the `AGE` column.

Let's try the other user, with only `view` privileges. First, log yourself out by deleting the cache:

```
rm -rf ~/.kube/cache/oidc-login
```

Then rerun the command which sets up the credentials.

```
kubectl config set-credentials oidc \
          --exec-api-version=client.authentication.k8s.io\
/v1beta1 \
          --exec-command=kubectl \
          --exec-arg=oidc-login \
          --exec-arg=get-token \
          --exec-arg=--oidc-issuer-url=https://fusionauth\
.local \
          --exec-arg=--oidc-client-id=<YOUR_CLIENT_ID> \
          --exec-arg=--oidc-client-secret=<YOUR_CLIENT_SE\
CRET> \
          --exec-arg=--insecure-skip-tls-verify
```

Try to delete the same Elasticsearch pod:

```
kubectl --user=oidc delete po elasticsearch-master-0
```

Use `viewer@fusionauth.local` to log in. This time, you should get an error:

```
Error from server (Forbidden): pods "elasticsearch-master\
-0" is forbidden: User "oidc:viewer@fusionauth.local" can\
not delete resource "pods" in API group "" in the namespa\
ce "default"
```

Voila! RBAC is working as intended. If you feel confident with the RBAC implementation, you can set the `oidc` context as default:

```
kubectl config set-context --current --user=oidc
```

Now, you can run kubectl commands without specifying the `--user=oidc` flag, and it will always authenticate with FusionAuth.

Summary

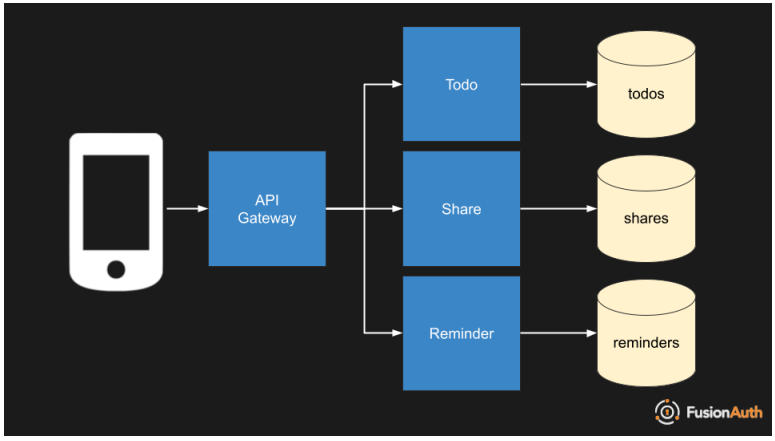
RBAC is an excellent authorization mechanism for managing Kubernetes infrastructure access. It makes permission management easy and flexible.

With an OIDC server, you can have one central location for authentication and authorization information across both your Kubernetes cluster and applications running on it.

Tokens at the Context Boundary

When you are using JWTs as part of your authorization solution in a microservices or Kubernetes based environment, you need to determine where to process them and how fine grained to make them.

Consider this simple microservices based system.



Simple microservices system architecture diagram.

We have three different services, all protected by an API gateway. This API gateway could be running NGINX, Apache, or some other open source system. It could be a commercial package such as HAProxy or Kong. It could also be a cloud vendor managed API gateway, such as an AWS Application Load Balancer or a Google Cloud Load Balancer.

When a request comes in, it will contain an access token. Getting this access token is beyond the scope of this chapter, but

is documented in the "Modern Guide to OAuth", another book by the FusionAuth team. This access token is often a JSON Web Token (JWT), which has intrinsic structure and can be signed and validated. It could use another format, whether custom or a different standard like PASETO.

This chapter will assume that the access token is a JWT, but similar concepts apply to any kind of token. It further assumes that the token is signed by a private key and that the API gateway has access to the corresponding private key to verify the signature is valid.

At the application gateway, two actions must be taken:

- The token's signature is validated. Therefore the token was signed by a trusted party (often an OAuth server).
- The token's claims are examined. Therefore the API gateway knows who the token was created for (the `aud` claim), that it has not expired (the current time is before the `exp` claim and after the `nbf` claim), and more.

But what happens after this? There are four scenarios:

- There is no authorization process within the system. Any request that has been validated by the API gateway is forwarded on. The token is passed on as well and data may be extracted from it, but it is trusted and not validated again.
- The token is passed through. Any request that has been validated by the API gateway is forwarded on. The token may be passed through and validated by each of the microservices. They can extract data from the JWT after the validation.
- The token is re-issued. The token may be parsed apart and re-issued. The data may be the same or sanitized. The signing algorithm, the lifetime and other attributes of the token can be modified. When the token arrives at the microservice, it can be validated and user data can be extracted.

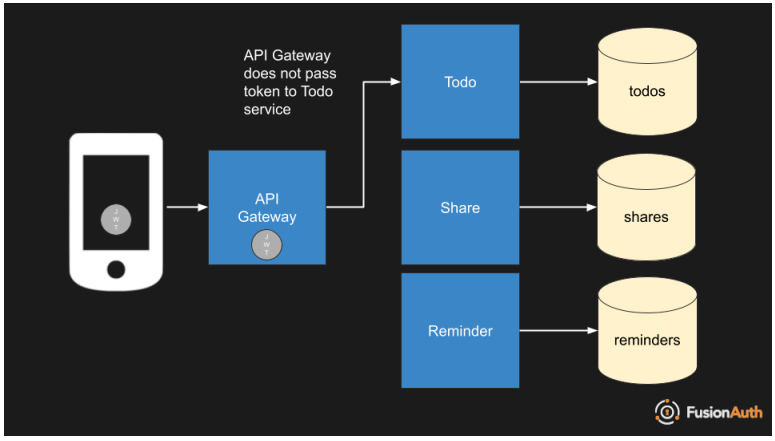
- The token data is completely extracted. Claims are pulled off the token and put into a headers or the body of the request. The API gateway provides an API key which is validated by the microservices.

All of these scenarios have different tradeoffs. Let's look at each one in more detail, with an example request made by a client to retrieve a user's todos.

Trust With No Validation

In this case, the API gateway's stamp of approval is enough for each microservice to trust the tokens provided. There may be additional authorization checks to ensure the request is legitimate, such as mutual TLS provided by a service mesh, but there is no validation of the token signature by the microservice.

The token is provided to the microservice which can decode the payload and examine the claims without worrying about the signature.



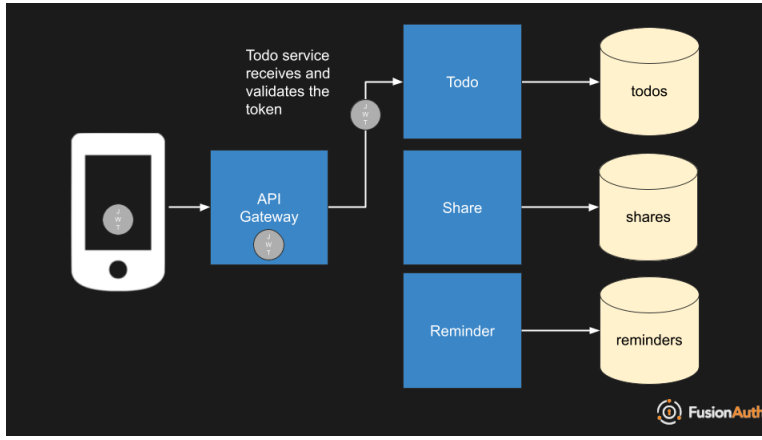
No microservice validation.

The benefits of this approach are simplicity of microservices implementation. They don't have to worry about JWT validation at all. They may have to decode the payload, but that can be done with the golang base64[29] package or other similar packages. In addition, no external network access is required (to retrieve the public keys for signature validation). Because there is no signature validation, processing will be faster.

However, this approach relies on lower layers of the authorization system being correctly implemented. It also foregoes one part of a defense in depth strategy. Be wary of this approach.

Passthrough

Here the token is provided to the microservices and they each validate the signature and the claims independently of the API gateway.



Each service validates the full token.

[29] https://pkg.go.dev/encoding/base64

You'll typically handle this with, in increasing order of effort and customizability:

- a service mesh such as Linkerd[30] or Istio[31]
- an ambassador container running NGINX plus an token processing NGINX library[32] or a similar proxy like airbag[33]
- the code embedded in a library in your microservice which can then validate the signature and claims

For example, if using Istio, you'd apply this command to create a request authentication policy, which confirms information about who created the JWT and validates the signature of the JWT.

This assumes you have a workload in the `development` namespace named `todos`. The JWT is present in the `Authorization` header, as either it was put there in the initial request or the API gateway has placed it there. Additionally, the below configuration assumes there is a prefix of `Bearer`, so the header looks like: `Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpX...`

```
apiVersion: security.istio.io/v1beta1
kind: RequestAuthentication
metadata:
  name: "jwt-auth"
  namespace: development
spec:
  selector:
    matchLabels:
      app: todos
  jwtRules:
  - issuer: "https://example.fusionauth.io"
    jwksUri: "https://example.fusionauth.io/.well-known/j\
```

[30] https://linkerd.io/
[31] https://istio.io/
[32] https://github.com/zmartzone/lua-resty-openidc
[33] https://github.com/Soluto/airbag

```
wks.json"
    outputPayloadToHeader: "X-payload"
  jwtHeader:
  - name: "Authorization"
    prefix: "Bearer "
```

This approach also passes the payload to the microservice so that it can examine claims, such as the `sub` claim, and retrieve data or otherwise process the request based on that information. Istio fails when the `jwksUri` is not accessible or empty. Also, ensure the headers from `jwtHeader` are passed through to all services that need to access them.

Apply these commands to create a request authorization policy, which examines additional claims in the JWT and allows or denies access based on them.

This authorization policy allows all requests to the `todos` workspace for anyone with the `admin` role:

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: "jwt-authz"
  namespace: default
spec:
  selector:
    matchLabels:
      app: todos
  action: ALLOW
  rules:
  - when:
    - key: request.auth.claims[iss]
      values: ["https://example.fusionauth.io"]
    - key: request.auth.claims[roles]
      values: ["admin"]
```

This policy denies access to all other requests with a token from the same issuer.

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: "jwt-authz-deny"
  namespace: default
spec:
  selector:
    matchLabels:
      app: todos
  action: DENY
  rules:
  - when:
    - key: request.auth.claims[iss]
      values: ["https://example.fusionauth.io"]
    - key: request.auth.claims[roles]
      notValues: ["admin"]
```

When setting these rules up in Istio, some debugging commands can be helpful:

First, view the logs of the istiod service:

```
kubectl -n istio-system logs --since=1h istiod-<id> -f
```

This shows you errors similar to:

```
Internal:Error adding/updating listener(s) virtualInbound\
: Provider 'origins-0' in jwt_authn config has invalid lo\
cal jwks: Jwks doesn't have any valid public key
```

Such errors indicate your JWKS endpoint which contains the public keys used to verify the signature of the token are invalid or inaccessible.

Second, look at the listener configs for your pods:

```
istioctl proxy-config listener -n default todos-v3-<id> -\
o json
```

This shows all of the listeners running against a given service. The output will include filters with names like `envoy.filters.http.jwt_authn` and `envoy.filters.http.rbac`. It provides a long JSON output useful for debugging rules.

You can learn more about these commands in the Istio documentation[34] and the troubleshooting documentation[35]

This approach pushes more authentication and business logic to your microservices, though by using a service mesh or ambassador you may be able to keep the microservice relatively ignorant of the implementation. The service still has to have access to the JWKS endpoint for signature validation, which requires external network access or a proxy.

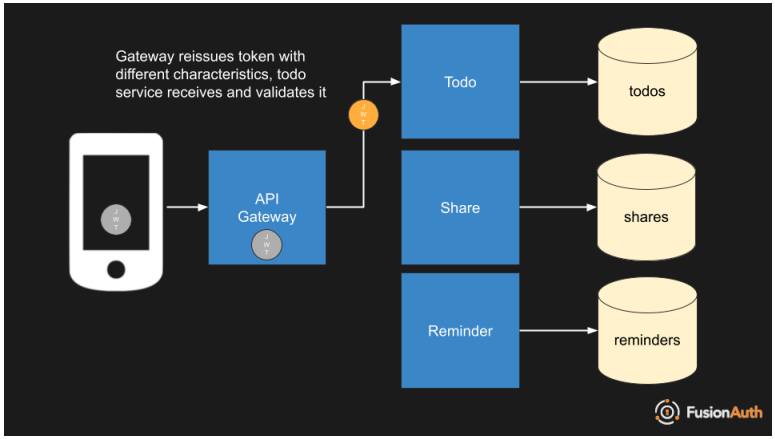
The API gateway remains relatively simple. It still performs validation, but doesn't have to do anything to the token beyond forwarding it. You also gain the benefits of token checking at both layers, so if something is in your network and presents an invalid token, the request will fail.

Re-Issue

In this case, the token is processed at the API gateway. With Kubernetes, you can add an adapter to your ingress, process the request and modify the token.

[34] https://istio.io/latest/docs/reference/config/security/

[35] https://istio.io/latest/docs/ops/common-problems/security-issues/#end-user-authentication-fails



The gateway re-issues the token.

Common ways to modify the token include:

- Removing unneeded information
- Shortening the lifetime
- Using a different signing key

You might want to remove unneeded information that is generated by an external token provider. Such tokens might include extra information which doesn't make sense to the services behind the API gateway. You might modify the `aud` claim to make it more specific as well.

Shortening the lifetime can help improve security. You can shorten the lifetime to between one and five seconds, essentially making this a one-time use token. Therefore, even if the token is somehow exfiltrated, it will be expired.

Finally, using a different signing key is advantageous in a number of ways. This allows you to use an internally managed signing key, and allows you to rotate and revoke it without affecting any system outside of your current environment. You can choose a different, more performant algorithm. You might receive a token

signed with RSA and re-issue it with an HMAC signature. While HMAC requires a shared secret which can be problematic to share between systems, within a trusted system it is acceptable. It also is far faster to verify.

Here's code which takes an RSA signed JWT and re-signs it with an HMAC secret. You'll want to make sure the HMAC secret is stored securely, either within an internal token generating service or using Kubernetes Secrets.

```
1  package main
2
3  import (
4          "crypto/rand"
5          "crypto/rsa"
6          "fmt"
7          "github.com/golang-jwt/jwt"
8          "time"
9  )
10
11 func main() {
12
13         privateKey, _ := rsa.GenerateKey(rand.Reader, 1024) // o\
14 nly used for example purposes
15         publicKey := privateKey.PublicKey // more probably going\
16  to be pulled from JWKS, but this is for example purposes
17
18         // receive token at the API gateway, validate the signat\
19 ure
20         decodedToken, err := jwt.Parse(tokenString, func(token *\
21 jwt.Token) (interface{}, error) {
22                 if _, ok := token.Method.(*jwt.SigningMethodRSA); !ok {
23                         return nil, fmt.Errorf("unexpected signing method:
24 , token.Header["alg"])
25                 }
26                 return &publicKey, nil
```

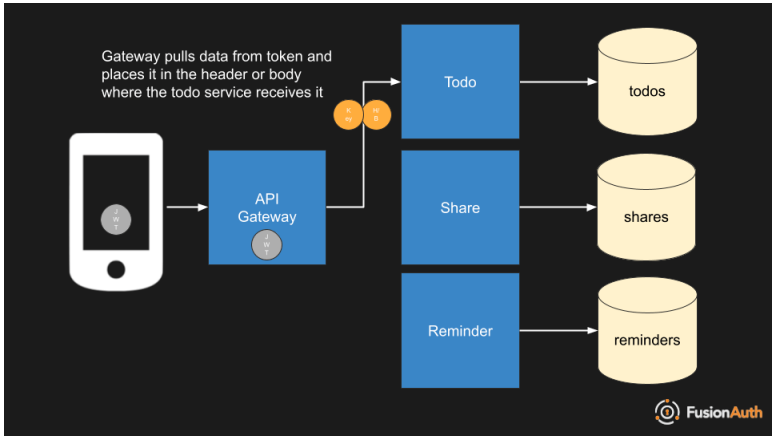
```
27          })
28
29          // validate the claims from decodedToken
30
31          if err != nil {
32                  fmt.Printf("Something Went Wrong: %s", err.Error())
33          }
34
35          // typically, you'll pull this from a secrets manager, n\
36  ot hard code it :)
37          var mySigningKey = []byte("hello gophers!!!")
38
39          hmacToken := jwt.New(jwt.SigningMethodHS256)
40          hmacClaims := hmacToken.Claims.(jwt.MapClaims)
41          for key, element := range decodedToken.Claims.(jwt.MapCl\
42  aims) {
43                  hmacClaims[key] = element
44          }
45
46          // modify claims here if needed
47          hmacClaims["exp"] = time.Now().Add(time.Second * 5).Unix\
48  ()
49
50          // sign JWT
51          hmacTokenString, err := hmacToken.SignedString(mySigning\
52  Key)
53
54          if err != nil {
55                  fmt.Println(fmt.Errorf("Something Went Wrong: %s", err.\
56  Error()))
57          }
58
59          // put hmacTokenString into the Authorization header for\
60   the future requests
61  }
```

You'll want to do key validation in the microservices, either with an ambassador container or in the code.

While this requires more custom code, the benefits of re-issuing the token mean that the microservices have less to deal with, the risk of a token being stolen have decreased, and you have more flexibility around performance and algorithm choice.

Full Extraction

Finally, the API gateway can extract the contents of the token entirely and turn it into a header or body parameter rather than re-issuing the token.



The gateway extracts needed data from the token as passes it as a header or form parameter.

This is helpful for situations where you are bolting on token based authentication, but the service expects values to be in normal HTTP headers or the body and it isn't worth it to upgrade it to process tokens. This could happen either right after the API gateway forwards the request or right before the container receives it.

In this case, make sure you don't just pass the extracted headers, but that you also pass an API key or other secret. That will allow the microservice to be sure that the request is coming from a valid source, the API gateway.

You might append the following values to the forwarded request using something like the ReverseProxy[36] or other proxy middleware.

```
http.Handle("/", &httputil.ReverseProxy{
    Director: func(r *http.Request) {
        r.URL.Scheme = "https"
        r.URL.Host = "microservice.url"
        r.Host = r.URL.Host
        r.Header.Set("X-API-key", "...") // known value t\
o assure the microservice of the request authenticity
        r.Header.Set("X-user-id", "...") // extracted fro\
m the token
        r.Header.Set("X-roles", "...") // extracted from \
the token
    },
})
```

The costs of this approach are that you have to build the token processor and extract out needed data. The benefits are that you can avoid modifying the microservice or using an ambassador container to proxy requests to it.

What about TLS?

Underlying each of these approaches is an understanding that traffic over the internal network (pod-to-pod) should use TLS unless there's a good reason not to. By using a service mesh, you

[36] https://pkg.go.dev/net/http/httputil#ReverseProxy

can avoid a nightmare of certificate renewals and transparently use TLS.

Using TLS between the different services lets you protect traffic in flight. With mutual TLS (client certificates) you can know that one service is allowed to call another, but you don't have the granularity that a token provides. As exhibited above, you can limit services or resources within a service to holders of tokens with certain claims, allowing for more fine grained security and control.

Conclusion

These four approaches to handling token based authentication allow you to protect your system components with tokens which are often in a format dictated by an external provider. The ability to pass through tokens, completely ignore them, re-issue them into a different algorithm, or extract the payload and transmute it to a different format, give you flexibility in meeting your security and performance needs.

Anatomy of a JWT

In this chapter, you'll learn more about what goes into a JSON Web Token (JWT) and how they are constructed.

JWTs are a commonly used portable token of identity. They have been standardized by the IETF and are created by many different identity providers and software libraries.

In Kubernetes, JWTs are often used to share identity information between an external user and an internal service, or between services.

Here's an example of a JWT, freshly minted:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCIsImtpZCI6ImY1ODg5MGQxO\
SJ9.eyJhdWQiOiI4NWEwMzg2Ny1kY2NmLTQ4ODItYWRkZS0xYTc5YWVlY\
zUwZGYiLCJleHAiOjE2NDQ4ODQxODUsImlhdCI6MTY0NDg4MDU4NSwiaX\
NzIjoiYWNtZS5jb20iLCJzdWIiOiIwMDAwMDAwMC0wMDAwLTAwMDAtMDA\
wMC0wMDAwMDAwMDEiLCJqdGkiOiIzZGQ2NDM0ZC03OWE5LTRkMTUt\
OThiNS03YjUxZGJiMmNkMzEiLCJhdXRoZW50aWNhdGlvbiR5cGUiOiJQQ\
VNTV09SRCIsImVtYWlsIjoiYWRtaW5AZnVzaW9uYXV0aC5pbyIsImVtYW\
lsX3ZlcmlmaWVkIjp0cnVlLCJhcHBsaWNhdGlvbklkIjoiODVhMDM4Njc\
tZGNjZi00ODgyLWFkZGUtMWE3OWFlZWM1MGRmIiwicm9sZXMiOlsiY2Vv\
Il19.dee-Ke6RzR0G9avaLNRZf1GUCDfe8Zbk9L2c7yaqKME
```

This may look like a lot of gibberish, but as you learn more about JWTs, it begins to make more sense.

There are a few types of JWTs, but I'll focus on signed JWTs as they are the most commonly used by systems like Kubernetes. A signed JWT may also be called a JWS and has three parts, separated by periods.

There's a header, which in the case of the JWT above, starts with `eyJhbGc`. Then there is a body or payload, which above starts with

`eyJhdWQ`. Finally, there is a signature, which starts with `dee-K` in the example JWT.

Let's break this example JWT apart and dig a bit deeper.

The header

`eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCIsImtpZCI6ImY1ODg5MGQxOSJ9` is the header of this JWT. The header contains metadata about a token, including the key identifier, what algorithm was used to sign in and other information.

If you run the above header through a base64 decoder:

```
1 echo 'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCIsImtpZCI6ImY1ODg\
2 5MGQxOSJ9'|base64 -d
```

You will see this JSON:

```
1 {"alg":"HS256","typ":"JWT","kid":"f58890d19"}%
```

`HS256` indicates that the JWT was signed with a symmetric algorithm, specifically HMAC using SHA-256.

The list of algorithms and implementation support level is below.

"alg" Param Value	Digital Signature or MAC Algorithm	Implementation Requirements
HS256	HMAC using SHA-256	Required
HS384	HMAC using SHA-384	Optional
HS512	HMAC using SHA-512	Optional
RS256	RSASSA-PKCS1-v1_5 using SHA-256	Recommended
RS384	RSASSA-PKCS1-v1_5 using SHA-384	Optional
RS512	RSASSA-PKCS1-v1_5 using SHA-512	Optional
ES256	ECDSA using P-256 and SHA-256	Recommended+
ES384	ECDSA using P-384 and SHA-384	Optional
ES512	ECDSA using P-521 and SHA-512	Optional
PS256	RSASSA-PSS using SHA-256 and MGF1 with SHA-256	Optional
PS384	RSASSA-PSS using SHA-384 and MGF1 with SHA-384	Optional
PS512	RSASSA-PSS using SHA-512 and MGF1 with SHA-512	Optional

"alg" Param Value	Digital Signature or MAC Algorithm	Implementation Requirements
none	No digital signature or MAC performed	Optional

This table is drawn from RFC 7518. As only HS256 is required to be compliant with the spec, consult the software or library used to create JWTs for details on supported algorithms.

Other metadata is also stored in this part of the token. The `typ` header indicates the type of the JWT. In this case, the value is `JWT`, but other values are valid. For instance, if the JWT conforms to RFC 9068, it may have the value `at+JWT` indicating it is an access token.

The `kid` value indicates what key was used to sign the JWT. For a symmetric key the `kid` could be used to look up a value in a secrets vault. For an asymmetric signing algorithm, this value lets the consumer of a JWT look up the correct public key corresponding to the private key which signed this JWT. Processing this value correctly is critical to signature verification and the integrity of the JWT payload.

Typically, you'll offload most of the processing of header values to a library. There are plenty of good open source JWT processing libraries. You should understand the values, but probably won't have to implement the actual processing.

The body

The payload, or body, is where things get interesting. This section contains the data that this JWT was created to transport. If the JWT, for instance, represents a user authorized to access certain data or functionality, the payload contains user data such as roles or other authorization info.

Here's the payload from the example JWT:

```
eyJhdWQiOiI4NWEwMzg2Ny1kY2NmLTQ4ODItYWRkZS0xYTc5YWVlYzUwZ\
GYiLCJleHAiOjE2NDQ4ODQxODUsImlhdCI6MTY0NDg4MDU4NSwiaXNzIj\
oiYWNtZS5jb20iLCJzdWIiOiIwMDAwMDAwMC0wMDAwLTAwMDAtMDAwMC0\
wMDAwMDAwMDAwMDEiLCJqdGkiOiIzZGQ2NDM0ZC03OWE5LTRkMTUtOThi\
NS03YjUxZGJiMmNkMzEiLCJhdXRoZW50aWNhdGlvblR5cGUiOiJQQVNTV\
09SRCIsImVtYWlsIjoiYWRtaW5AZnVzaW9uYXV0aC5pbyIsImVtYWlsX3\
ZlcmlmaWVkIjp0cnVlLCJhcHBsaWNhdGlvbklkIjoiODVhMDM4NjctZGN\
jZi00ODgyLWFkZGUtMWE3OWFlZWM1MGRmIiwicm9sZXMiOlsiY2VvIl19
```

If you run the sample payload through a base64 decoder:

```
echo 'eyJhdWQiOiI4NWEwMzg2Ny1kY2NmLTQ4ODItYWRkZS0xYTc5YWV\
lYzUwZGYiLCJleHAiOjE2NDQ4ODQxODUsImlhdCI6MTY0NDg4MDU4NSwi\
aXNzIjoiYWNtZS5jb20iLCJzdWIiOiIwMDAwMDAwMC0wMDAwLTAwMDAtM\
DAwMC0wMDAwMDAwMDEiLCJqdGkiOiIzZGQ2NDM0ZC03OWE5LTRkMT\
UtOThiNS03YjUxZGJiMmNkMzEiLCJhdXRoZW50aWNhdGlvblR5cGUiOiJ\
QQVNTV09SRCIsImVtYWlsIjoiYWRtaW5AZnVzaW9uYXV0aC5pbyIsImVt\
YWlsX3ZlcmlmaWVkIjp0cnVlLCJhcHBsaWNhdGlvbklkIjoiODVhMDM4N\
jctZGNjZi00ODgyLWFkZGUtMWE3OWFlZWM1MGRmIiwicm9sZXMiOlsiY2\
VvIl19' |base64 -d
```

You'll see this JSON:

```
{
  "aud": "85a03867-dccf-4882-adde-1a79aeec50df",
  "exp": 1644884185,
  "iat": 1644880585,
  "iss": "acme.com",
  "sub": "00000000-0000-0000-0000-000000000001",
  "jti": "3dd6434d-79a9-4d15-98b5-7b51dbb2cd31",
  "authenticationType": "PASSWORD",
  "email": "admin@fusionauth.io",
  "email_verified": true,
```

```
  "applicationId": "85a03867-dccf-4882-adde-1a79aeec50df",
  "roles": [
    "ceo"
  ]
}
```

Note that the algorithm to create signed JWTs can remove base64 padding, so there may be missing = signs at the end of the JWT. You may need to add that back in order to decode a JWT. This depends on the length of the content. You can learn more about it here[37].

As mentioned above, the payload is what your application or microservice cares about, so let's take a look at this JSON more closely. Each of the keys of the object are called "claims".

Some claims are well known with meanings dictated by standards bodies such as the IETF. You can view examples of such claims here[38]. These include the `iss` and `aud` claims from the example token. Both of these have defined meanings when present in the payload of a JWT.

There are other non-standard claims, such as `authenticationType`. These claims may represent business domain or custom data. For example, `authenticationType` is a proprietary claim used by FusionAuth to indicate the method of authentication, such as password, refresh token or via a passwordless link.

You may add any claims you want to a JWT, including data useful to downstream consumers of the JWT. As you can see from the `roles` claim, claims don't have to be simple JSON primitives. They can be any data structure which can be represented in JSON.

[37] https://datatracker.ietf.org/doc/html/rfc7515#appendix-C
[38] https://www.iana.org/assignments/jwt/jwt.xhtml

Claims to verify

When code is presented with a JWT, it should verify certain claims in the payload and header.

Header claims like `kid` are typically verified by the signature verification step, which is usually done by a library.

At a minimum, these claims in the payloard should be verified:

- `iss` identifies the issuer of the JWT. It doesn't matter exactly what this string is (UUID, domain name, URL or something else) as long as the issuer and consumer of the JWT agree on valid values, and that the consumer validates the claim matches a known good value.
- `aud` identifies the audience of the token, that is, who should be consuming it. `aud` may be a scalar or an array value. Again, the issuer and the consumer of the JWT should agree on the specific values considered acceptable.
- `nbf` and `exp`. These claims determine the timeframe for which the token is valid. The `nbf` claim can be useful if you are issuing a token for future use. The `exp` claim, a time beyond which the JWT is no longer valid, should always be set. Unlike other claims, these have a defined value format: seconds since the unix epoch.

Checking can be performed by a microservice, an ambassador container, or a service mesh.

In addition to these, verify business domain specific claims. For instance, someone consuming the above JWT could deny access when `authenticationType` is an unknown value.

Avoid putting unused claims into a JWT. While there is no limit to the size of a JWT, in general the larger they are, the more CPU is required to sign and verify them and the more time it takes to transport them. Benchmark expected JWTs to have an understanding of the performance characteristics.

Claims and security

The claims of a signed JWT are visible to anyone who possesses the token. This is typically mitigated by sending all traffic over TLS, but there are additional steps to take.

All you need to view the claims in plaintext is a base64 decoder, which is available at every command line and everywhere in the internet. Do not put anything that should remain secret into a JWT.

This includes:

- private information such as government Ids
- secrets like passwords
- anything that would leak information like an integer Id

Another security concern is related to the verification of the `aud` claim.

Since consuming code already possesses the token, isn't verifying the `aud` claim extra work? The `aud` claim indicates who should receive this JWT, but the code already has it. Nope, always verify this claim.

Why?

Imagine a scenario where you have two different services. One is to create and manage todos and the other is a billing service, used to transfer money. Both expect some users to have a role of `admin`. However, that role means vastly different things in functionality terms.

If neither the todo and billing services verify any given JWT was created for them, an attacker could take a JWT from the todo service with the `admin` role and present it to the billing service.

This would be at best a bug and at worst an escalation of privilege with negative ramifications for bank accounts.

Signature

The signature of a JWT is critical, because it guarantees the integrity of the payload and the header. Verifying the signature must be the first step that any consumer of a JWT performs. If the signature doesn't match, no further processing should take place.

While you can read the relevant portion of the specification[39] to learn how the signature is generated, the high level overview is:

- the header is turned into a base64 URL encoded string
- the payload is turned into a base64 URL encoded string
- they are concatenated with a `.`
- the resulting string is run through the cryptographic algorithm selected, along with the corresponding key
- the signature is base64 URL encoded
- the encoded signature is appended to the string with a `.` as a separator

When the JWT is received, the same operations can be performed. If the generated signature is correct, the contents of the JWT are unchanged from when it was created.

Symmetric vs Asymmetric Signing Algorithms

Above you saw that there are a variety of algorithms you might expect to see. How can you choose?

Symmetric algorithms are best used inside a trust boundary, because they require a shared secret. If you are generating JWTs to authenticate microservices internally, a symmetric algorithm like HMAC will be quick and easy to understand.

[39] https://datatracker.ietf.org/doc/html/rfc7515#page-15

Asymmetric algorithms are best used between trust boundaries, because they require no shared secret. Instead, you are using public/private key algorithms such as RSA or ECC. In this case, the verifier must have access to the public key in order to confirm the integrity of the signed JWT. This imposes either deployment requirements or network access requirements.

You may reissue tokens at trust boundaries. This is described in a previous chapter.

Limits

In the specifications, there are no hard limits on length of JWTs. In practical terms, think about:

- Where are you going to store the JWT
- What is the performance penalty of large JWTs

Storage

JWTs can be sent in HTTP headers, typically the `Authorization` header, or placed in the HTTP body. In these scenarios, the storage dictates the maximum allowed JWT length.

For example, the limit on HTTP headers varies widely based on software components, but 8192 bytes seems to be a common value.

Consult the relevant specifications or other resources for limits in your particular use case, but rest assured that JWTs have no intrinsic size limits.

Performance penalty

Since JWTs can contain many different kinds of user information, developers may be tempted to put too much in them. This can

degrade performance, both in the signing and verification steps as well as in transport.

For an example of the former, here are the results of a benchmark from signing and verifying two different JWTs. Each operation was done 50,000 times.

This first JWT had a body approximately 180 characters in length; the total encoded token length was between 300 and 600, depending on the signing algorithm used.

```
hmac sign
  1.632396   0.011794   1.644190 (  1.656177)
hmac verify
  2.452983   0.015723   2.468706 (  2.487930)
rsa sign
 28.409793   0.117695  28.527488 ( 28.697615)
rsa verify
  3.086154   0.011869   3.098023 (  3.109780)
ecc sign
  4.248960   0.017153   4.266113 (  4.285231)
ecc verify
  7.057758   0.027116   7.084874 (  7.113594)
```

The next JWT payload was of approximately 1800 characters, so ten times the size of the previous token. This had a total token length of 2400 to 2700 characters.

```
hmac sign
  3.356960   0.018175   3.375135 (  3.389963)
hmac verify
  4.283810   0.018320   4.302130 (  4.321095)
rsa sign
 32.703723   0.172346  32.876069 ( 33.072665)
rsa verify
  5.300321   0.027455   5.327776 (  5.358079)
ecc sign
  6.557596   0.032239   6.589835 (  6.624320)
ecc verify
  9.184033   0.035617   9.219650 (  9.259225)
```

You can see that the total time increased for the longer JWT, but typically not linearly. The increase in time taken ranges from about 20% for RSA signing to approximately 100% for HMAC signing.

Be mindful of additional time taken to transport longer JWT; this can be tested and optimized in the same way you would with any other API or HTML content.

Summary

Signed JWTs have a header, body, and signature. Each plays a role in delivering auth information to components which require it.

When using JWTs to secure applications running in Kubernetes, understanding all three of these components are critical.

Conclusion

This book has covered a fair bit of ground on the topic authentication, authorization and Kubernetes.

I hope you enjoyed learning more about how you can safely authenticate and authorize access to Kubernetes resources, both those internal to clusters as well as applications running on top of them.

You also learned about JSON web tokens and how token based auth can be used within microservices.

At the end of the day, Kubernetes is an extremely powerful system with a lot of functionality included. However, authorization and authentication are not.

While service meshes and API gateways offer some functionality, it's important to understand your options as you build your system on top of the orchestration framework.