Distributed Authentication Mesh*

Declarative Adhoc Conversion of Credentials

Christoph Bühler

Spring Semester 2021
University of Applied Science of Eastern Switzerland (OST)

*I would like to thank Dr. Olaf Zimmermann and Florian Forster. TODO.

TODO, this will contain the abstract of the project report.

Contents

List of Tables

List of Figures

1 Introduction

Modern cloud environments solve many problems like the discovery of services and data transfer or communiation between services in general. One modern way of solving service discovery and communication is a Service Mesh, which introduces an additional infrastructure layer that manages the communication between services (Li et al. 2019, sec. 2).

However, a specific problem is not solved yet: "dynamic" trusted communication between services. When a service, that is capable of handling OpenID Connect (OIDC) credentials, wants to communicate with a service that only knows Basic Authentication that originating service must implement some sort of conversion or know static credentials to communicate with the basic auth service. Generally, this introduces changes to the software of services. In small applications which consist of one or two services, implementing this conversion may be a feasable option. If we look at an application which spans over a big landscape and a multitude of services, implementing each and every possible authentication mechanism and the according conversions will be error prone work and does not scale well[1].

The goal of the project "Distributed Authentication Mesh" is to provide a solution for this problem.

TODO.

[1] According to the matrix problem: X services $* Y$ authentication methods

2 Definitions and Boundaries

This section provides general information about the project, the context and prerequisite knowledge. It gives an overview of the context as well as terminology and general definitions.

2.1 Context

This project aims at the specific problem of declarative conversion of credentials to ensure authorized communication between services. The solution may be runnable on various platforms but will be implemented according to Kubernetes standards. Kubernetes[1] is an orchestration platform that works with containerized applications. The solution introduces an operator pattern, as explained in Section 2.2.3

The deliverables of this project may aid services to communicate with each other despite different authentication mechanisms. As an example, this could be used to enable a modern web application that uses OpenID Connect (OIDC) as the authentication and authorization mechanism to communicate with a legacy application that was deployed on the Kubernetes cluster but not yet rewritten. This transformation of credentials (from OIDC to Basic Auth) is done by the solution of this project instead of manual work which may introduc code changes to either service.

To use the proposed solution of this project, no service mesh or other complex layer is needed. The solution runs without those additional parts on a Kubernetes cluster. To provide service discovery, the default internal DNS capabilities of Kubernetes are sufficient.

2.2 Kubernetes

2.2.1 What is Kubernetes

Kubernetes is an open source platform that manages containerized workloads and applications. Workloads may be accessed via “Services” that use a DNS naming system. Kubernetes uses declarative definitions to compare the actual state of the system with the expected state (CNCF 2021).

[1] https://kubernetes.io/

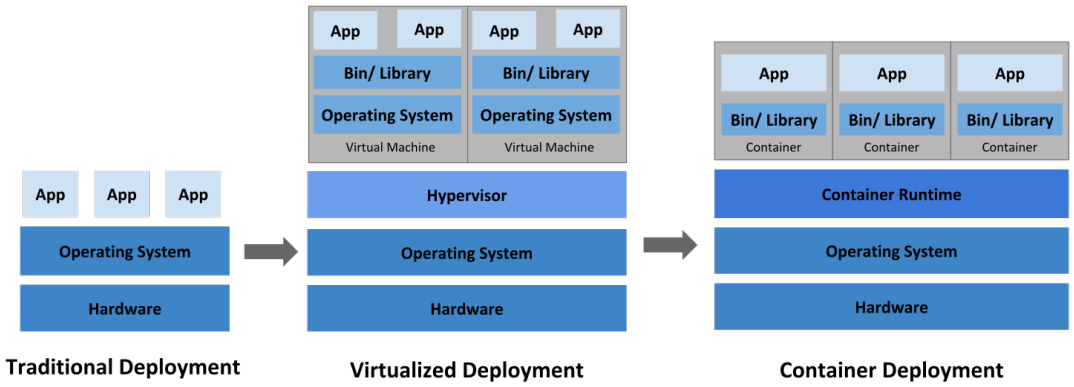

Figure 2.1: Container and Deployment Evolution. Description of the evolution of deployments as found on the documentation website of Kubernetes (CNCF 2021). This image is licensed under the CC BY 4.0 license (Creative Commons 2021).

According to the Kubernetes team, the way of deploying applications has evolved. As shown in Figure 2.1, the "Traditional Era" was the time, when applications were deployed via FTP access and started manually (e.g. on an Apache webserver). Then the revolution to virtual machines came and technologies that could virtualize a whole operating system, such as VMWare, were born. The latest stage, "Container Era," defines a new way deploying workloads by virtualizing processes instead of operating systems and therefore better use the given resources (CNCF 2021).

Kubernetes is a major player in "Container Deployment" as seen in Figure 2.1 and supports teams with the following features according to the documentation (CNCF 2021):

- **Service discovery and load balancing**: Use DNS names or IP addresses to route traffic to a container and if the traffic is high and multiple instances are available, Kubernetes does load balance the traffic
- **Storage orchestration**: Automatically provide storage in the form of mountable volumes
- **Automated rollouts and rollbacks**: When a new desired state is provided Kubernetes tries to achieve the state at a controlled rate and has the possibility of performing rollbacks
- **Automatec bin packing**: Kubernetes only needs to know how much CPU and RAM a workload needs and then takes care of placing the workload on a fitting node in the cluster
- **Self-healing**: If workloads are failing, Kubernetes tries to restart the applications and even kills services that do not respond to the configured health checks
- **Secret and configuration management**: Kubernetes has a store for sensitive data as well as configurational data that may change the behaviour of a workload

2.2.2 Terminology

Find the common Kubernetes terminology attached in Table 2.1. The table provides a list of terms that will be used to explain concepts like the operator pattern in Section 2.2.3.

Table 2.1: Common Kubernetes Terminology

Term	Description
Container	Smallest possible unit in a deployment. Contains the definition of the workload. A container consists of a container image, arguments, volumes and other specific information to carry out a task.
Pod	Composed of multiple containers. Is ran by kubernetes as an instance of a deployment. Pods may be scaled according to definitions or "pod scalers."
Deployment	
Service	
Resource	
CRD	
Operator	
Watcher	
Validator	
Mutator	
Data Plane	
Control Plane	

2.2.3 Operator

An operator in Kubernetes is an extension to the Kubernetes API itself. A custom operator typically manages the whole lifecycle of an appliction it manages (Dobies and Wood 2020). Such a custom operator can further be used to reconcile normal Kubernetes resources or any combination thereof.

Some examples of application operators are:

- Prometheus Operator[2]: Manages instances of Prometheus in a cluster
- Postgres Operator[3]: Manages PostgreSQL clusters inside Kubernetes, with the support of multiple instance database clusters

There exists a broad list of operators, which can be (partially) viewed on operatorhub.io.

[2]https://github.com/prometheus-operator/prometheus-operator
[3]https://github.com/zalando/postgres-operator

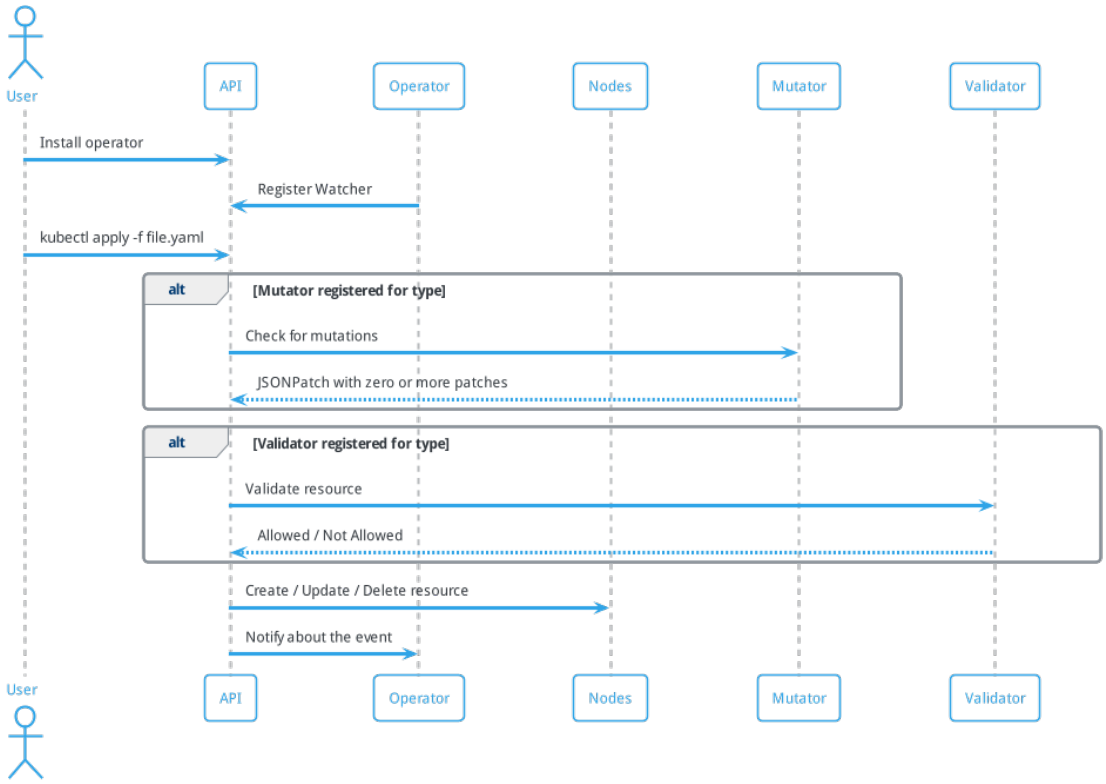

Figure 2.2: Kubernetes Operator Workflow

Figure 2.2 shows the general workflow of an event that is managed by an operator. When an operator is installed and running on a Kubernetes cluster, it registers "Resource Watchers" with the API and receives notifications when the master node modifies resources a watched resource. The overviewed events are "Added," "Modified" and "Deleted." There are two additional events that may be returned by the API ("Error" and "Bookmark") but they are typically not needed in an operator.

When the user interacts with the Kubernetes API (for example via the `kubectl` executable) and creates a new instance of a resource, the API will first call any "Mutator" in a serial manner. After the mutators, the API will call any "Validators" in parallel and if no validator objects against the creation, the API will then store the resource and tries to apply the transition for the new desired state. Now, the operator receives the notification about the watched resource and may interact with the event. Such an action may include to update resources, create more resources or even delete other instances.

2.2.4 Sidecar

The sidecar pattern is the most common pattern for multi-container deployments. Sidecars are containers that enhance the functionality of the main container in a pod. An example for such a sidecar is a log collector, that collects log files written to the file system and forwards them to some log processing software (Burns and Oppenheimer 2016, sec. 4.1). Another example is the Google CloudSQL Proxy[4], which provides access to a CloudSQL instance from a pod without routing the whole traffic through Kubernetes services.

The example shown in Figure 2.3 is extensible. Such sidecars may be injected by a mutator or an operator to extend functionality.

2.2.5 Service Mesh

A "Service Mesh" is a dedicated infrastructure layer that handles intercommunication between services. It is responsible for the delivery of requests in a modern cloud application (Li et al. 2019, sec. 2). An example from the practice is "Istio"[5]. When using Istio, the applications do not need to know if there is a service mesh installed or not. Istio will inject a sidecar (see Section 2.2.4) into pods and handle the communication with the injected services.

The service mesh provides a set of features (Li et al. 2019, sec. 2):

- **Service discovery**: The mechanism to locate and communicate with a workload / service. In a cloud environment, the location of services will likely change, thus the service mesh provides a way to access the services in the cloud.

[4] https://github.com/GoogleCloudPlatform/cloudsql-proxy
[5] https://istio.io/

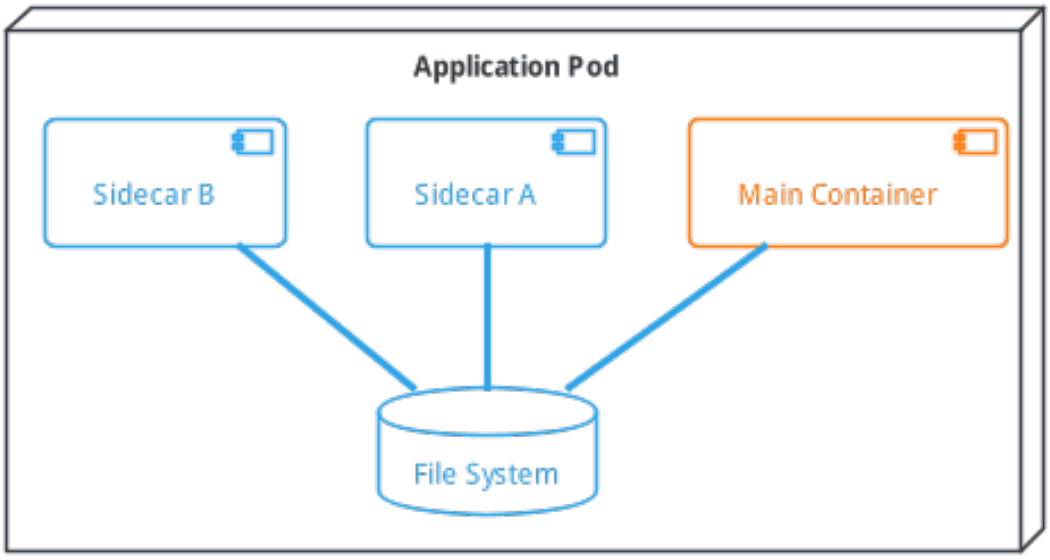

Figure 2.3: Sidecar container extending a main container in a pod. As example, this could be a log collector (Burns and Oppenheimer 2016, fig. 1).

- **Load balancing**: As an addition to the service discovery, the mesh provides load balancing mechanisms as is done by Kubernetes itself.
- **Fault tolerance**: The router in a service mesh is responsible to route traffic to healthy services. If a service is unavailable or even reports a crash, traffic should not be routed to this instance.
- **Traffic monitoring**: In contrast to the default Kubernetes possibilities, with a service mesh, the traffic from and to various services can be monitored in detail. This offers the opportunity to derive reports per target, success rates and other metrics.
- **Circuit breaking**: The ability to cut off an overloaded service and back off the remaining requests instead of totally failing the service under stress. A circuit breaker pattern measures the failure rate of a service and applies states to the service: “Closed” - requests are passed to the service, “Open” - requests are not passed to this instance, “Half-Open” - only a limited number is passed (Montesi and Weber 2016, sec. 2).
- **Authentication and access control**: Through the control plane, a service mesh may define the rules of communication. It defines which services can communicate with one another.

As observed in the list above, many of the features of a service mesh are already provided by Kubernetes. Service discovery, load balancing, fault tolerance and - though limited - traffic monitoring is already possible with Kubernetes. Introducing a service mesh into a cluster enables administrators to build more complex scenarios and deployments.

2.3 Authentication and Authorization

2.3.1 Basic

briefly describe basic auth.

2.3.2 OpenID Connect (OIDC)

briefly describe OIDC

3 State of the Art, the Practice and Deficiencies

This section gives an overview over the current state of the art as well as the deficiencies according to the author. Following the description of the current situation, a definition of the should situation gives an overview of the purposed solution.

- Describe the IS situation.
- Describe the SHOULD situation.
- describe service mesh (image from referecne could be used)

Test:

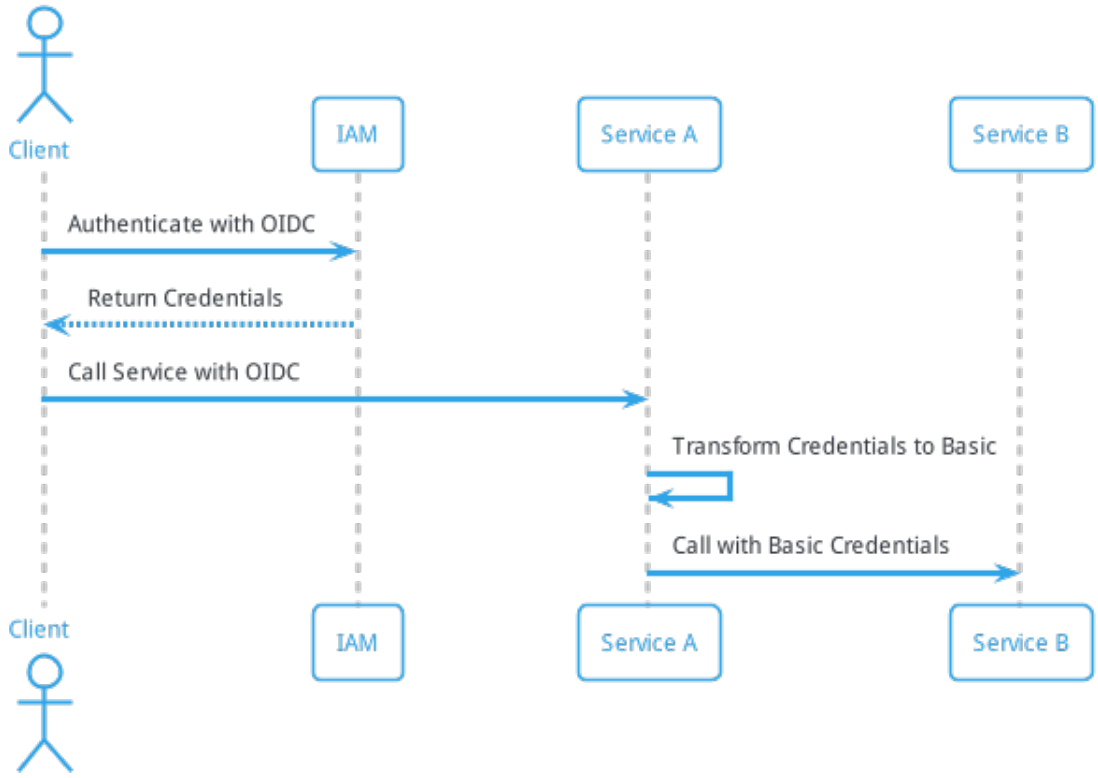

Figure 3.1: Diagram Test

This Figure 3.1 shows a test.

4 Grober Roter Faden Projektbericht

1. Introduction
    - Das Erreichte und den Überblick der Arbeit wiedergeben
    - Erklären wie die Arbeit aufgebaut ist und welche Details wo zu finden sind
2. Einführung ins Thema
    - Leser ausbilden
    - Backgroundinformationen liefern
    - Context der Arbeit
    - Begrifflichkeiten erklären
3. IST / SOLL Beschreibung
    - Was gibt es
    - Wie arbeit die Leute
    - Wo sind die Defizite
4. Wie lösen wir die Probleme
    - Planung der Software (Diagramme etc.)
    - Wie die Lösung aussieht
    - Umsetzung der Konzepte
5. Nachweis
    - Beweis anführen, dass das vorgeschlagene Konzept funktioniert
    - Beispiele nutzen um Leute abzuholen
6. Conclusion
    - Ausblick (Referenz auf weitere Projektarbeit)
    - Was man gemacht hat (klassischer Paperdiamant)

5 Todos

Todos:

- nope.

Bibliography

Burns, Brendan, and David Oppenheimer. 2016. “Design Patterns for Container-Based Distributed Systems.” In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. Denver, CO: USENIX Association. https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/burns.

CNCF. 2021. “Kubernetes Website.” *GitHub Repository*. https://github.com/kubernetes/website; GitHub.

Creative Commons. 2021. “Attribution 4.0 International (CC BY 4.0).” https://creativecommons.org/licenses/by/4.0/.

Dobies, Jason, and Joshua Wood. 2020. *Kubernetes Operators: Automating the Container Orchestration Platform*. " O’Reilly Media, Inc.".

Li, W., Y. Lemieux, J. Gao, Z. Zhao, and Y. Han. 2019. “Service Mesh: Challenges, State of the Art, and Future Research Opportunities.” In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, 122–25. https://doi.org/10.1109/SOSE.2019.00026.

Montesi, Fabrizio, and Janine Weber. 2016. “Circuit Breakers, Discovery, and API Gateways in Microservices.” *CoRR* abs/1609.05830. http://arxiv.org/abs/1609.05830.