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 and further projects 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 the projects instead of manual work which may introduc code changes to either service.

This specific project provides a proof of concept (PoC) with an initial version on a GitHub repository. The PoC demonstrates that it is possible to instruct an Envoy[2] proxy to communicate with an injected service to modify authentication credentials in-flight.

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.

[1]https://kubernetes.io/

[2]https://www.envoyproxy.io/

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).

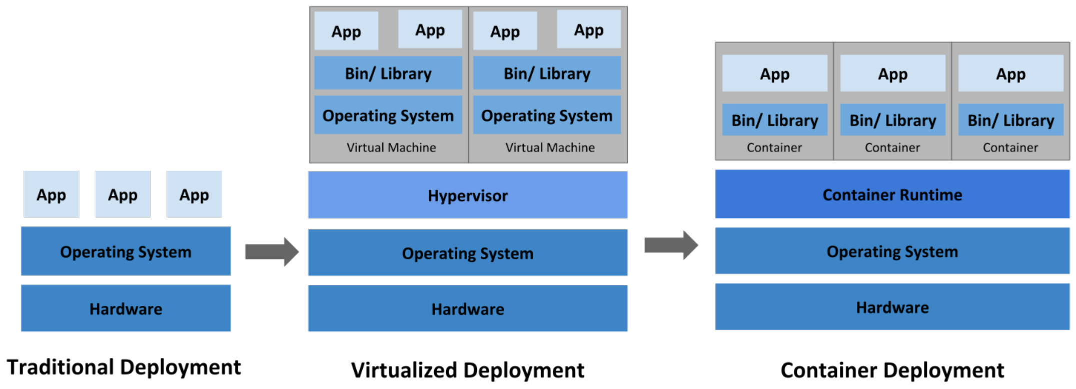

Figure 2.1: Container and Deployment Evolution. Description of the evolution of deployments as found on the documentation website of Kubernetes (CNCF 2021). This image is licensed under the CC BY 4.0 license (Creative Commons 2021).

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

The list of features is not complete. There are many concepts in Kubernetes which help to build complex deployment scenarios and enable teams to ship their applications in an agile manner.

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." Highly coupled tasks are deployed together in a pod (i.e. multiple coupled containers in a pod).
Deployment	A deployment is a managed instance of a pod. Kubernetes will run the described pod with the desired replica count on the best possible worker node. Deployments may be scaled with auto-scaling mechanisms.
Service	A service enables communciation with one or multiple pods. The service contains a selector that points to a certain number of pods and then ensures that the pods are accessable via a DNS name. The name is typically a combination of the servicename and the namespace (e.g. `my-service.namespace`).

Term	Description
Resource	A resource is something that can be managed by Kubernetes. It defines an API endpoint on the master node and allows Kubernetes to store a collection of such API objects. Examples are: `Deployment`, `Service` and `Pod`, to name a few of the built-in resources.
CRD	A Custom Resource Definition (CRD) enables developers to extend the default Kubernetes API. With a CRD, it is possible to create own resources which creates an API endpoint on the Kubernetes API. An example of such a CRD is the `Mapping` resource of Ambassador[3].
Operator	An operator is a software that manages Kubernetes resources and their lifecycle. Operators may use CRDs to define custom objects on which they react when some event (`Added`, `Modified` or `Deleted`) triggers on a resource. For a more in-depth description, see Section 2.2.3.
Watcher	A watcher is a constant connection from a client to the Kubernetes API. The watcher defines some search and filter parameters and receives events for the found resources.
Validator	A validator is a service that may reject the creation, modification or deletion of resources.
Mutator	Mutators are called before Kubernetes validates and stores a resource. Mutators may return JSON patches (Bryan and Nottingham 2013) to instruct Kubernetes to modify a resource prior to validating and storing them.

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[4]: Manages instances of Prometheus in a cluster
- Postgres Operator[5]: Manages PostgreSQL clusters inside Kubernetes, with the support of multiple instance database clusters

[3]https://www.getambassador.io/
[4]https://github.com/prometheus-operator/prometheus-operator
[5]https://github.com/zalando/postgres-operator

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

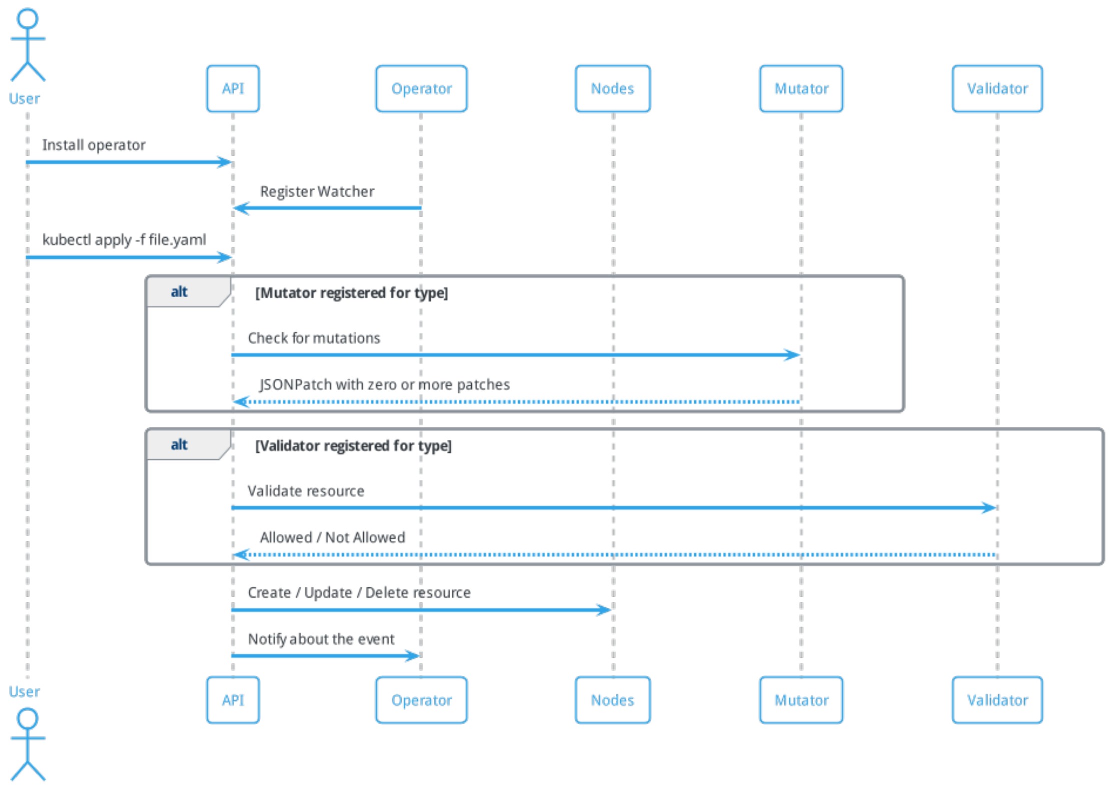

Figure 2.2: Kubernetes Operator Workflow

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 towards some log processing software (Burns and Oppenheimer 2016, sec. 4.1). Another example is the Google CloudSQL Proxy[6], which provides access to a CloudSQL instance from a pod without routing the whole traffic through Kubernetes services.

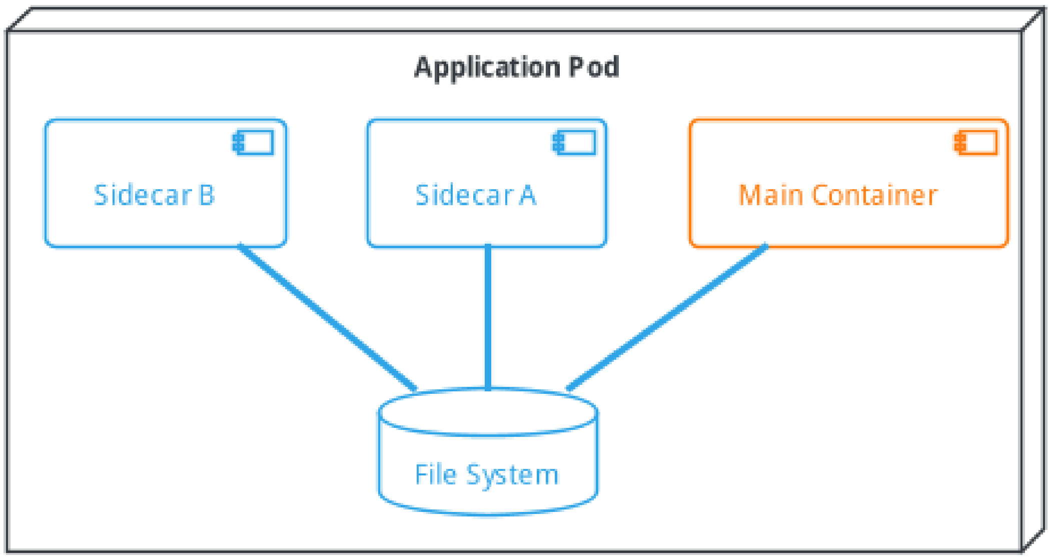

Figure 2.3: Sidecar container extending a main container in a pod. As example, this could be a log collector (Burns and Oppenheimer 2016, fig. 1).

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”[7]. 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.

[6]https://github.com/GoogleCloudPlatform/cloudsql-proxy
[7]https://istio.io/

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.
- **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

The `Basic` authentication scheme is a trivial authentication that accepts a username and a password encoded in Base64. To transmit the credentials, a construct with the schematics of `<username>:<password>` is created and inserted into the http request as the `Authorization` header with the prefix `Basic` (Reschke 2015, sec. 2). An example with the username `ChristophBuehler` and password `SuperSecure` would result in the following header: `Authorization: Basic Q2hyaXN0b3BoQnVlaGxlcjpTdXBlclNlY3VyZQ==`.

2.3.2 OpenID Connect (OIDC)

OpenID Connect is an authenticating mechanism, that builds upon the `OAuth 2.0` authorization protocol. OAuth 2.0 deals with authorization only and grants access to data and features on a specific application. OAuth by itself does not define *how* the credentials are transmitted and exchanged (Hardt and others 2012). OIDC adds a layer on top of OAuth 2.0 that defines *how* these credentials must be exchanged. This adds login and profile capabilities to any application that uses OIDC (Sakimura et al. 2014).

When a user wants to authenticate himself with OIDC, one of the possible “flows” is the “Authorization Code Flow” (Sakimura et al. 2014, sec. 3.1). Other possible flows are the “Implicit Flow” (Sakimura et al. 2014, sec. 3.2) and the “Hybrid Flow” (Sakimura et al. 2014, sec. 3.3). In Figure 2.4, the “Authorization Code Flow” is depicted. A user that wants to access a certain resource on a relying party (i.e. something that relies on the information about the user) and is not authenticated and authorized, the relying party forwards the user to the identity provider (IdP). The user provides his credentials to the IdP and is returned to the relying party with an authorization code. The relying party can then exchange the authorization code to valid tokens on the token endpoint of the IdP. Typically, `access_token` and `id_token` are provided. While the `id_token` must be a JSON Web Token (JWT) (Sakimura et al. 2014, sec. 2), the `access_token` can be in any format (Sakimura et al. 2014, sec. 3.3.3.8).

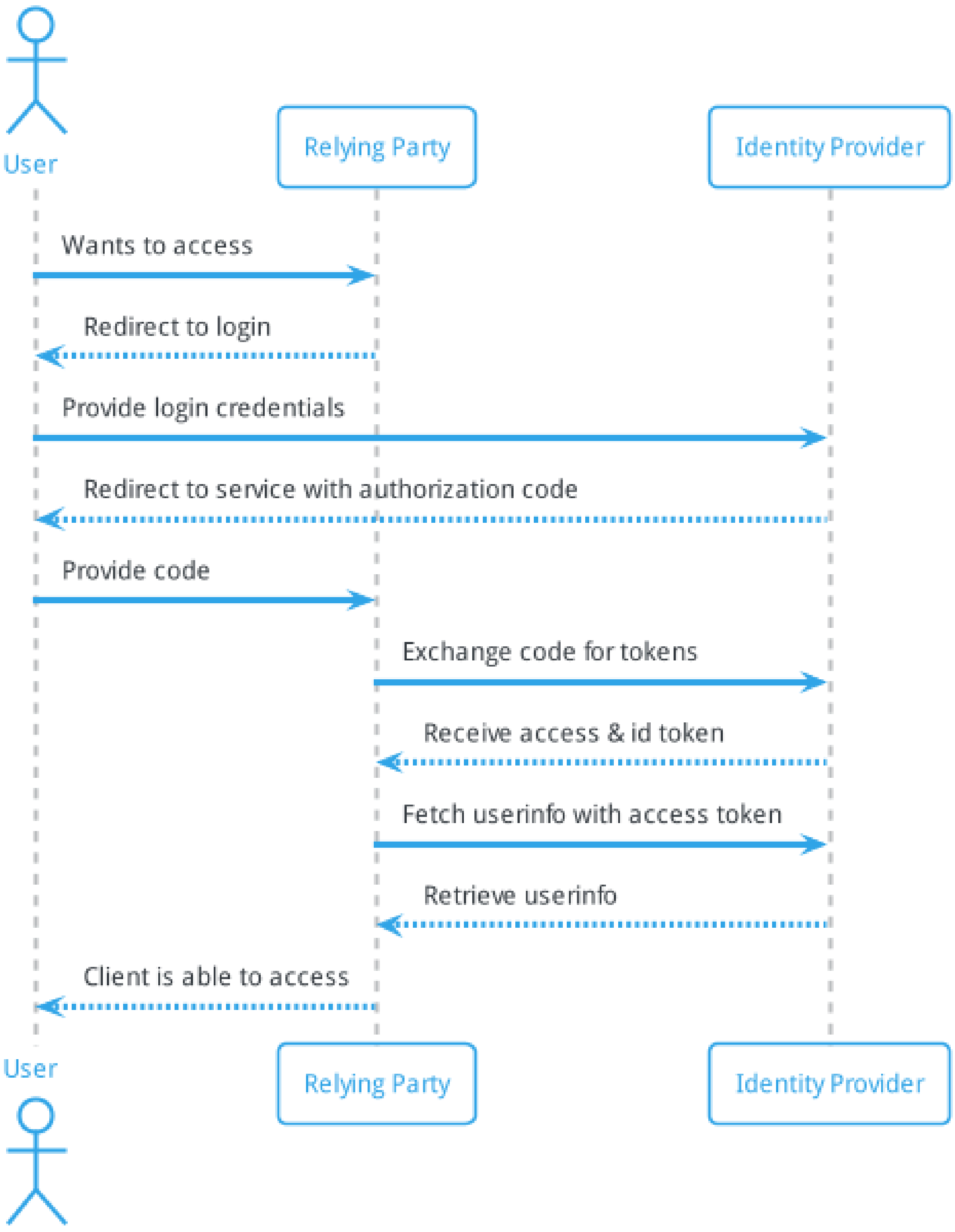

Figure 2.4: OIDC code authorization flow (Sakimura et al. 2014).

3 State of the Art, the Practice and Deficiencies

This section gives an overview over the current state of the art, the practice, as well as the deficiencies to an optimal situation. Furthermore this section states an assessment of the current practice and the solutions found.

3.1 State of the Art

In cloud environments, a problem that is well solved is the transmission of data from one point to another. Kubernetes, for example, uses “Services” that provide a DNS name for a specified workload. In terms of authentication and authorization, there exist a variety of schemes that enable an application to authenticate and authorize their users. OpenID Connect (OIDC) (see Section 2.3.2) is a modern authentication scheme, that builds upon OAuth 2.0, that in turn handles authorization (Sakimura et al. 2014).

Modern software architectures that are specifically designed for the cloud are called “Cloud Native Applications” (CNA). Kratzke and Peinl (2016) define a CNA as:

> “A cloud-native application is a distributed, elastic and horizontal scalable system composed of (micro)services which isolates state in a minimum of stateful components. The application and each self-contained deployment unit of that application is designed according to cloud-focused design patterns and operated on a self-service elastic platform.” (Kratzke and Peinl 2016, sec. 3).

However, with CNAs and the general movement to cloud environments, not all applications get that chance to adjust. For various reasons like budget, time or complexity, legacy applications and monoliths are not refactored or re-written before they are deployed into a cloud environment. If the legacy applications are mixed with modern systems, then the need of “translation” arises. Assuming, that the modern part is a secure application, that uses OIDC to authorize its users and the application needs to fetch data from the legacy system that does not understand OIDC, code changes must be made. Following the previous assumption, the code changes will likely be introduced into the modern application, since it is better maintainable and deployable than the legacy monolith. Hence, the modern application receives changes that may introduce new bugs or security vulnerabilities.

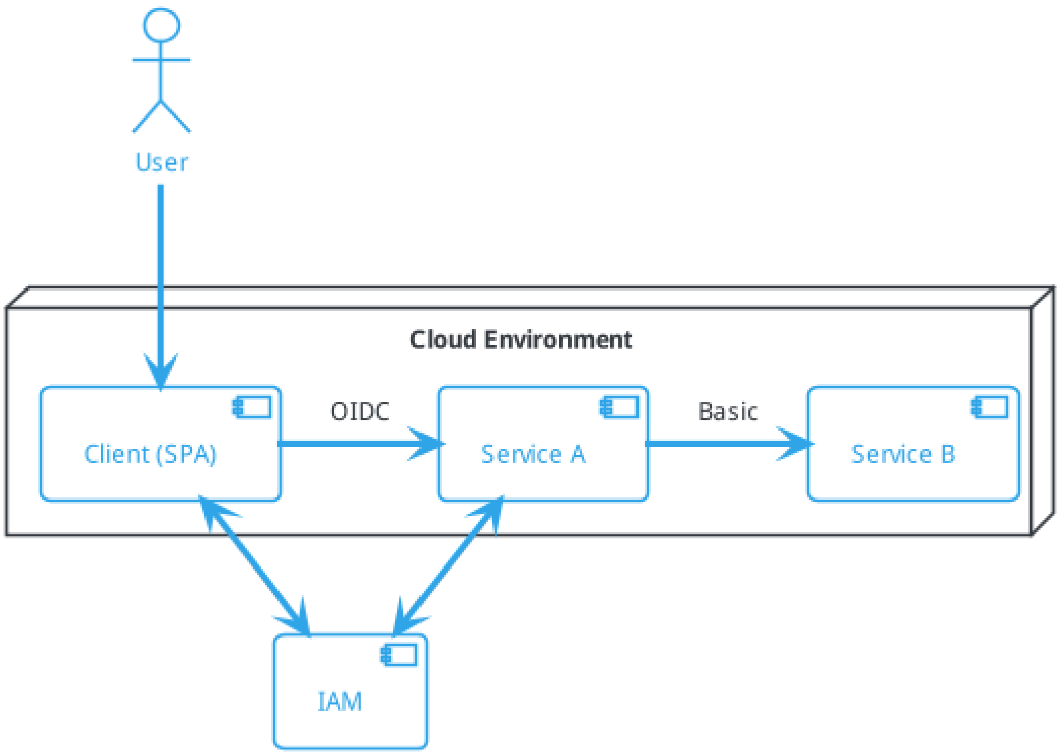

Figure 3.1: Microservice Architecture that contains modern applications as well as legacy services.

We consider the components in Figure 3.1:

- **User**: A person with access to the application
- **Client**: A modern single page application (SPA)
- **IAM**: Identity Provider for the solution (does not necessarily reside in the same cloud)
- **Service A**: A modern API application and primary access point for the client
- **Service B**: Legacy service that is called by service a to fetch some additional data

The stated scenario is quite common. Legacy services may not be the primary use-case, but there exist other reasons to have the need of using a translation of credentials. Another case is the usage of third party applications which only support certain authentication mechanisms.

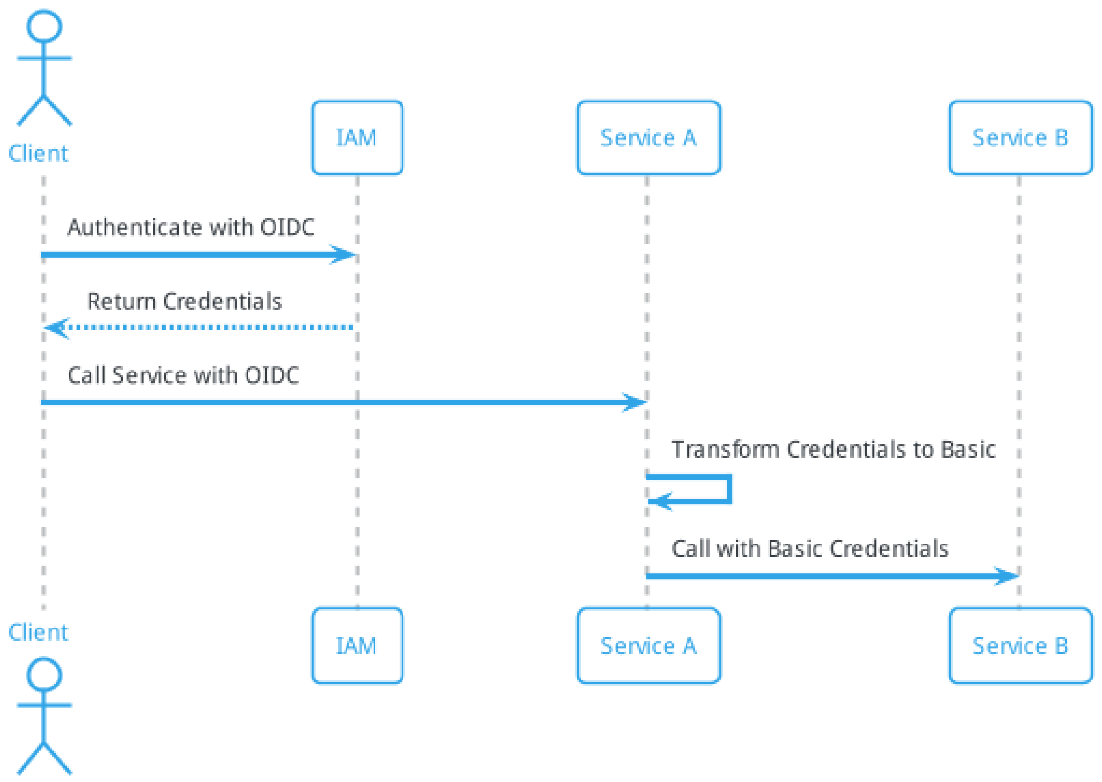

Figure 3.2: Current state of the art of accessing legacy systems from modern services with differing authentication schemes.

The process in Figure 3.2 shows the process of communication in such a described scenario. In Figure 3.2, the “Client” is the single page application (SPA), that authenticates against an arbitrary Identity and Access Management System (IAM). “Service A” is the modern backend that supports the client as backend API. Therefore, “Service A” provides functionality for the client. “Service B” is a legacy application, for example an old ERP with order information, that was moved into the cloud, but is not refactored nor rewritten to communicate with modern authentication technologies.

In this scenario, the client calls some API on “Service A” that then will call “Service B” to get additional information to present to the user. Since the client and “Service A” communicate with the same authentication technology, the call is straight forward. The client authenticates himself and obtains an access token. When calling the service (“Service A”), the token is transmitted and the service can check with the IAM if the user is authorized to access the system. When “Service A” then calls “Service B” for additional information, it needs to translate the user provided credentials to a format that “Service B” understands. In the provided example, “Service B” is only able to handle Basic Authentication, as explained in Section 2.3.1. This means, if “Service A” wants to communicate with “Service B,” it must implement some translation logic to change the credentials to a format that B understands. This introduces code changes to “Service A,” since “Service B” is a legacy application that is not maintainable.

3.2 Deficiencies

The situation described in the previous section introduces several problems. It does not matter if “Service B” is a third party application to which no code changes can be applied to, or if it is a legacy application that cannot be updated for the time being. Most likely, the code change to provide the ability to communicate will be introduced into the “Service A.” This adds the risk of errors since new code must be produced, which would not be necessary if the legacy service would be refactored. Also, changing “Service A” to communicate wih B may be a feasable solution in a small setup. But as the landscape of the microservice architecture grows, this solution does not scale well. The matrix problem X services $* Y$ authentication methods describes this problematic. As the landscape and the different methods of authentication grows, it is not a feasable solution to implement each and every authentication scheme in all the services.

TODO: describe SHOULD solution

4 Further work

- create common domain model for user
- use this domain language model to translate credentials from and to a specific implementation
- use JWT to harden these credentials (signing)
- operator must create and deliver keys
- operator must have set of public keys

5 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)

6 Todos

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)

Bibliography

Bryan, Paul, and Mark Nottingham. 2013. "Javascript Object Notation (Json) Patch." *RFC 6902 (Proposed Standard).*

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.".

Hardt, Dick, and others. 2012. "The OAuth 2.0 Authorization Framework." RFC 6749, October.

Kratzke, N., and R. Peinl. 2016. "ClouNS - a Cloud-Native Application Reference Model for Enterprise Architects." In *2016 IEEE 20th International Enterprise Distributed Object Computing Workshop (EDOCW)*, 1–10. https://doi.org/10.1109/EDOCW.2016.7584353.

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.

Reschke, Julian. 2015. "The'basic'http Authentication Scheme." *Work in Progress, Draft-Ietf-Httpauth-Basicauth-Update-07.*

Sakimura, Natsuhiko, John Bradley, Mike Jones, Breno De Medeiros, and Chuck Mortimore. 2014. "Openid Connect Core 1.0." *The OpenID Foundation*, S3.