PolyCode Technical Architecture Document

Alexis Bernard

Table of Contents

Introduction . . . 1
1. Fonctionnal quarters . . . 2
  1.1. User story vocabulary . . . 2
  1.2. User stories . . . 3
  1.3. PolyCode microservices architecture . . . 8
2. Authentication . . . 10
  2.1. Oauth 2.0 . . . 10
  2.2. JWT . . . 14
  2.3. PKCE . . . 16
  2.4. Oauth 2.1 . . . 17
  2.5. SSO . . . 18
  2.6. OpenID Connect . . . 18
  2.7. LDAP . . . 19
  2.8. Tools . . . 19
  2.9. PolyCode authentication migration . . . 20
3. Inter-process communication . . . 30
  3.1. Methods . . . 30
  3.2. Architecture patterns . . . 33
  3.3. Service discovery . . . 39
  3.4. PolyCode integration . . . 45
4. Data Management . . . 51
  4.1. Data Consistency . . . 51
  4.2. Data Backup . . . 51
  4.3. PolyCode Data Migration . . . 52
5. Tracing . . . 53
6. Data Management . . . 54
  6.1. Data Consistency . . . 54
  6.2. Data Backup . . . 55
  6.3. PolyCode Data Migration . . . 55
7. Security . . . 56
  7.1. Application level . . . 56
  7.2. Communication level . . . 57
  7.3. Data Security . . . 62
  7.4. Secrets Management . . . 62
  7.5. PolyCode Integration . . . 63
Glossary . . . 64
Bibliography . . . 65

Introduction

Polycode is a web application. His goal is to provide a simple and intuitive way to learn programming, get certifications, participate in coding campaings, and to make coding interviews.

The application is available at polycode.do-2021.fr.

Polycode is actually based on a monolithic architecture. The goal of this document is to explain why and how bring polycode to a microservice architecture, and proof thoose choices by multiples POCs.

Chapter 1. Fonctionnal quarters

To start with the microservices architecture, it is necessary to define the functional quarters of the application, to initiate a first separation of the application into microservices. A functionnal quarter is define by a set of user stories.

1.1. User story vocabulary

The following table contains the vocabulary of PolyCode. He is used to describe the user stories in the next section.

Guest

Someone who doesn't have a PolyCode account.

User

Someone who has a PolyCode account.

Module

A coherent set of module and contents, organized in a tree structure. A module can be a challenge, a practice, a certification, a submodule, a test, etc.

Content

A coherent set of components, organized in a tree structure. It's a branch of the tree that are the modules. A content can be an exercice, a lesson, a question, etc.

Component

A coherent set of components, organized in a tree structure. It's a branch or a leave of another branch that are the contents or the components. A component can be an code editor, a quizz, a markdown, a mcq, a container, etc.

Test

A type of module. He is made to evaluate users. It has a grade, which is the average of the grades of all contents of the test.

Assessment

A section of PolyCode made to evaluate users, by passing tests.

Campaign

The realization of an assessment in a set of users.

Candidate

Someone who is invited to participate to a campaign. Became an user if he accepts the invitation.

Assessment creator

A user who can create an test or an assessment, then invite someone to participate to it, as a candidate.

Practice creator

A user who can manage his modules.

Submitable

A component that can be submitted by the user. It can be a code editor, a quizz, a mcq, etc.

Validator

The excepted response to a submitable.

Submission

A user answer to a submitable. It is verified by a set of validators.

Tag

A keyword that can be associated to a content, a candidate, etc.

Item

A purchasable object. It can be a hint, a validator, etc.

Purchase

The action of buying an item.

Hint

A type of item that gives a clue to the user to solve a component. It is bought with polypoints.

Polypoints

A virtual money responsible of the gamification of PolyCode. It can be used to buy items, like hints. It also used to rank users. The polypoints are collected by the users by passing modules and contents.

Team

A group of users. The team polypoints are the sum of the polypoints of all users of the team, and is used to rank the teams.

Captain

A user who can manage his team.

1.2. User stories

The following tables contains the most of the user stories of PolyCode. It is not exhaustive, but it contains the most important user stories. She is organized by functional quarters.

Account

As	I want	So that
Guest	Create my account	Connect to my account
User	Show my account details	See my user information, like my email or my account settings

As	I want	So that
User	Update my account details	Edit my user information, like my email or my account settings
User	Delete my account	Delete my user information
User	Connect to my account with my email and my password	Use the PolyCode features
User	Logout from my account	Remove my session and come back as a guest
User	Reset my password	Edit my credentials
User	Show an account profile	Get some information about a user (pseudo, rank, etc)
User	Show the ranking of the users	Compare my rank with other users

Team

As	I want	So that
User	Create a new team	Become a captain of a team
User	Show my team details	See my team information, like the name or the description
Captain	Update my team details	Edit my team information
Captain	Delete my team	Dissolve my team
Captain	Invite other users in my team	Grow my team and make it more active
Captain	Kick a member from my team	Remove a member who is not active anymore or who is not a good fit for my team
User	Accept an invitation to join a team	Join a team
User	Reject an invitation to join a team	Don’t join a team
User	Leave a team	No longer be a member of a team
Captain	Give the captain role to another member of my team	Promote a member of my team who is more active than me
User	Show a team profile	Get some information about a team (name, rank, etc)
User	Show the ranking of the teams	Compare my team with other teams

Practice

As	I want	So that
Content creator	Create a content with some components	Provide a way to learn or practice a specific notion or a specific language
Content creator	Edit one of my contents	Update my content information
Content creator	Delete one of my contents	Remove my content that is not useful anymore
Content creator	Create a module	Organize contents by notion or by language
Content creator	Edit one of my modules	Update my module information
Content creator	Delete one of my modules	Remove my module that is not useful anymore
Content creator	Add a sub-module to one of my modules	Aggregate modules into my module
Content creator	Add a content to one of my modules or sub-modules	Aggregate contents into my module or my sub-module
User	Show module list	Choose a module to do
User	Show the description of a module	Take information about the module
User	Show content list	Choose an content to do
User	Show the description of a content	Take information about the content
User	Show new modules and contents	Find modules that I never did
User	Add files to the code editor	Write my code solution in multiple files
User	Remove files from the code editor	Remove files that I don’t need anymore
User	Follow my progression in modules and contents	Know how much I have done

Assessment

As	I want	So that
Assessment creator	Create a test	Create an exam to test candidate’s knowledge
Assessment creator	Edit one of my tests	Update my test information
Assessment creator	Delete one of my tests	Remove my test that is not useful anymore
Assessment creator	Set a time limit of my tests	Candidates have to conclude the test in a limited time

As	I want	So that
Assessment creator	Set a time limit for each question of a test	Candidates have to answer to a question in a limited time
Assessment creator	Set an amount of points for each question of a test	Candidates cumulate points through the test
Assessment creator	Create a campaign	Evaluate a group of candidates
Assessment creator	Edit one of my campaigns	Update my campaign information
Assessment creator	Delete one of my campaigns	Remove a campaign that is not useful anymore
Assessment creator	Set a start date for one of my campaigns	Start automatically my campaign at a specific date, candidates can then start the test
Assessment creator	Add candidates to one of my campaigns by web interface	Make candidates participate to my campaign
Assessment creator	Add candidates to one of my campaigns by csv file	Make candidates participate to my campaign
Assessment creator	Add candidates to one of my campaigns by api call	Make candidates participate to my campaign
Assessment creator	Remove candidates from one of my campaigns by web interface	Remove candidates from my campaign
Assessment creator	Remove candidates from one of my campaigns by csv file	Remove candidates from my campaign
Assessment creator	Remove candidates from one of my campaigns by api call	Remove candidates from my campaign
Assessment creator	Add tags to candidates in one of my campaigns	Organize candidates of my campaign in groups
Candidate	Accept a campaign invitation	Participate in a campaign
Candidate	Decline a campaign invitation	Do not participate in a campaign
User	Show test list	Choose a test to do
User	Show the description of a test	Take information about the test
User	Start a test of a campaign of which I am a candidate	Participate in a campaign
Candidate	Come back to a test of a campaign that I didn’t finish	Finish a test that I didn’t finish before

Submission

As	I want	So that
User	Submit a solution to a submitable	Complete a content then earn polypoints, or complete a test
User	For a code editor, submit a solution for a specific validator	Check if my solution is correct for this validator
User	Show the last submissions of a submitable who passed all validators	Improve my old solution and submit it again

Stats

As	I want	So that
Assessment creator	See the results of one of my campaigns in raw data	Analyze the results of my campaign to have a first overview
Assessment creator	See the results of one of my campaigns in raw data in a chart	Analyze the results of my campaign in a more visual way
Assessment creator	Sort candidates in one of my campaigns (by tag, by score, by time, etc.)	Find best candidates easily
Assessment creator	Export a general report of one of my campaign in a pdf	Save the results of my campaign
Assessment creator	Export a detailed report of one of my campaign in a pdf	Save the results of my campaign
Assessment creator	Export a detailed report of one of my campaign in a csv	Save the results of my campaign and use it in a spreadsheet

Shop

As	I want	So that
User	Buy hints for a submitable with polypoints	Get some help to solve a submitable

Mailer

As	I want	So that
User	Receive a welcome email when I sign up	Know that my account is created
User	Receive an email to confirm my email address	Verify that my email address is correct
User	Receive an email when I’m invited in an team	Get notified and get links to accept or decline the invitation

As	I want	So that
User	Receive an email when I'm kicked out of a team	Get notified when I'm removed from a team
User	Receive an email when my team is dissolved	Get notified when my team is deleted
Candidate	Receive an email when I'm invited in a campaign	Get notified and get links to accept or decline the invitation
Assessment creator	Send again a campaign invitation email to a candidate	Be sure that a candidate has received the invitation
Candidate	Receive an email when my participation in the test of a campaign is received	Be sure that I have send my answers

1.3. PolyCode microservices architecture

Below is a proposed architecture for PolyCode microservices.

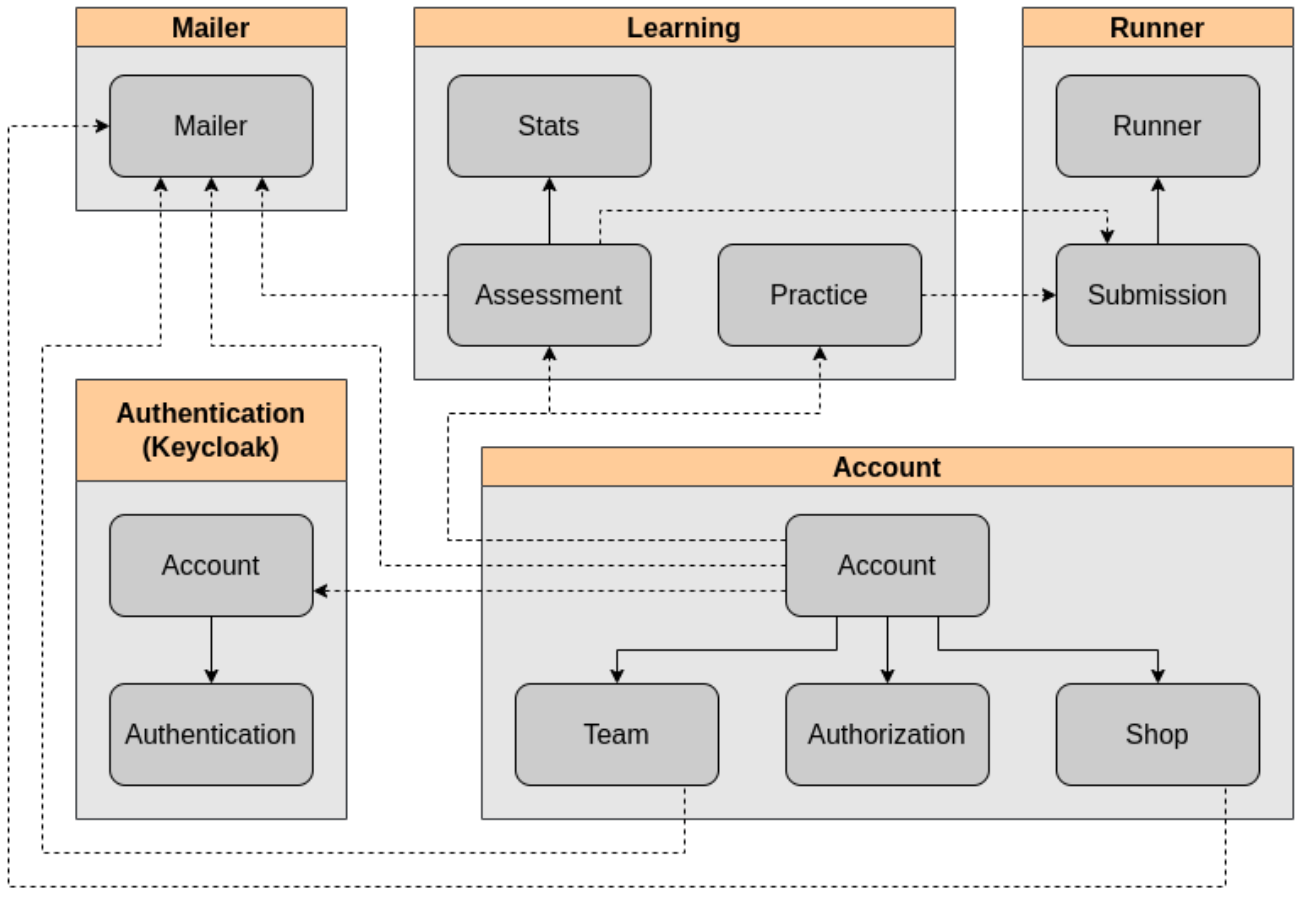

Figure 1. PolyCode microservice architecture schema

The main application will be detached into several microservices. This proposal is made of five microservices, themselves made of several modules. In the future, each microservice module may be detached into a new microservice.

For now, the five microservices are:

- `Authentication` (Keycloak): This microservice will be used for all PolyCode users and services authentication.
- `Account`: This microservice will be used for all PolyCode users and services account management, including the user profile, the user preferences, the user teams, etc. It also used to manage the user's transactions (shop). For now, this microservice will aslo do the authorization part, because it is already implemented.
- `Mailer`: This microservice will be used to send emails to users.
- `Learning`: This microservice will be used to manage the learning part of PolyCode, including the campaigns, the tests, the contents, the modules, the submitables, the validators, etc. It also used to get statistics about the assessments part.
- `Runner`: This microservice will be used to run the user's code and return standard output and standard error.

In a first time, we will only detach the authentication microservice from the main application. The other microservices will be detached later. The authorization part will be migrated to the authentication microservice in a second time.

Chapter 2. Authentication

Authentication is the process of verifying the identity of a user or process. In this part, we will see the different authentication methods and how to implement them.

2.1. Oauth 2.0

OAuth 2.0 is an framework that enables websites or applications to access resources from others web applications, without having to share the user's credentials.

Clearly, it allows a user to grant limited access to their resources on one site, to another site, without having to expose their password.

2.1.1. Terminologies

With Oauth 2.0, it's important to understand the different terms. Here is a list of the most important of them :

Protected resource

The data you want to access. For example, a user's profile information.

Resource server

The server hosting the protected resource. This is the API you want to access.

Authorization server

The server that authenticates the user and issues access tokens after getting proper authorization. This is the server that you use to log in to your account.

Client

The application that wants to access the protected resource. For example, a mobile app or a website.

Resource owner

The user that owns the protected resource. When the resource owner is a person, it is referred to as an end-user.

Redirect URI

The URI to which the authorization server will redirect the user after obtaining authorization.

Response type

The type of the response used to get the authorization code. The authorization code is used to get the access token.

Scope

The scope of the access request. This is a list of space-delimited, case-sensitive strings, defined by the authorization server.

Consent

The small box that asks the user if they want to allow the client to access their protected resources.

Client ID

The client identifier issued to the client during the registration process. This is the public identifier for the client.

Client Secret

The client secret issued to the client during the registration process. This is the secret known only to the client and the authorization server.

Authorization Code

The temporary code issued by the authorization server in exchange for the access token request.

Access token

The token issued by the authorization server. The client uses the access token to access the protected resource.

Access token

The token issued by the authorization server. The client uses the refresh token to get a new access token.

2.1.2. Flows

There is mutliple flow to authenticate a user. Here is the most common ones, and his differences.

Flow property	Code	Implicit	Hybrid
Browser redirection step	✓	✓	✓
Backend request step	✓	×	✓
Tokens revealed to browser	×	✓	✓
Client can be authenticated	✓	×	✓

Figure 2. Oauth flows comparaison

Authorization code flow

The authorization code flow is the most secure one. It's the one we will use in this project. Here is a diagram of this flow :

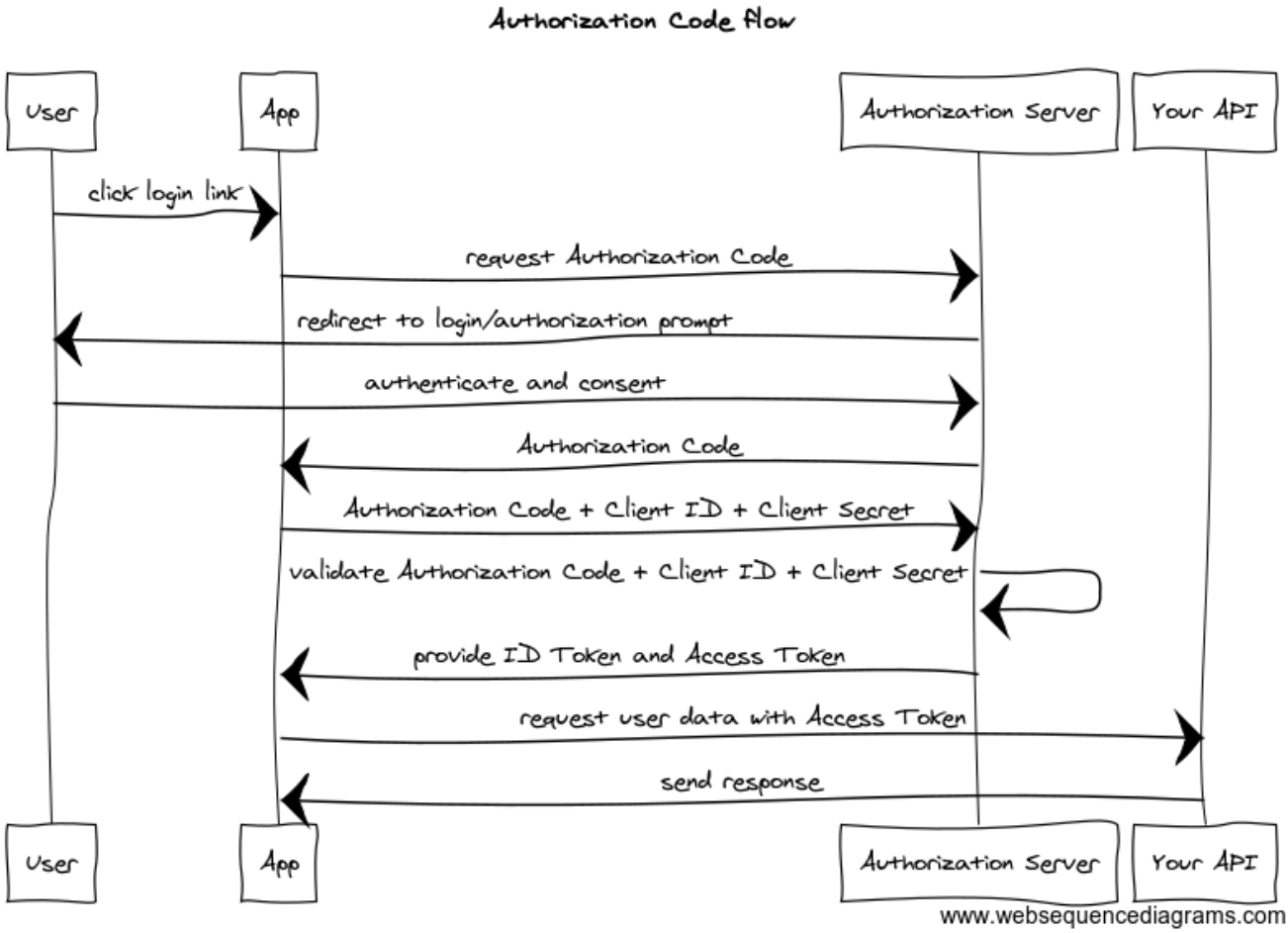

Figure 3. Authorization code flow

Because this flow will be used in PolyCode, we will explain it in details. Here are the steps of this flow :

- The user makes a request to the client. In our case, it's a login request.
- The client request the authorization server to get an authorization code.
- The authorization server redirect the user to a authentication and consent page
- The user authenticate and grant the requested scope.
- The authorization server redirect the user to the client, with the authorization code.
- The client send the authorization code, with the client ID and the client secret, to the authorization server.
- The authorization server validate the client ID and the client secret.
- The authorization server send back the access token and the refresh token (and a ID token if the scope contains openid).
- The client can now use the access token to access the protected resource in the resource server.

Implicit flow

The implicit flow is the most simple one. In this flow, the authorization server return directly the token. This method is not secure and should not be used in production. Here is a diagram of this flow :

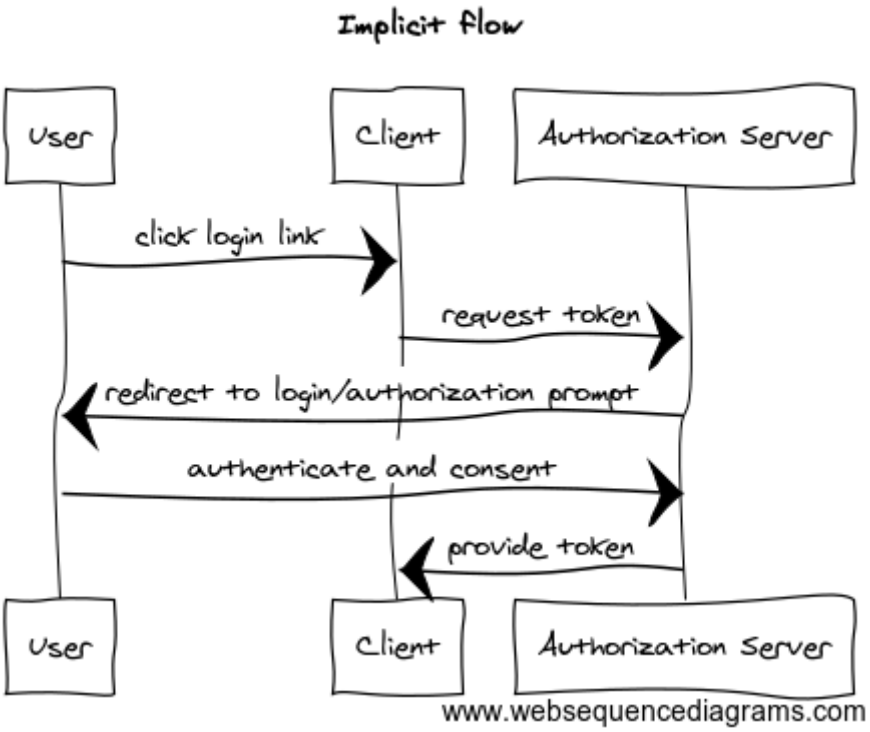

Figure 4. Implicit flow

Hybrid flow

The hybrid flow is the same as authorization code flow, but there is a token server that deliver the token, after authentication by the authorization server. Here is a diagram that explain the request path :

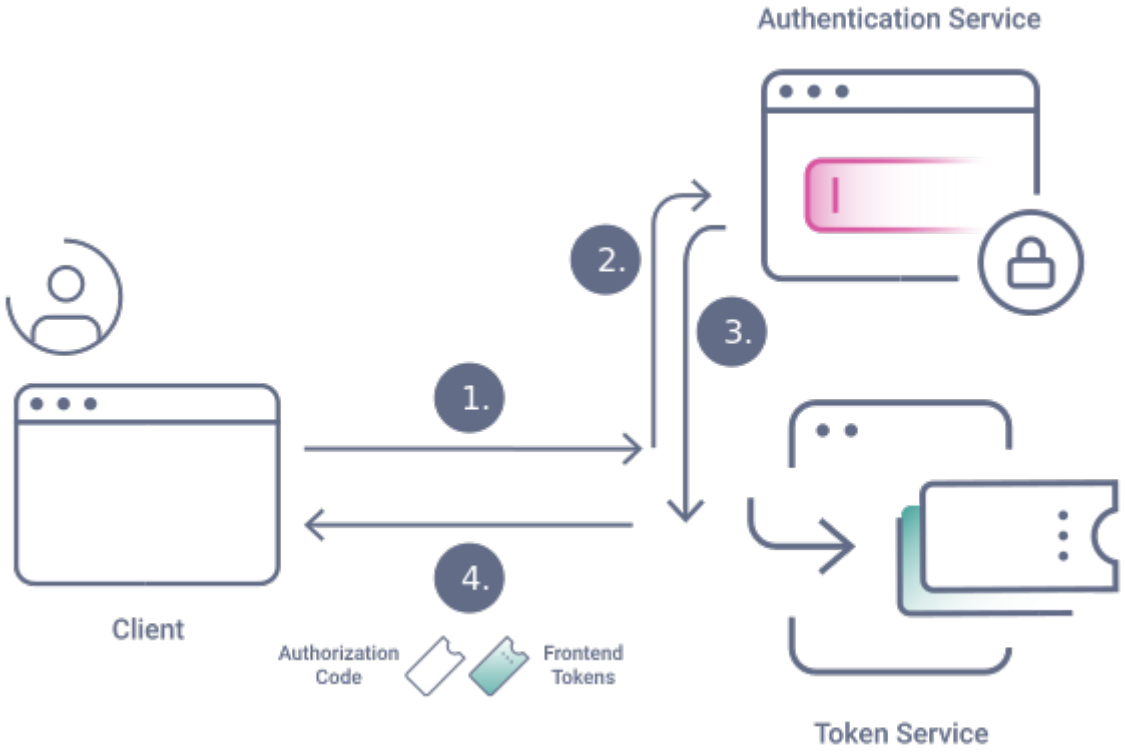

Figure 5. Hybrid flow

2.1.3. Scope

A access token permits to access to a protected resource. But, it's not enough to access to all the resources. The authorization server can restrict the access, depending on the scope of the access token.

The score are asked by the client, in the query. Once the user is authenticated, the authorization server ask the user if he want to grant the requested scope, by displaying a consent page. If the user accept, the authorization server return the access token, with the requested scope.

2.2. JWT

JWT (JSON Web Token) is an open standard that define a compact and URL-safe way to represent claims to be transferred between two parties. The claims in a JWT are encoded as a JSON object. He have a header, a payload and a signature. He also have a expiration date.

The JWT is used for the authentication in OpenID Connect. His integrity is verified by the signature, then no one can modify the data.

2.2.1. Anatomy

Here is an explaination of the JWT parts :

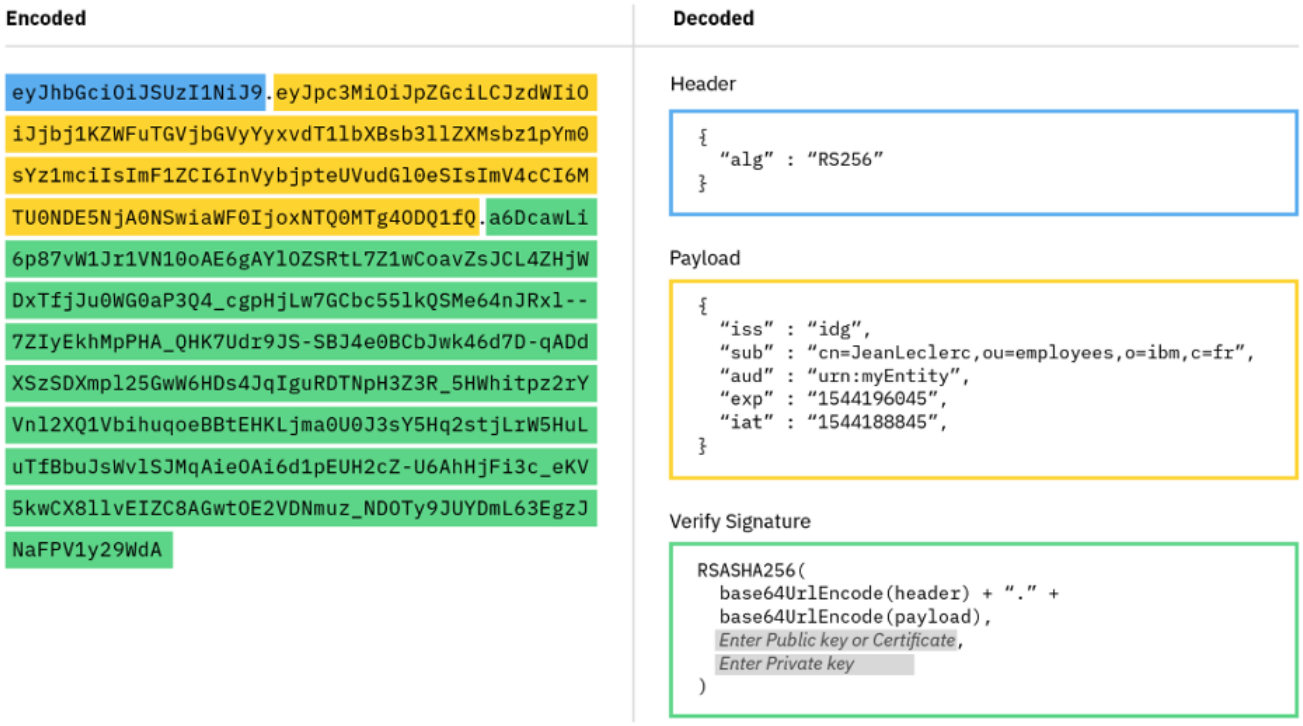

Figure 6. JWT decomposition

2.2.2. Claims

The payload of a JWT contains claims. Claims are statements about an entity (typically, a user) and additional metadata.

There are three types of claims :

- `Registered claims`: These are a set of predefined claims which are not mandatory but recommended. Some of them are : iss (issuer), sub (subject), aud (audience), exp (expiration time), nbf (not before), iat (issued at), and jti (JWT ID).
- `Public claims`: These can be defined at will by those using JWTs. This can be used to share information between parties that agree on using the same claim names.
- `Private claims`: These are the custom claims created to share information between parties that agree on using them and are neither registered or public claims.

Here is an payload of an basic ID token used in OpenID Connect, with the definition of each claims :

Features of the ID token:

- Asserts the identity of the user, called *subject* in OpenID (sub).
- Specifies the issuing authority (iss).
- Is generated for a particular *audience*, i.e. client (aud).
- May contain a nonce (nonce).
- May specify when (auth_time) and how, in terms of strength (acr), the user was authenticated.
- Has an issue (iat) and expiration time (exp).
- May include additional requested details about the subject, such as name and email address.
- Is digitally signed, so it can be verified by the intended recipients.
- May optionally be encrypted for confidentiality.

The ID token statements, or claims, are packaged in a simple JSON object:

```
{
  "sub"       : "alice",
  "iss"       : "https://openid.c2id.com",
  "aud"       : "client-12345",
  "nonce"     : "n-0S6_WzA2Mj",
  "auth_time" : 1311280969,
  "acr"       : "c2id.loa.hisec",
  "iat"       : 1311280970,
  "exp"       : 1311281970
}
```

Figure 7. ID token payload

2.2.3. Refresh token

Once the user is authenticated, the authorization server return an access token, valid for a certain amount of time. Once this token expires, the client can ask for a new one, using the refresh token. Here is a diagram of the flow to get a new token :

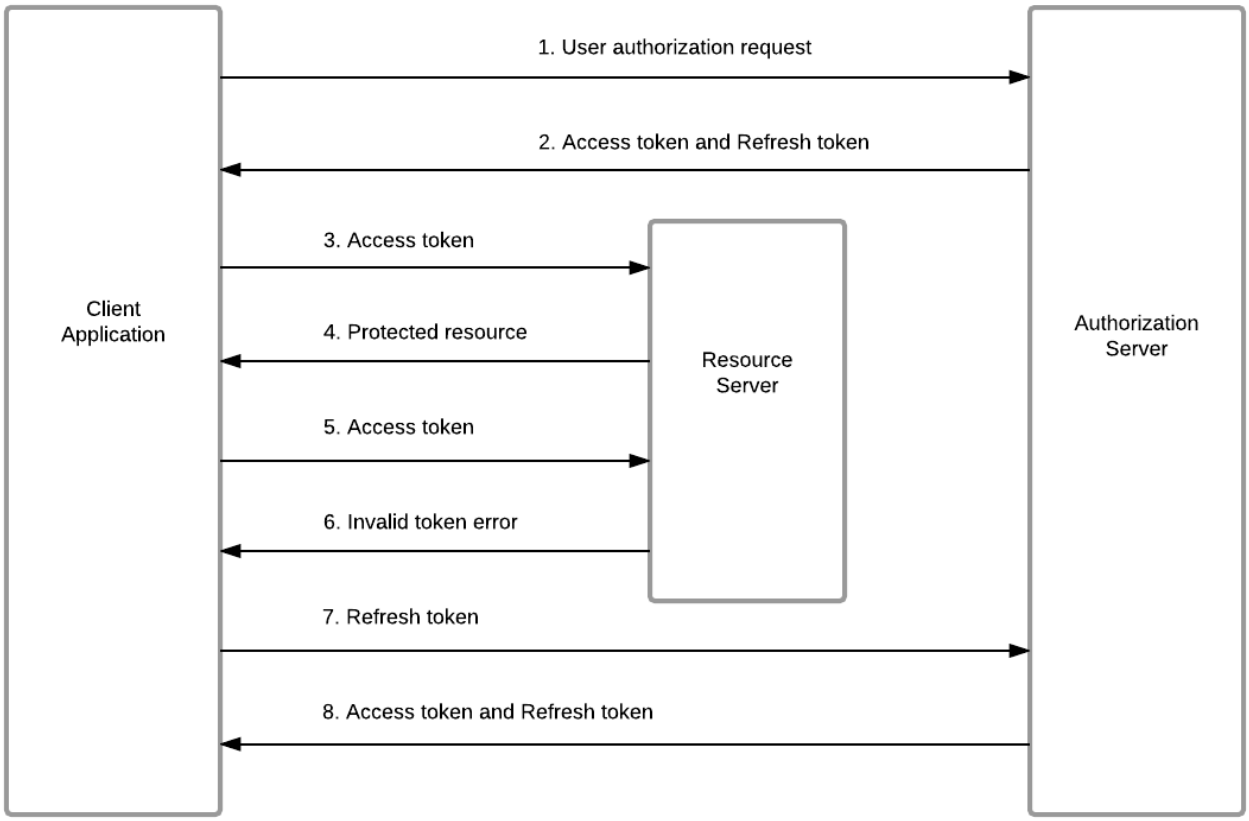

Figure 8. Refresh and access token

2.3. PKCE

PKCE (Proof Key for Code Exchange) is an extension of Oauth 2.0. It's a security feature that prevents an attacker from stealing the authorization code.

2.3.1. Terminologies

Here is the terminologies used in PKCE :

Code Verifier

A huge random string (43 to 128 chars) generated by the client. It's used to generate the code challenge, and is not sent to the authorization server.

Code challenge

A base 64 encoded string of the code verifier. It's sent to the authorization server.

2.3.2. Flow

When the client send the authorization request to the authorization server, he also send the code challenge and the code challenge method (but not the code verifier).

Then, when the client send back the authorization code to the authorization server, he also send the code verifier. The authorization server then hash the code verifier and compare it to the code challenge. If they match, the authorization server send back the access token. Here is a diagram of this flow :

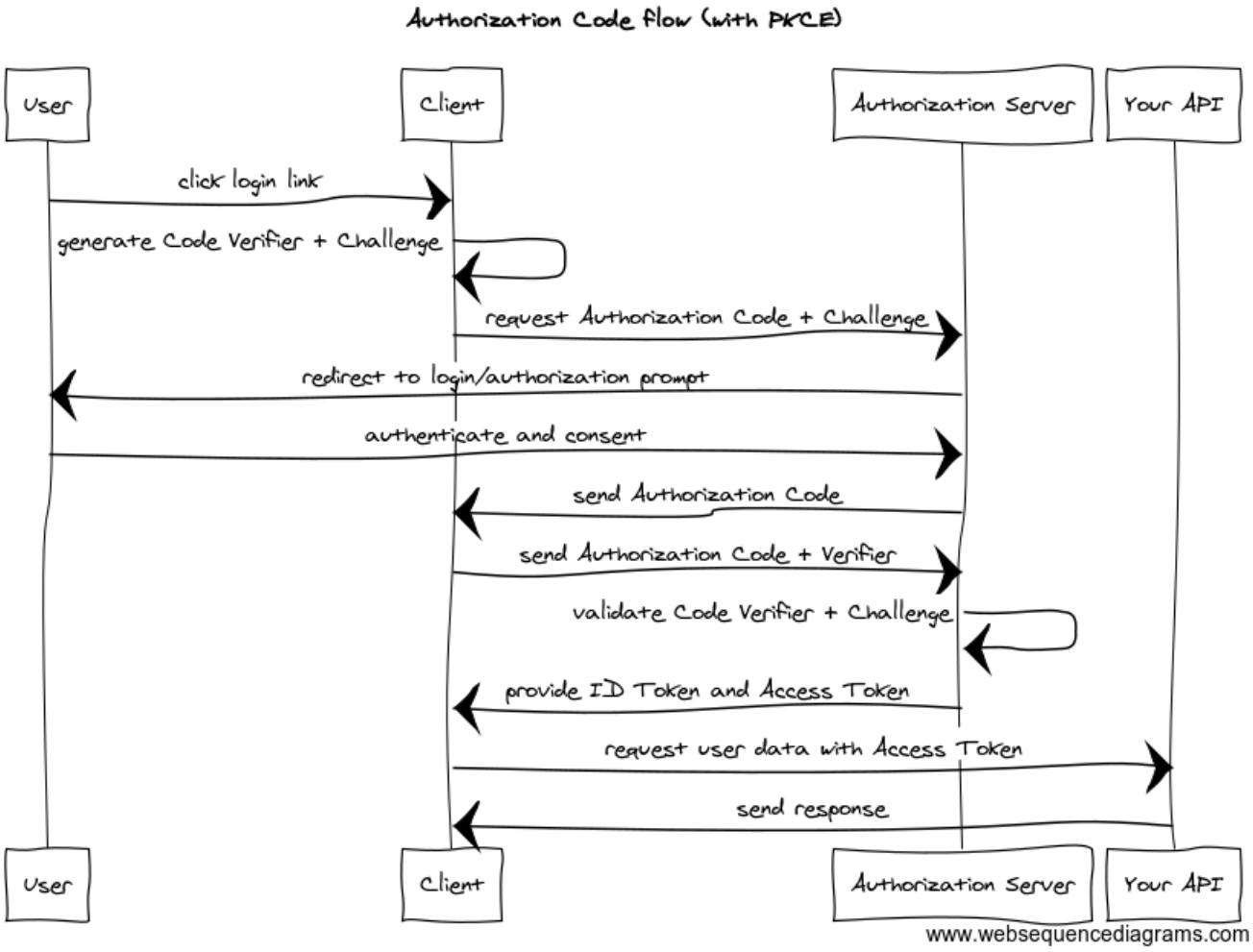

Figure 9. Authorization code flow with PKCE

2.4. Oauth 2.1

Oauth 2.1 is not a new protocol, but a reference document. It's a consolidation of best practices in Oauth 2.0. Here is the differences between Oauth 2.0 and Oauth 2.1 :

- PKCE (Proof Key for Code Exchange) is mandatory.
- Redirect URI must be compared using exact string matching.
- Implicit flow is not allowed.
- Resource owner password credentials grant flow is not allowed (because with this flow, the client can access to the user's password).

- Bearer token usage requires the use of the HTTP Authorization header field instead of the query parameter.
- Refresh tokens must be bound to the client that requested them, or be one-time use.

2.5. SSO

SSO (Single Sign On) is a protocol that allows a user to authenticate once to access multiple applications. With this property, a user logs in with a single ID and password to gain access to a connected system. The user is then signed in to all other systems that are part of the SSO infrastructure. This is in contrast to having to log in separately to each system.

An example of SSO is a student logging in to a university's portal to access email, course registration, and other services.

2.6. OpenID Connect

OpenID Connect (OIDC) is an authentication layer on top of Oauth 2.0. It's a protocol that allows clients to verify the identity of the end-user based on the authentication performed by an authorization server, as well as to obtain basic profile information about the end-user in an interoperable and REST-like manner.

2.6.1. OIDC scopes

There is multiple scopes in OIDC. Let's see the most common ones :

Scope value	Associated claims
email	email, email_verified
phone	phone_number, phone_number_verified
profile	name, family_name, given_name, middle_name, nickname, preferred_username, profile, picture, website, gender, birthdate, zoneinfo, locale, updated_at
address	address

Figure 10. OIDC scopes

2.6.2. Endpoints

Here is the endpoints defines in OIDC :

Core endpoints	Optional endpoints
■ Authorisation ■ Token ■ UserInfo	■ WebFinger ■ Provider metadata ■ Provider JWK set ■ Client registration ■ Session management

Figure 11. OIDC endpoints

2.7. LDAP

LDAP (Lightweight Directory Access Protocol) is a protocol that allows to access to a directory service. It's a client-server protocol, and the client can be a user or an application.

In the case of PolyCode, we will use the polytech LDAP server to manage the users and groups of our applications. In this way, any guest who has an account on the Polytech network can access to polycode without creating a new account.

2.8. Tools

There is multiple tools to help you to implement Oauth 2.0 and OpenID Connect in your application. This part will explain the most common ones and how to use them.

2.8.1. KeyCloak

KeyCloak is an open source identity and access management solution. It's a single sign-on solution that allows you to manage users, applications, roles, groups, etc. It's an Oauth 2.0, OpenID Connect and SAML 2.0 compliant solution.

With the graphical interface, you can manage easily and quickly your resources. It supports multiple factors authentication and social login, all with a lot of adapters to integrate it in your application.

Here is an image of the keycloak architecture :

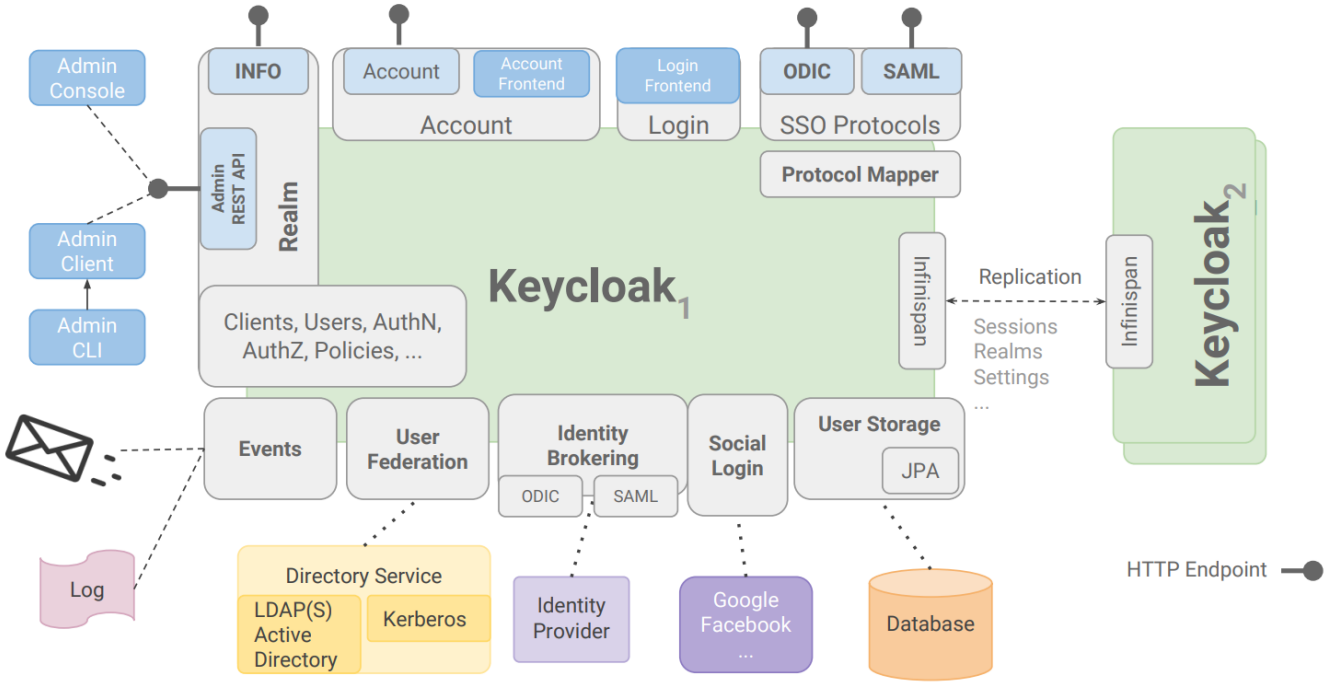

Figure 12. Keycloak architecture

In the case of PolyCode, we will use KeyCloak to manage the authentication and authorization of our users and services.

2.8.2. Others solutions

There is multiple other solutions to manage the authentication and authorization of your users and services. Because we will use KeyCloak for PolyCode, we have detailed it in the previous section.

Here is a list of other possible solutions :

- `Okta` (Cloud service)
- `Auth0` (Cloud service)
- `FusionAuth` (Open source solution)
- `AWS Cognito` (Cloud service)

2.9. PolyCode authentication migration

For now, we use an Oauth2 system, with our own implementation. We will migrate this to KeyCloak, to have a more robust and scalable solution. Note that we will not migrate the authorization part, because is not the purpose here.

2.9.1. Architecture

Here is a diagram of the wanted architecture :

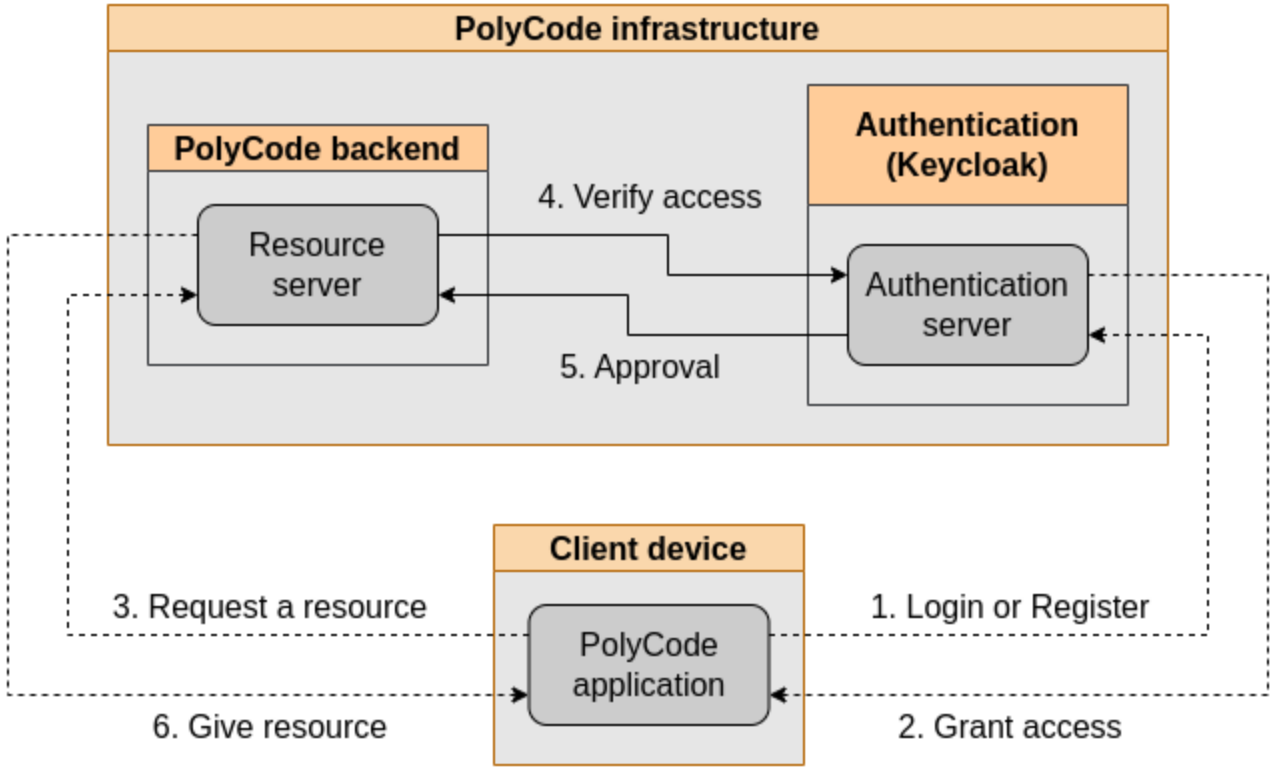

Figure 13. PolyCode authentication architecture schema

The goal is to have a KeyCloak instance, with a client for each service. Each service will have a client secret, and will use it to authenticate to KeyCloak.

The user, when he wants to access to some resources, will be redirected to KeyCloak. He will authenticate with his credentials (or create new), and will be redirected back to the PolyCode UI with a code. The UI will then exchange this code with KeyCloak, and will get an access token. This token will be used to authenticate to PolyCode, verify by KeyCloak against the Polycode API, and will be used to access to the resources.

To do this, we will follow multiple steps. This document is based on the official KeyCloak documentation.

2.9.2. Setup KeyCloak

First, we need to setup a KeyCloak instance. You can use the docker image, or install it on your server. You can run a KeyCloak instance in dev mode with the following command :

```
docker run -p 8080:8080 -e KEYCLOAK_ADMIN=admin -e KEYCLOAK_ADMIN_PASSWORD=admin
quay.io/keycloak/keycloak:20.0.2 start-dev
```

2.9.3. Configure KeyCloak

Once the KeyCloak instance is running, you can access it at localhost:8080 with the `admin`/`admin` credentials if you used the command above.

Once logged in, you can create a new realm. A realm is a group of users, applications, and clients. You can create multiple realms in KeyCloak, and each realm is isolated from the others. For

PolyCode, we will create a `polycode` realm.

2.9.4. KeyCloak clients

A client is an application that wants to use KeyCloak to authenticate users. You can create multiple clients in a realm. For PolyCode, we will create two clients :

- `polycode`: The client for the PolyCode application frontend.
- `polycode-api`: The client for the PolyCode API.

Each client have multiple settings :

- `Client ID`: The unique identifier of the client.
- `Client Protocol`: The protocol used by the client. We will use `openid-connect`.
- `Valid Redirect URIs`: The list of valid redirect URIs for the client. For the `polycode` client, we will use `<frontend_url>/auth/callback`. For the `polycode-api` client, we will use `<api_url>/auth/callback`.
- `Web Origins`: The list of valid web origins for the client. For now, we will use `*` to allow all origins.
- `Credentials`: The credentials used by the client to authenticate to KeyCloak. You can find this in the `Credentials` tab.

For better security, you can also uncheck the `Implicit Flow` option.

2.9.5. KeyCloak users

In our case, we don't need to configure users, roles, groups, etc. because authorization will be managed by the API.

2.9.6. PolyCode integration - POC

Now, we need to integrate KeyCloak in PolyCode. We will explain how to do this with a POC. Here is the project links :

- Polycode-keycloak
- Polycode-keycloak-api

To run the POC, you need first to run the backend, then the frontend, and finally the runner. The backend contains already a KeyCloak instance (not configured).

Then, you can access the polycode-keycloak application at localhost:3001.

2.9.7. Deployments

Here is the deployment diagram of the future PolyCode authentication architecture :

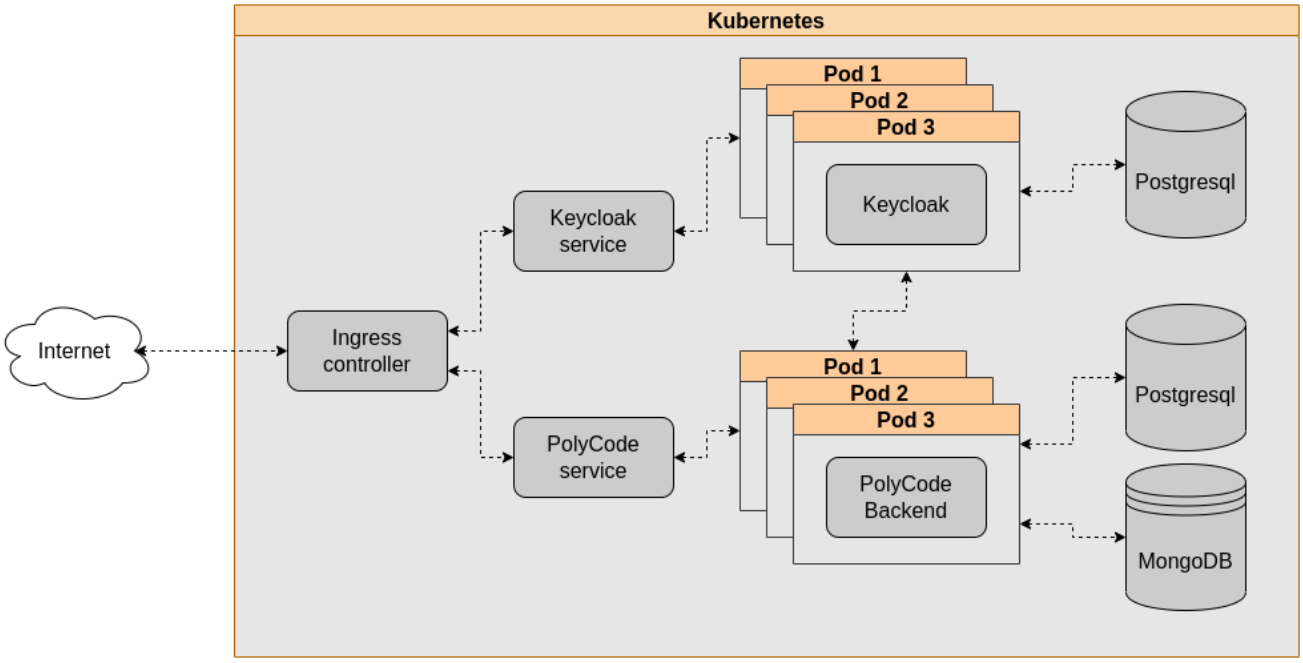

Figure 14. PolyCode authentication deployment schema

All communications between those services are in HTTP, and are secured with TLS.

2.9.8. Sequence diagrams

Here is some sequence diagrams to explain the differences between the authentication process modes.

Vanilla user account creation

Here is the sequence diagram of the user account creation :

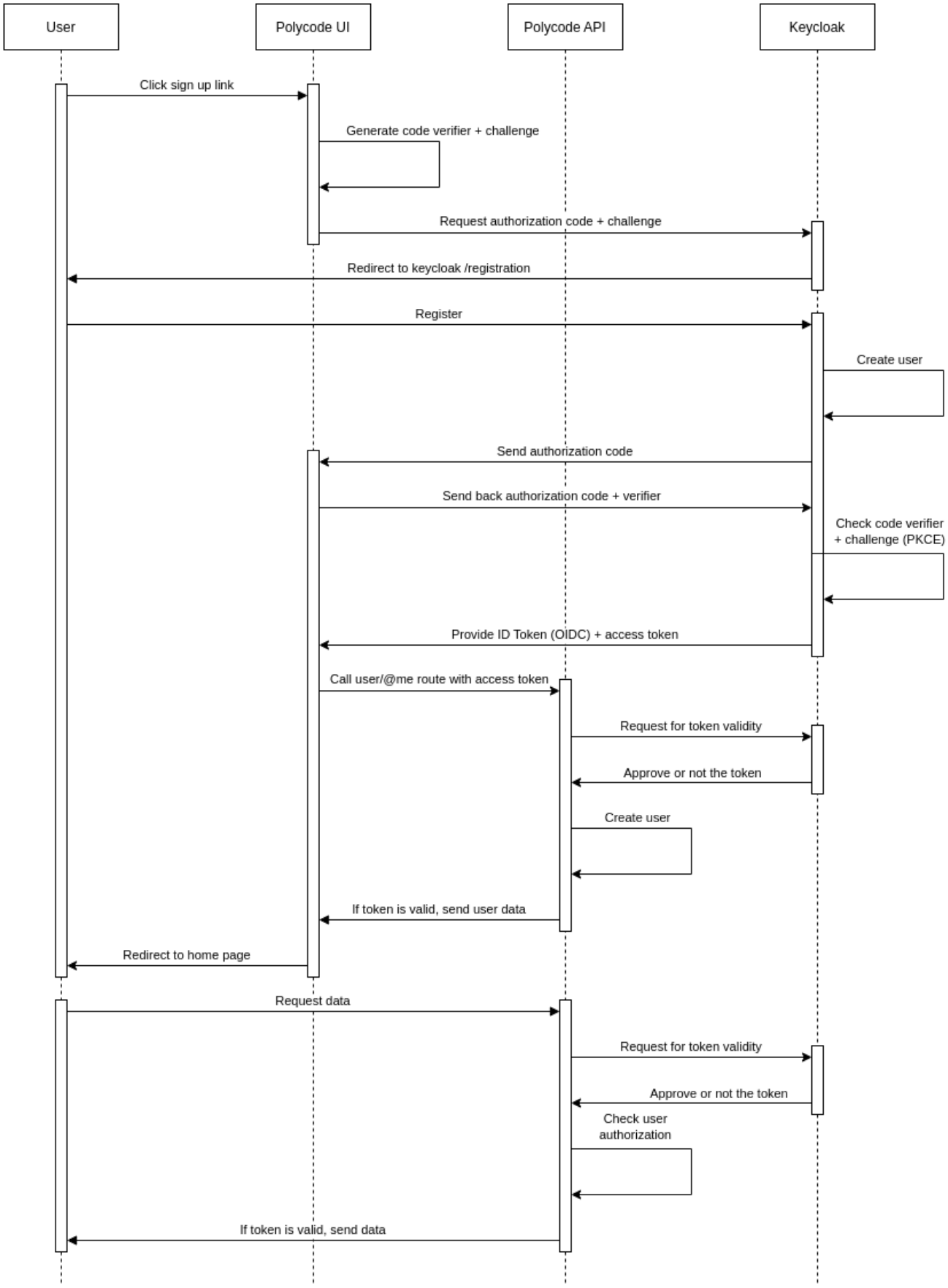

Figure 15. PolyCode authentication sequence diagram - Vanilla user account creation

User account creation via polytech LDAP

Here is the sequence diagram of the user account creation via polytech LDAP :

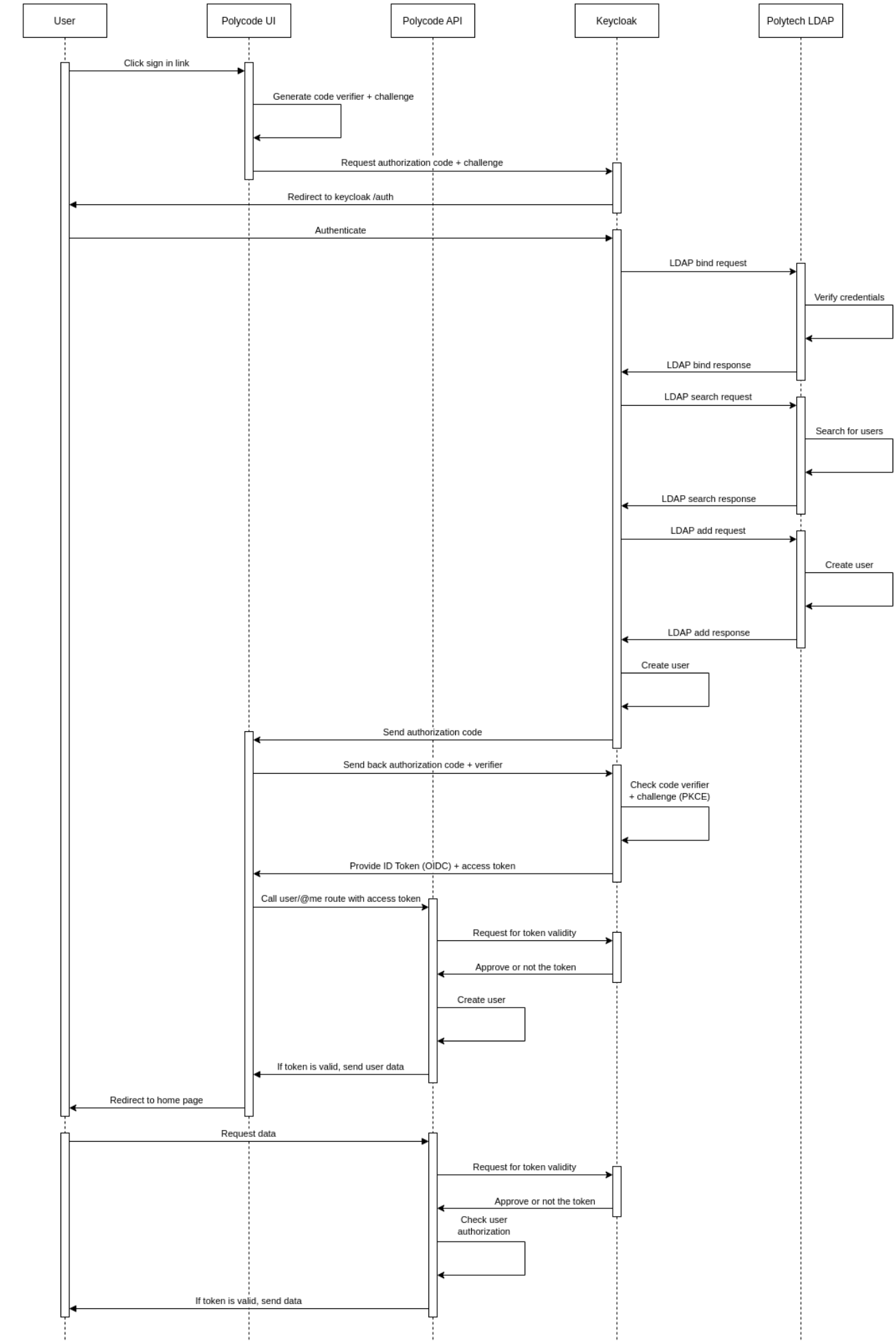

Figure 16. PolyCode authentication sequence diagram - Polytech LDAP account creation

User account creation via google

Here is the sequence diagram of the user account creation via google :

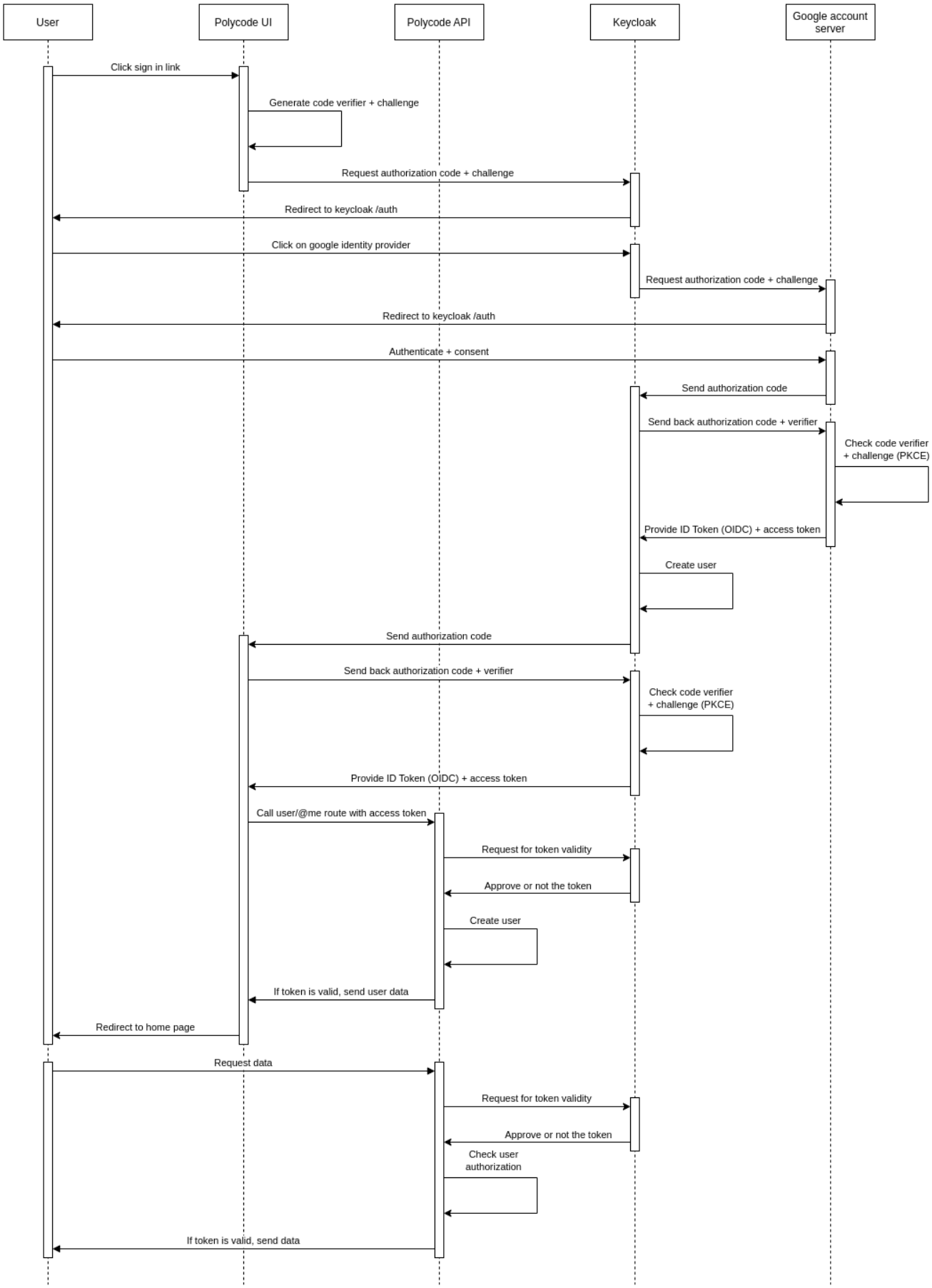

Figure 17. PolyCode authentication sequence diagram - Google identity provider user account creation

Vanilla user authentication

Here is the sequence diagram of the user authentication :

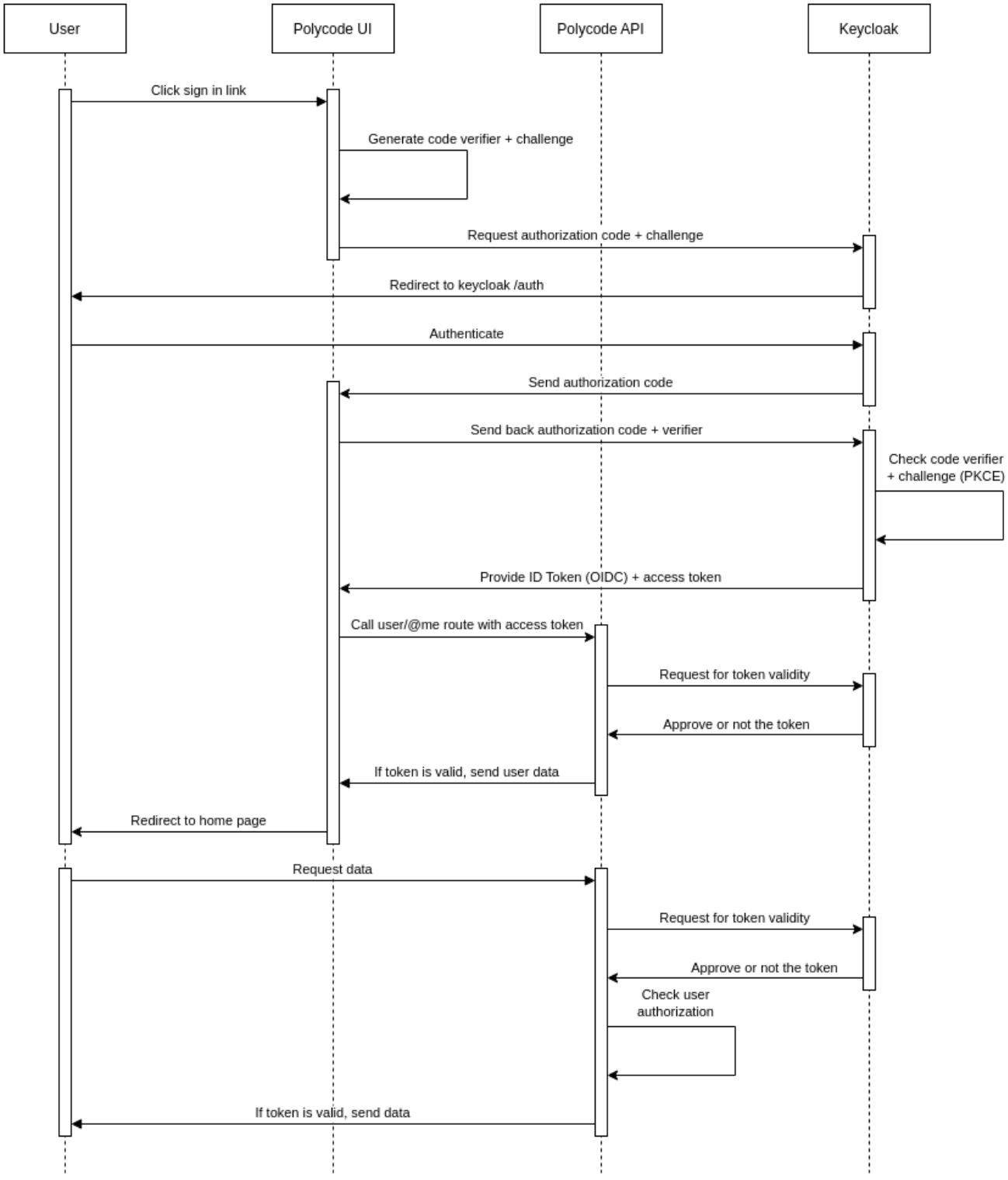

Figure 18. PolyCode authentication sequence diagram - Vanilla user authentication

Polytech LDAP user authentication

Here is the sequence diagram of the Polytech LDAP user authentication :

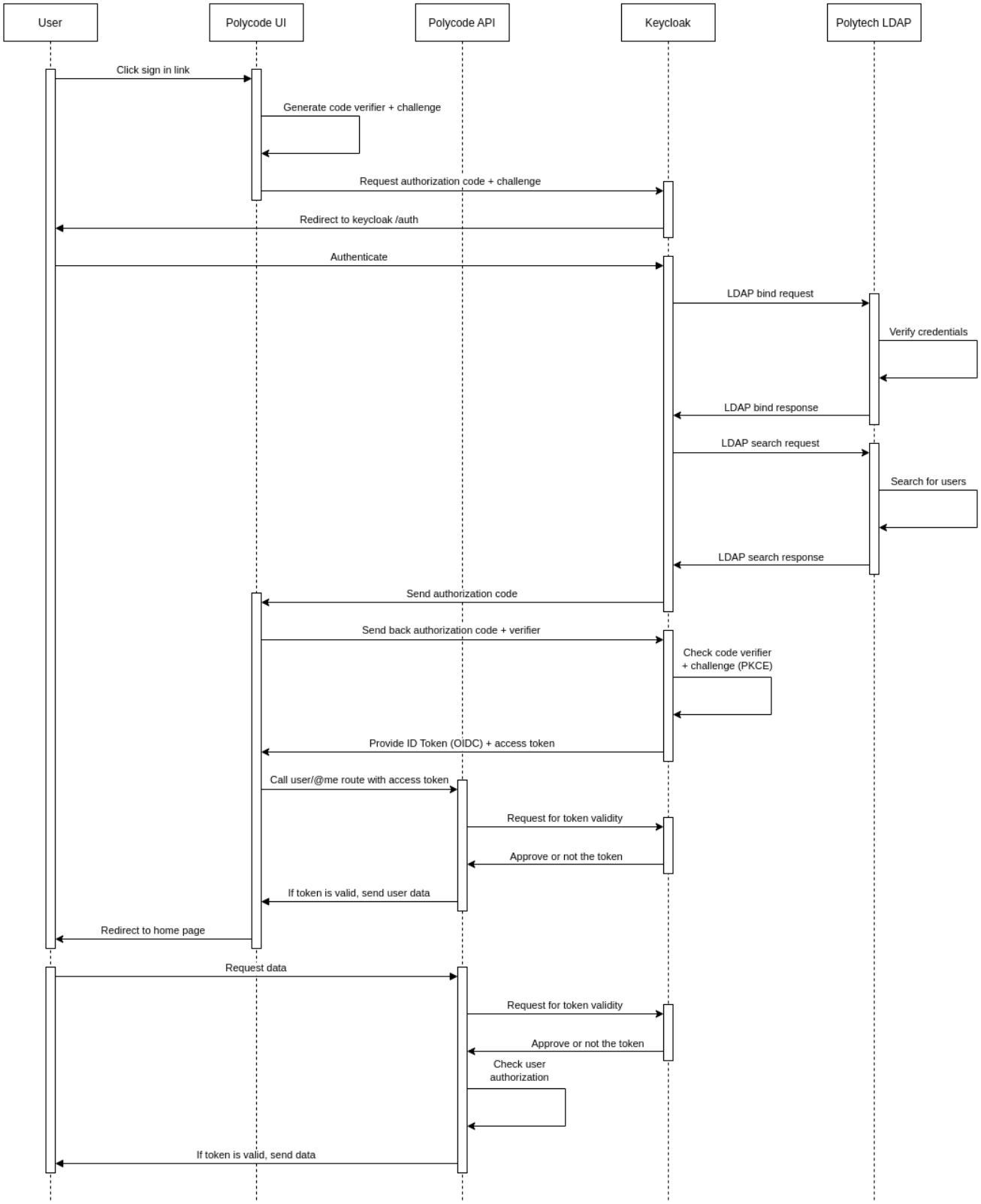

Figure 19. PolyCode authentication sequence diagram - Polytech LDAP user authentication

User authentication via google

Here is the sequence diagram of the user authentication via google :

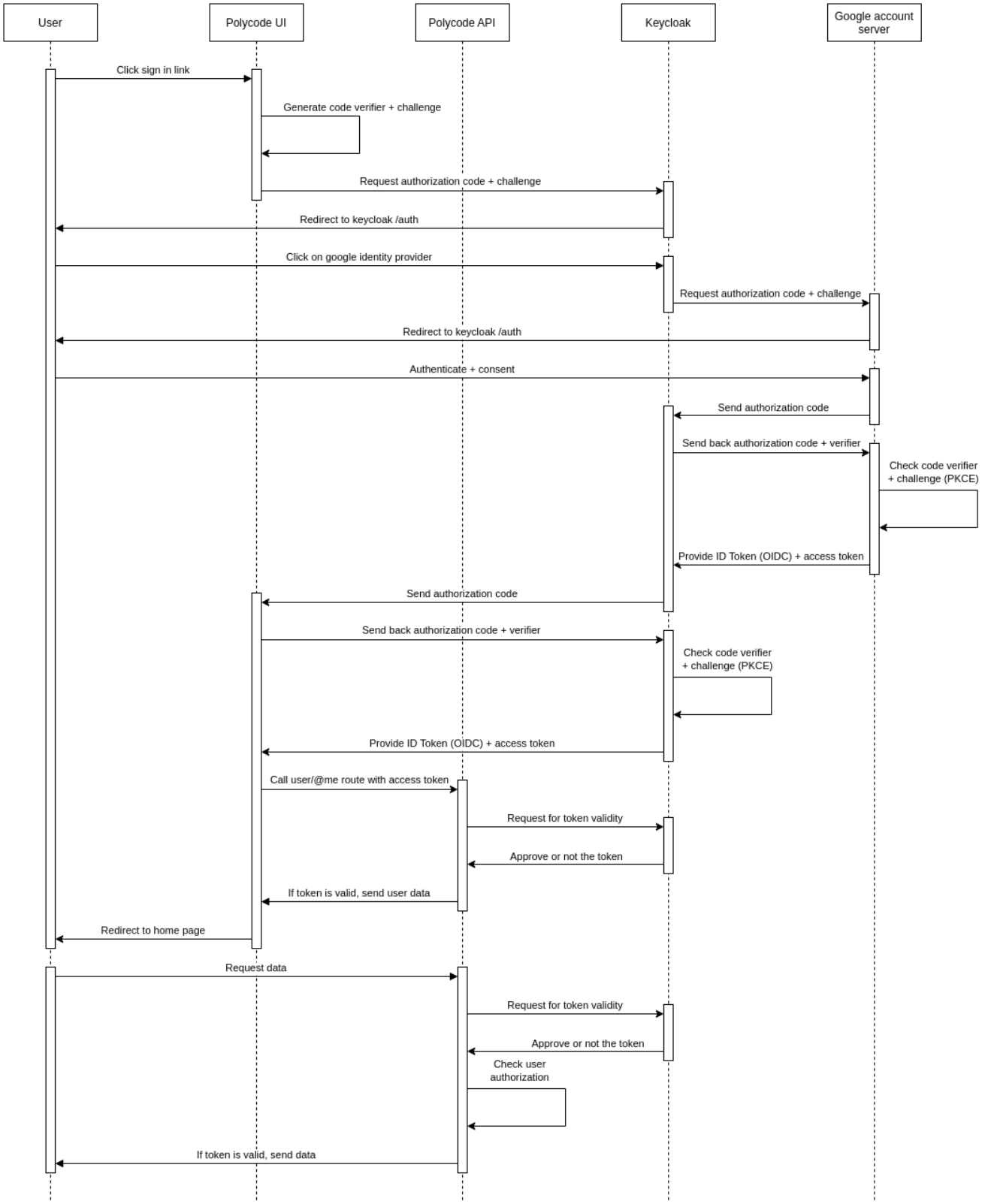

Figure 20. PolyCode authentication sequence diagram - Google identity provider user authentication

Chapter 3. Inter-process communication

In a microservices application, the required load can grow or shrink according on the demand. This means that the number of instances of a service can change at any time. This is a problem because the instances of a service need to communicate with each other. This is where inter-process communication (IPC) comes in.

IPC refers to the mechanisms used by different processes to communicate with each other. Here, we approach this subject within the context of a microservice architecture application. This is important because microservices are typically designed to be independently deployable and scalable, which means they are often deployed on different machines or in different locations. This makes IPC a critical component of application architectures.

3.1. Methods

There is a wide range of IPC mechanisms available, and each has its own advantages and disadvantages. Here, we will focus on the most popular ones, by category. We will also compare them in terms of their suitability for microservices architectures.

3.1.1. Synchonous communication

Synchronous communication is a type of communication where the sender service waits for a response from the invoker service before continuing. There are two main mechanisms for synchronous communication: REST and RPC.

REST

REST (Representational State Transfer) is a architectural style that defines a set of constraints to be used for creating Web services. RESTful Web services allow the requesting systems to access and manipulate textual representations of Web resources by using a uniform and predefined set of operations. RESTful Web services are stateless, which means that the server does not need to know anything about the state of the client that is making the request.

In the context of REST, a resource is a piece of data that can be accessed via a unique identifier, such as a URL. Resources are manipulated using a set of standard HTTP methods, such as GET, POST, PUT, and DELETE. These methods are used to retrieve, create, update, and delete resources, respectively.

REST APIs use HTTP as the communication protocol and can support a wide range of data formats, including JSON and XML. They are simple and lightweight, making them well-suited for use in microservices architectures.

Overall, REST is a widely used and well-established approach for building web services, and it is particularly well-suited for microservices because of its simplicity and flexibility.

Rest is guided by the 6 following principles:

- `Client-server`: The client and server are independent of each other. The two systems can be developed and deployed independently. A client should know only resource URIs, and that's all.

- `Stateless`: The server does not store any client context between requests. Each request from the client to the server must contain all the information necessary to understand the request, and the server must respond to that request as new.
- `Cacheable`: The server must indicate if a response can be cached or not. If a response is cacheable, then a client or a server side cache can store it and reuse it later, saving bandwidth and server resources.
- `Uniform interface`: The interface between the client and the server must be uniform. This means that the interface should be simple and intuitive. The interface should be independent of the underlying implementation.
- `Layered system`: The client should not be aware of the underlying layers of the system. The client should only be aware of the immediate layer with which it is communicating. For example, a client cannot ordinarily tell whether it is connected directly to the end server or an intermediary along the way.
- `Code on demand` (optional): The server can optionally return executable code to the client. This code can be used to extend the functionality of the client.

RPC

RPC (Remote Procedure Call) is a mechanism for a computer program to call and execute a procedure (subroutine) in another program, through the network. For a developer, the call is like a normal local procedure call, without any details for the remote interaction. The RPC mechanism make agnostic the developer about the language, the platform, the network, the operating system, etc.

RPC is used in microservices architecture applicaions to communicate between processes running on different computers over a network. As a procedure, RPC is a synchronous communication mechanism, which means that the sender service waits for a response from the invoker service before continuing.

gRPC is a modern, open source, high-performance remote procedure call (RPC) framework that can run in any environment. It can efficiently connect services with pluggable support for load balancing, tracing, health checking, and authentication.

By default, gRPC uses proto files to define services and messages. The following example shows a simple hello world service definition:

```
// The greeter service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}
```

```
// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

3.1.2. Asynchronous communication

Asynchronous communication is a type of communication where the sender service does not wait for a response from the invoker service before continuing. Let's speak about the most popular mechanisms for asynchronous communication : Message queues and Publish-subscribe.

Message queues

A message queue is a temporary storage area where messages are placed while waiting to be processed. Message queues are typically used to decouple the sender and receiver of a message, and to provide a buffer for the sender and receiver to handle the message. Here is a simple example of a message queue:

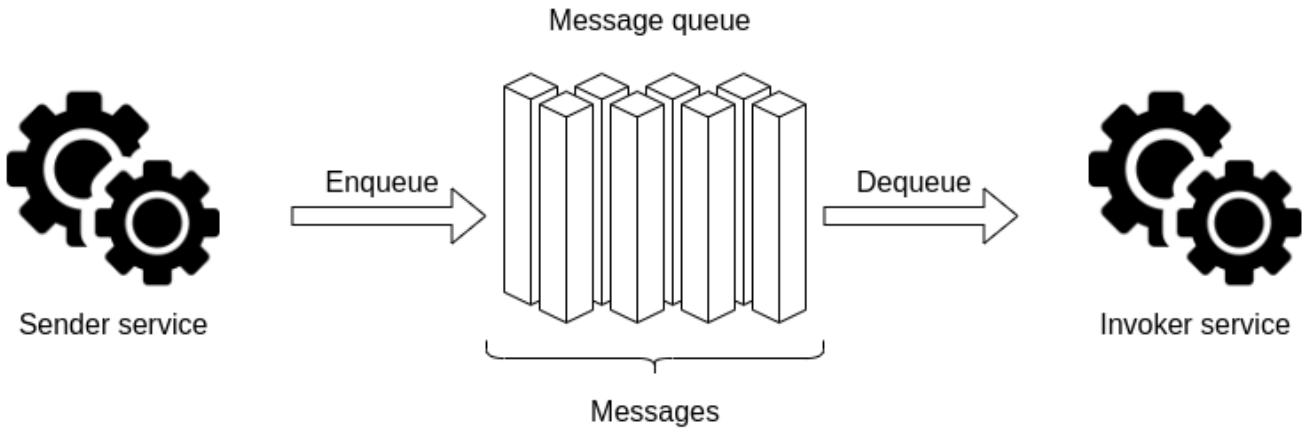

Figure 21. Message queue simple example

In this example, the sender service sends a message to the message queue, and the receiver service reads the message from the queue. The sender and receiver services do not need to be running at the same time, and they do not need to know anything about each other. This decoupling makes message queues well-suited for microservices architectures.

Message queues are typically implemented using a queueing system, such as RabbitMQ, Apache ActiveMQ, or IBM MQ. These systems are typically deployed as a separate service, and they can be used by multiple applications.

Publish-subscribe (pub/sub)

Publish-subscribe is a messaging pattern where senders of messages, called publishers, do not program the messages to be sent directly to specific receivers, called subscribers. Instead, published messages are characterized into classes without knowledge of which subscribers, if any, there may be. Similarly, subscribers express interest in one or more classes and only receive messages that are of interest, without knowledge of which publishers, if any, there are. The message router, or message broker, is responsible for managing the information flow between publishers and

subscribers.

The following diagram shows a simple example of a publish-subscribe system:

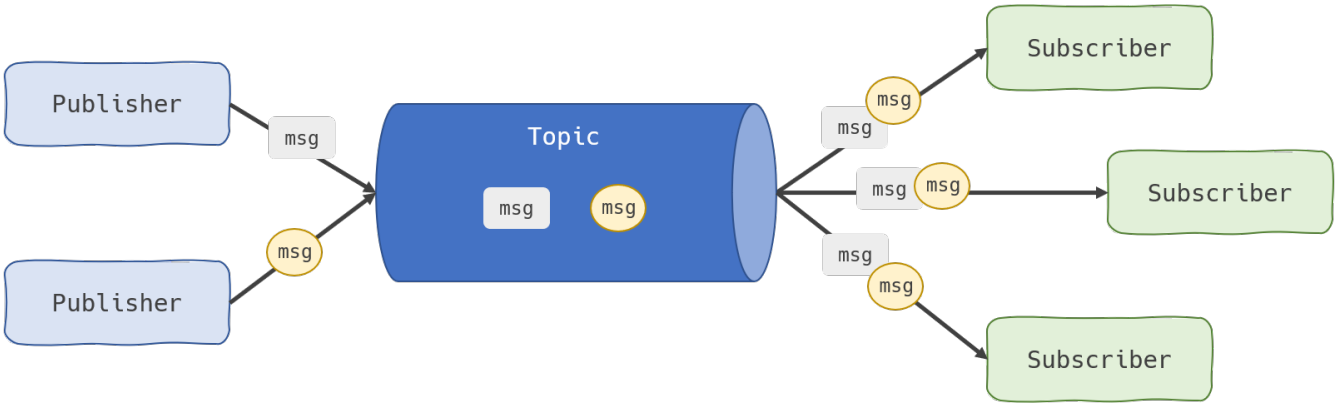

Figure 22. Publish-subscribe simple example

In this example, the publisher service publishes a message to the message broker, and the subscriber service subscribes to the message broker. The publisher and subscriber services do not need to be running at the same time, and they do not need to know anything about each other. This decoupling makes publish-subscribe well-suited for microservices architectures.

Publish-subscribe is typically implemented using a message broker, who is responsible for storing messages and delivering them to subscribers., such as Apache Kafka, Google pub/sub, AWS SNS or Redis pub/sub. These systems are typically deployed as a separate service, and they can be used by multiple applications.

3.2. Architecture patterns

In this section, we will see the most common patterns used to communicate between microservices.

3.2.1. API gateway

An API gateway is a single entry point for clients to access the functionality of a distributed application. The API gateway is responsible for routing requests to the appropriate microservice, and for aggregating the results. It works as a reverse proxy, and it can be used to implement security, rate limiting load balancing and other functionality.

The following diagram shows a simple example of an API gateway:

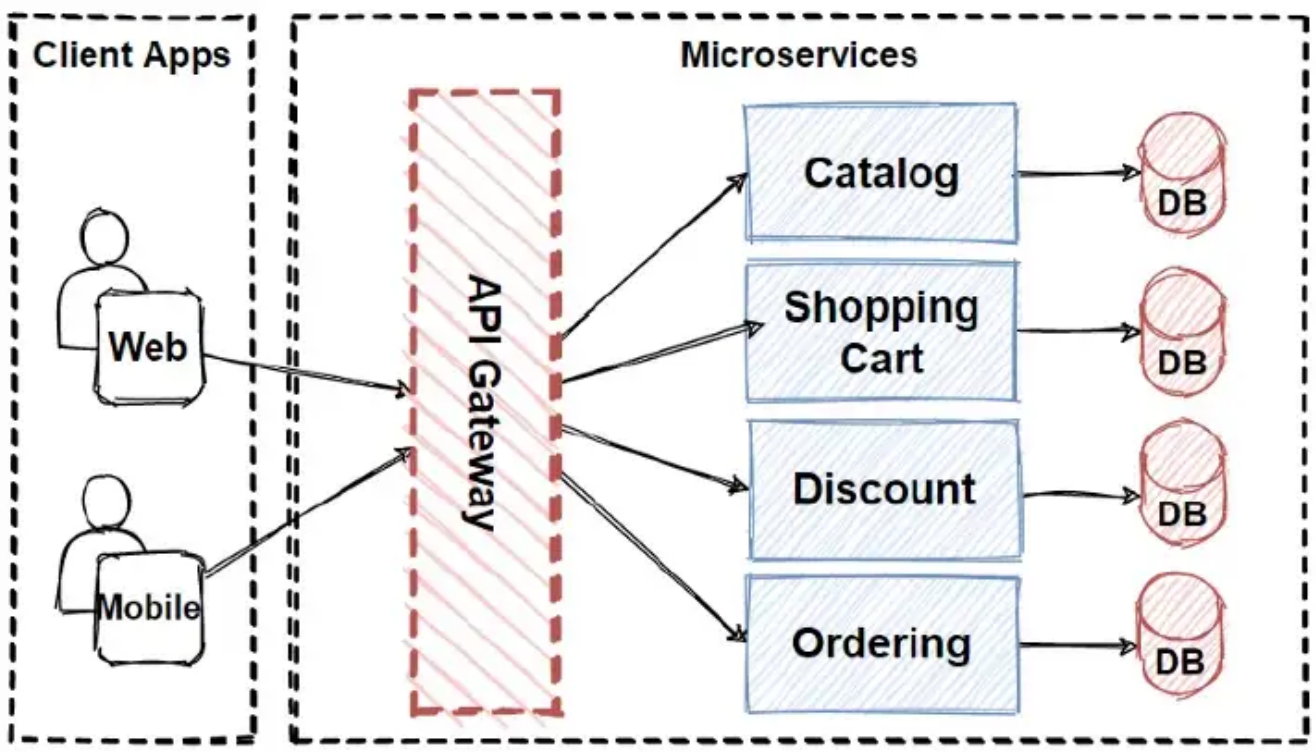

Figure 23. API gateway

In this example, the API gateway is responsible for routing requests to the appropriate microservice, and for aggregating the results.

3.2.2. Circuit breaker

The circuit breaker pattern is a mechanism that allows a service to fail fast and recover quickly. It is used to prevent cascading failures in a distributed system. A circuit breaker is a state machine that can be in one of two states: closed or open. When the circuit breaker is closed, the service can be called. When the circuit breaker is open, the service cannot be called. The circuit breaker can be in the open state for a fixed amount of time, or it can be in the open state until a certain number of calls have been made.

The following diagram shows a simple example of a closed circuit breaker:

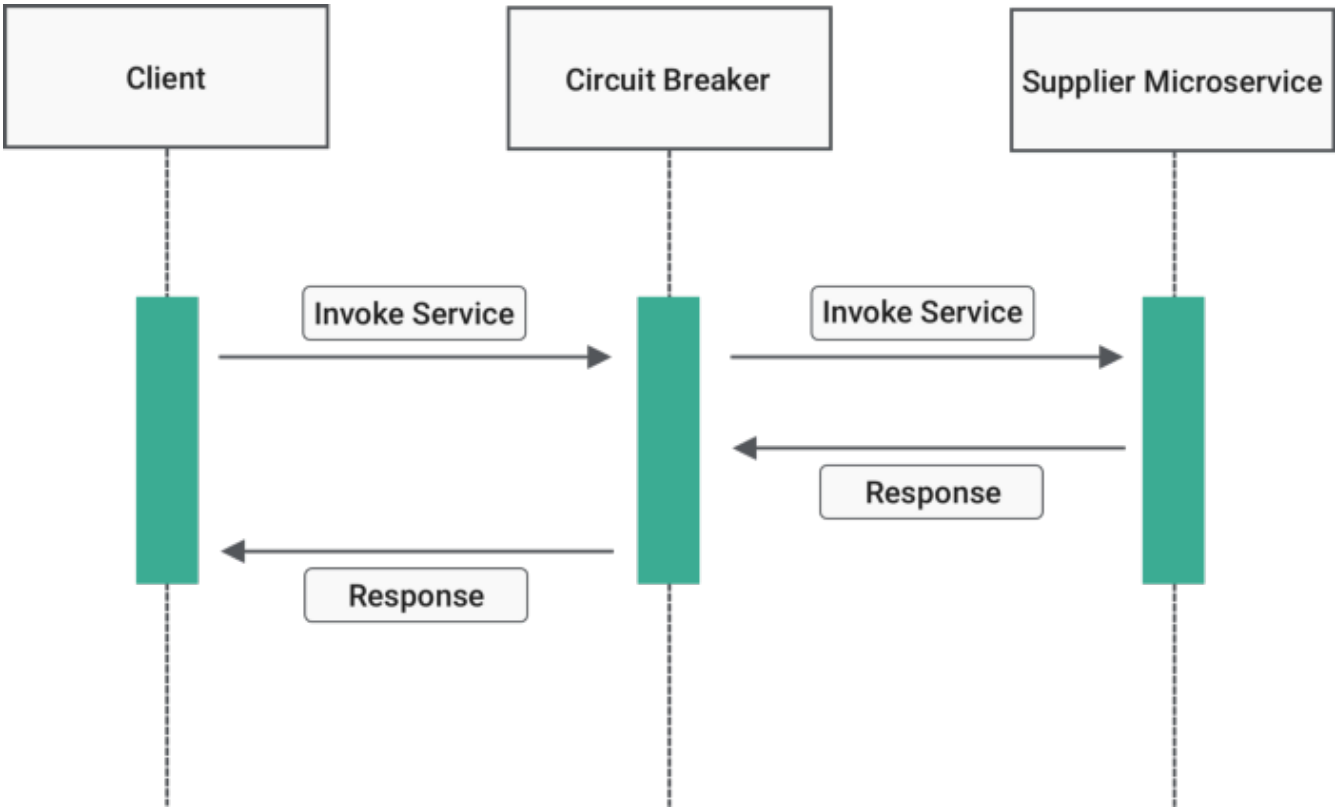

Figure 24. Closed circuit breaker

In this example, the circuit breaker is closed, and the service can be called. Until the service succeeds, the circuit breaker remains closed.

The following diagram shows a simple example of an open circuit breaker:

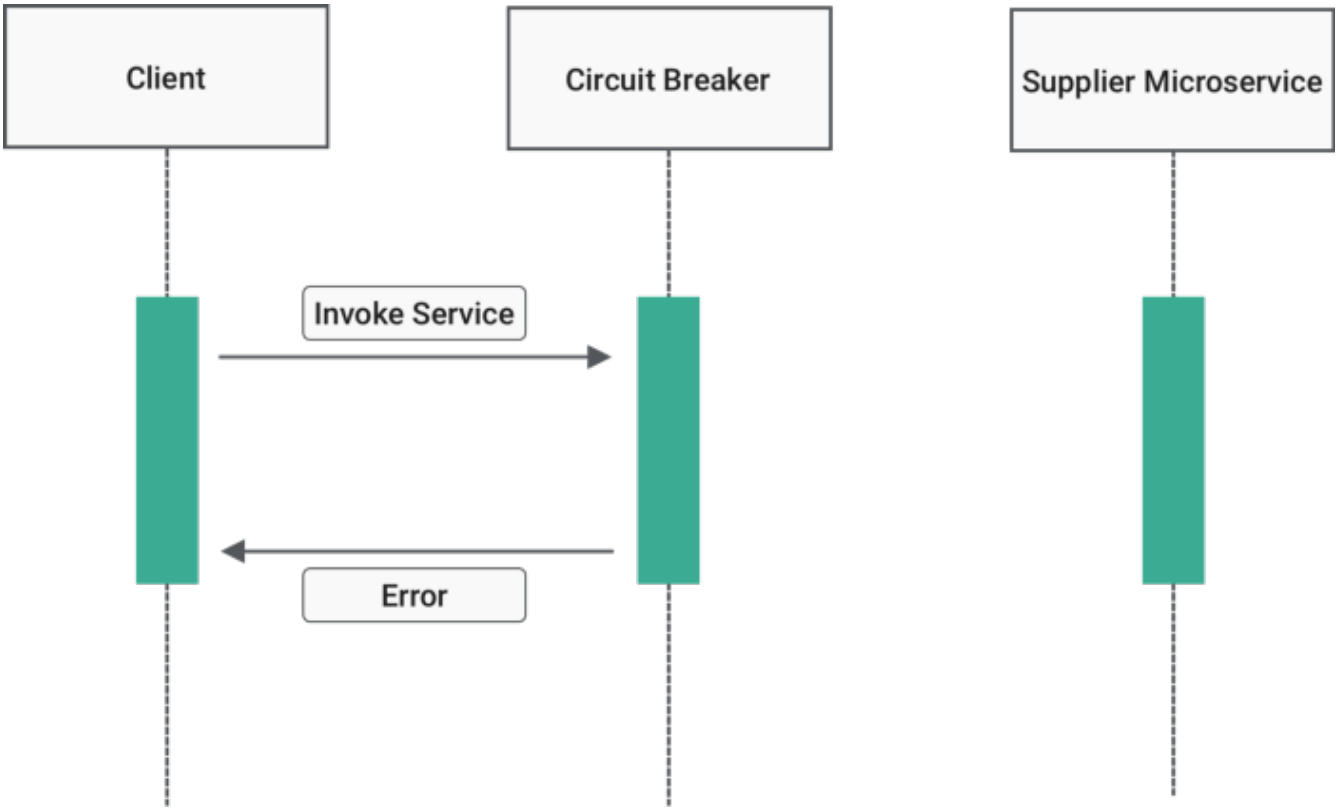

Figure 25. Open circuit breaker

In this example, the circuit breaker is open, and the service cannot be called. The circuit breaker

will remain open for a fixed amount of time, or until a certain number of calls have been made. When the circuit breaker is open, the client receives an error message by the circuit breaker.

3.2.3. Bulkhead

The bulkhead pattern is a mechanism that allows a service to limit the number of concurrent connections it can handle. It is used to prevent cascading failures in a distributed system. It applies when multiple applications needs to connect to a component by requesting a connection to that component.

The following diagram shows a simple example of the bulkhead pattern:

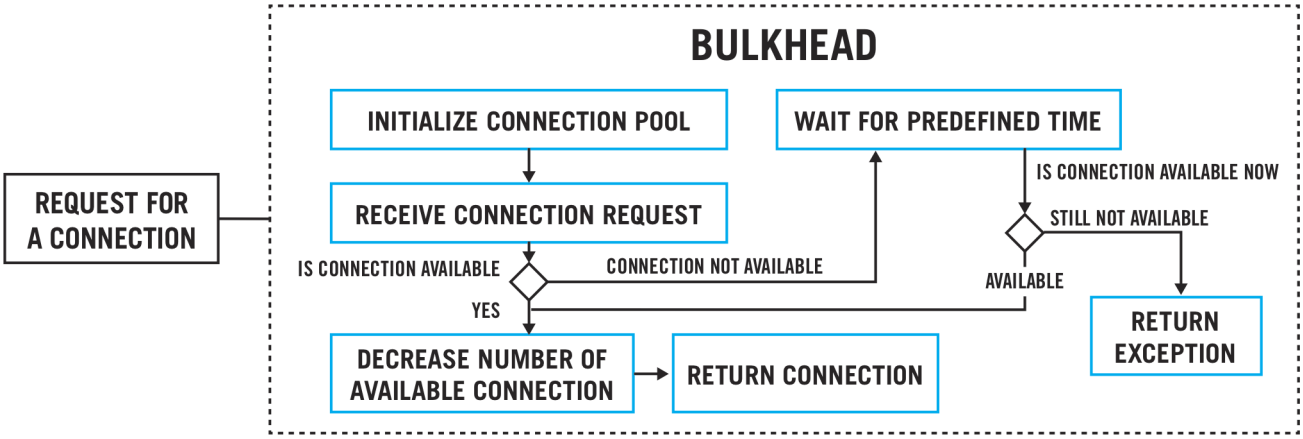

Figure 26. Bulkhead pattern

This pattern works as follows:

- A request for a wen connection is made.
- The bulkhead will check is the connection to the requested component is availiable to serve the request.
- If the connection is availiable, the bulkhead will serve the request.
- If the connection is not availiable, the bulkhead will wait for a pre-defined time interval.
- If any connection becomes availiable during this period, it will be allocated to serve the waiting request.
- If no connection becomes availiable during this period, the bulkhead return an exception.

3.2.4. Sidecar

The sidecar pattern is a mechanism that allows a service to add additional functionality to a microservice, such as logging, monitoring, or authentication. It is typically implemented as a separate process that runs alongside the microservice, such as a service proxy.

The following diagram shows a simple example of the sidecar pattern:

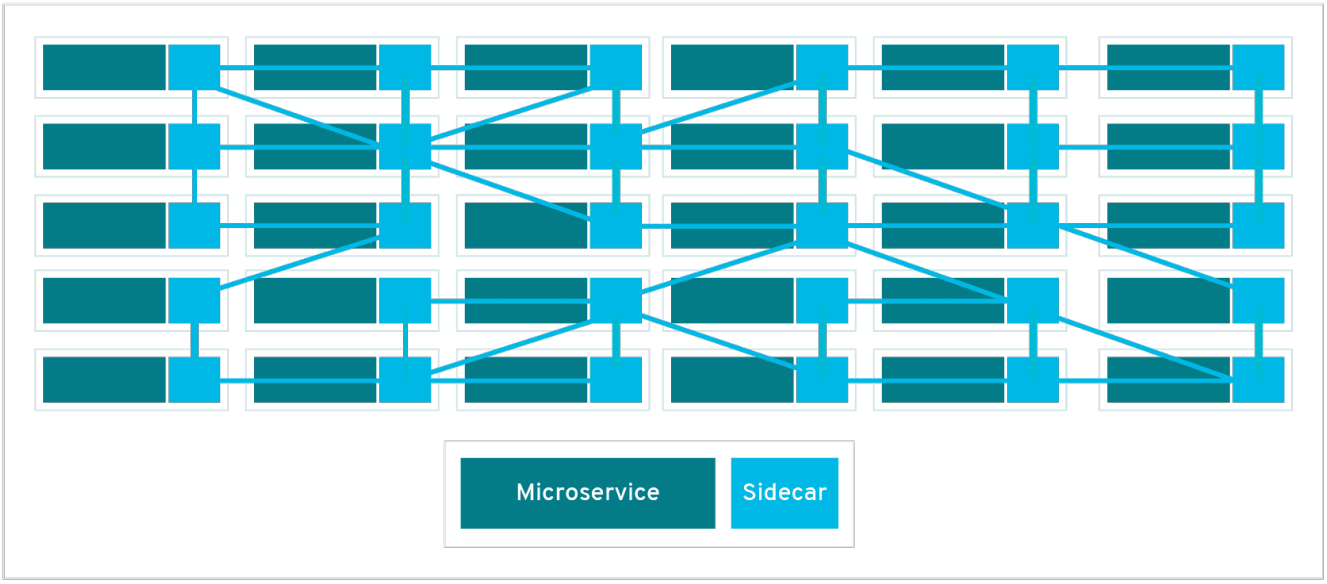

Figure 27. Sidecar pattern

With this pattern, each microservice is composed of two parts: the microservice and the sidecar. When a request is made from or to the microservice, the sidecar intercepts the request and adds additional functionality, then forwards the request. Let's explain in detail how it works.

The following diagram shows the traffic flow when a request is made to the microservice, with a service proxy sidecar:

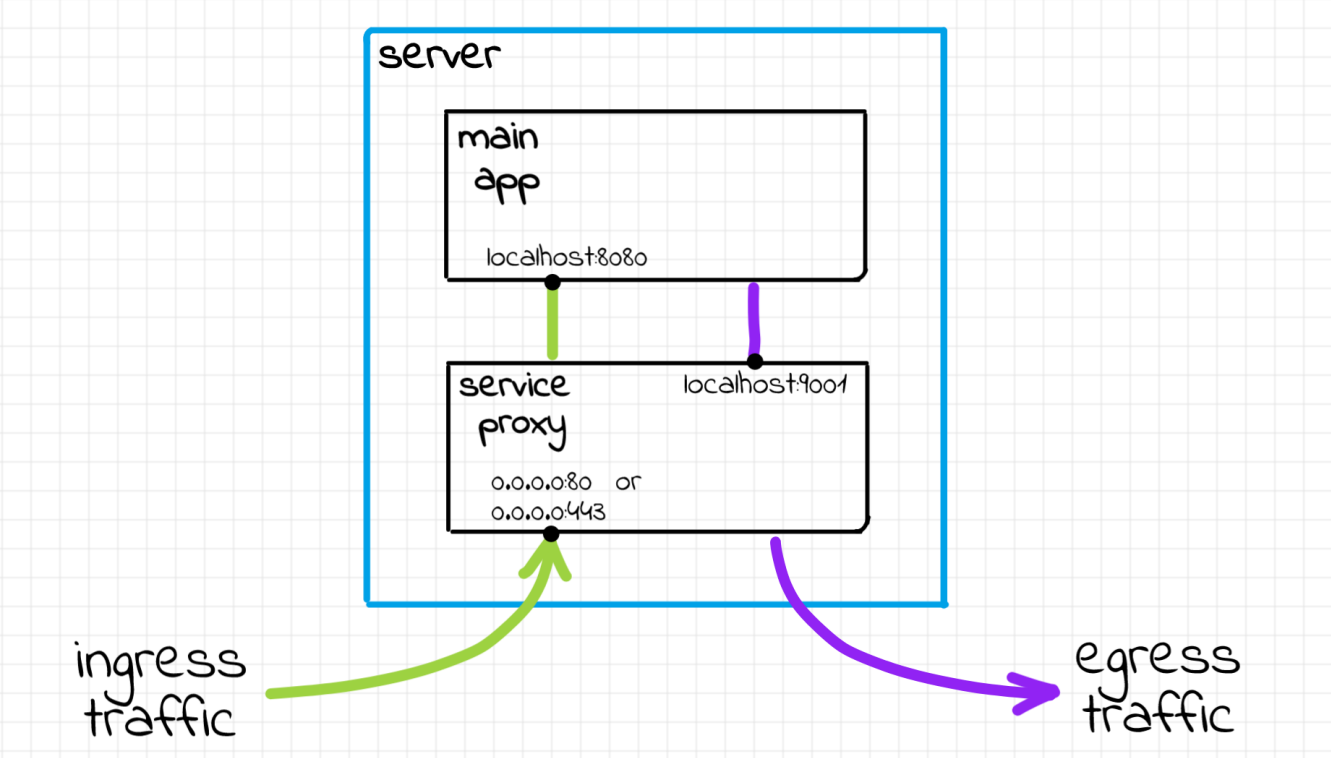

Figure 28. Sidecar pattern - request to microservice

When a request is made to the microservice, the service proxy intercepts the request (and can adds additional functionalities), then forwards the request to the microservice. The microservice processes the request, and returns the response to the service proxy (and can too adds additional functionalities), which forwards it to the client.

3.2.5. Service mesh

A service mesh is a dedicated infrastructure layer for handling service-to-service communication. It works with a sidecar proxy that is deployed alongside each service. The sidecar proxy intercepts all network communication between microservices, and adds some functionnalities.

By this way, each microservice does not have to be coded with inter process communication logic, but only with the business logic. The service mesh is responsible for the inter process communication logic.

The most popular service mesh is Istio. It is an open source service mesh that provides traffic management, policy enforcement, and observability. Istio provide all following features:

- Secure service-to-service communication in a cluster with TLS encryption
- Strong identity-based authentication and authorization
- Automatic load balancing for HTTP, gRPC, WebSocket, and TCP traffic
- Fine-grained control of traffic behavior with rich routing rules, retries, failovers, and fault injection
- A pluggable policy layer and configuration API supporting access controls, rate limits and quotas
- Automatic metrics, logs, and traces for all traffic within a cluster, including cluster ingress and egress

Here is a schema of the Istio service mesh, with the envoy proxy sidecar:

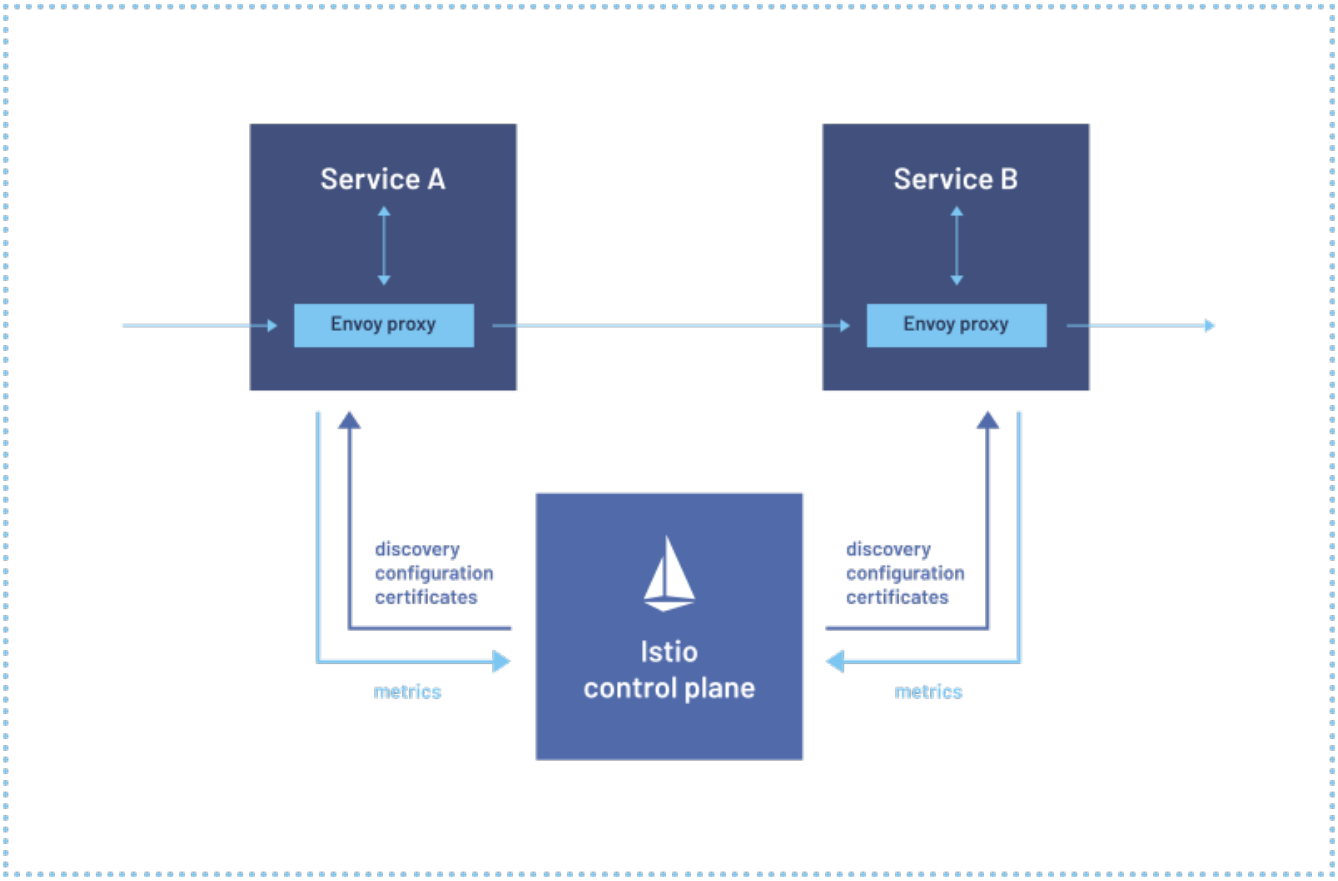

Figure 29. Istio service mesh

3.3. Service discovery

This part is principally based on the following article: Service Discovery in a Microservices Architecture

Service discovery is the process of automatically detecting the location of a service. It is used to allow a service to find other services without having to know their location. This problem appeared because in microservice architecture, the different services are deployed on different servers, and the location of a service can change over time because of server failures, autoscaling or load balancing.

Service discovery is typically implemented using a service registry, such as Consul, etcd, or ZooKeeper. These systems are typically deployed as a separate service, and they can be used by multiple applications.

There is two main types of discovery patterns: Client-side and Server-side. Before explain those patterns, let's define what is a service registry.

3.3.1. Service registry

A service registry is a key part of service discovery. It's a database that contains the network location of all the services in a distributed system. He must be highly available. Some softwares implements this functionality, such as Consul, etcd, ZooKeeper, or Netflix Eureka.

Note that some systems such Kubernetes, Marathon and AWS have their own service registry implementation.

In those solutions, there are two common patterns for a service to register itself to the service registry: Self-registration and Third-party registration.

Self-registration

In self-registration, the service is responsible for registering and deregistering in the services register and can send heartbeat if necessary. This pattern is the most simple one and is used in many systems.

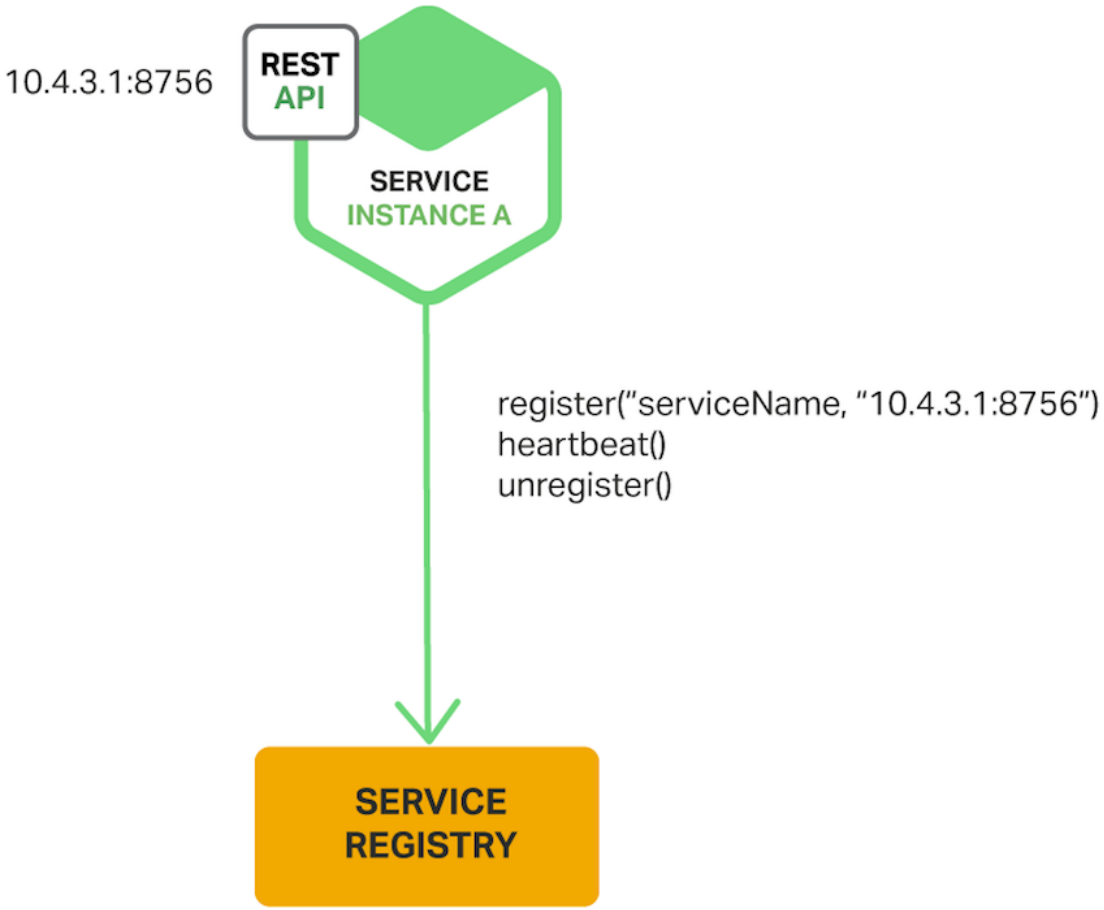

Figure 30. Self-registration

The real benefit of this pattern is its simplicity. You don't need any other system component.

Third-party registration

In third-party registration, a specific service is responsible to register all services in the service registry. This service is called a registrar. The registrar, implemented as a separate service, should be highly available.

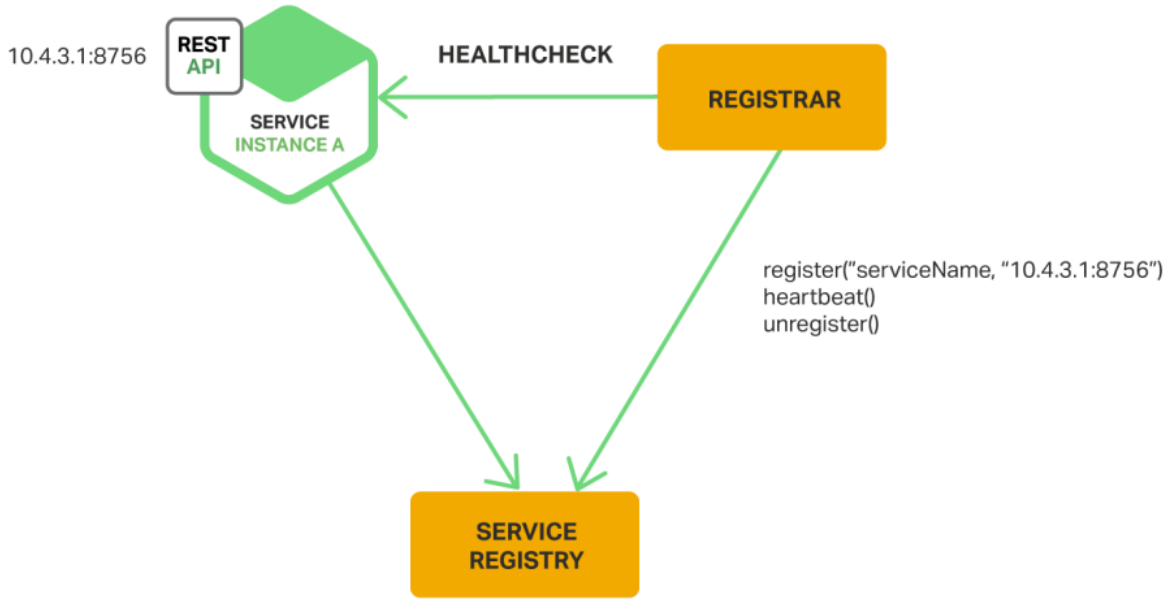

Figure 31. Third-party registration

The benefits of this pattern are:

- All services are decoupled from the service registry.
- A service does not have to implement register logic.

The main drawback of this pattern is that it requires an additional service, the registrar.

Consul

Let's see how Consul implements the service registry.

Consul is a service networking that offers some features. Here, we will speak about the service discovery feature, that allows to register services and to query them, in a kubernetes cluster. It is a third-party registration.

Consul is made up of two parts:

- `Consul server`: The consul server is made up of some server agents (a leader and some followers), that are responsible to store all services states. Consul follows the consensus protocol to ensure that all agents have the same state, and be fault tolerant.
- `Consul client node`: A consul client is made up of two parts: the consul client agent, that is responsible to register, deregister and query the services, and the proxy, that makes the bridge between the agent and the service. Generally, there is one consul client agent per consul client node. The consul client node can be a kubernetes node, or a VM.

The services does not need to know the location of the others services, and can query them using a domain name. The proxy is responsible to resolve the domain name to a service location, and to forward him the request.

Here is the schema of the service discovery with Consul:

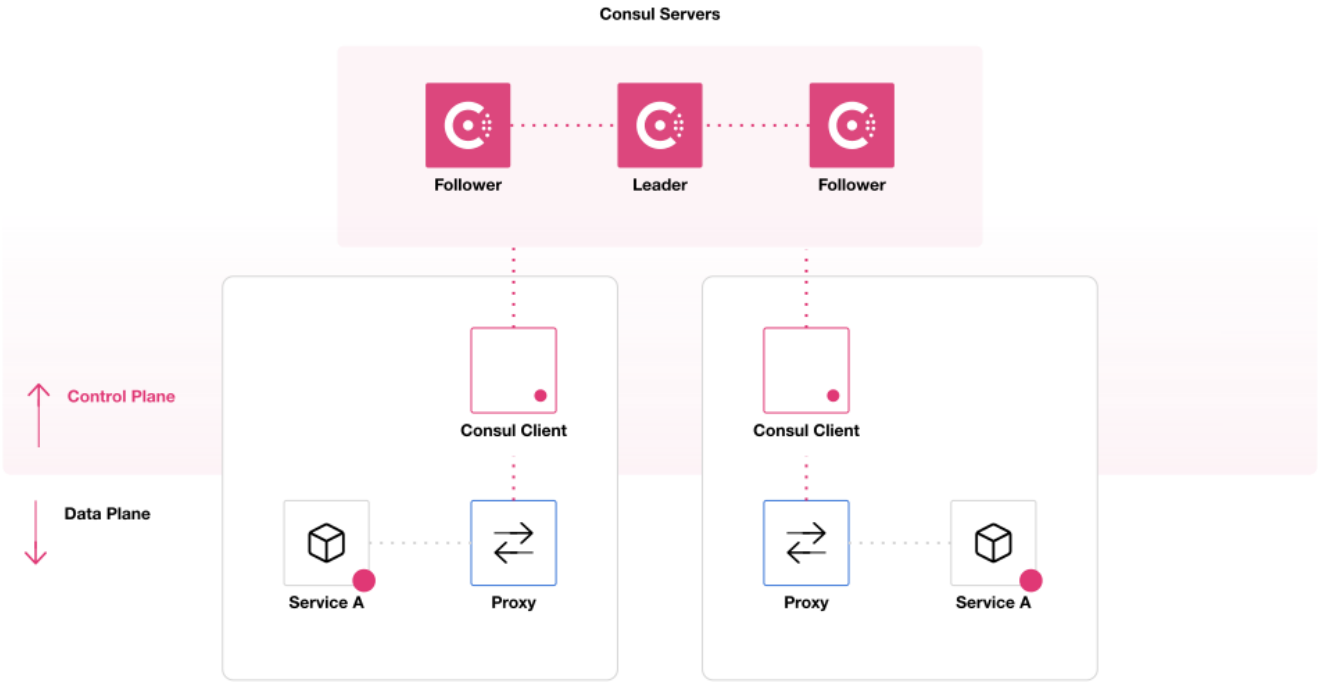

Figure 32. Service discovery with Consul

It is the basic functioning of Consul. In a kubernetes cluster, the consul client agent is not necessary, because the kubelet can replace it.

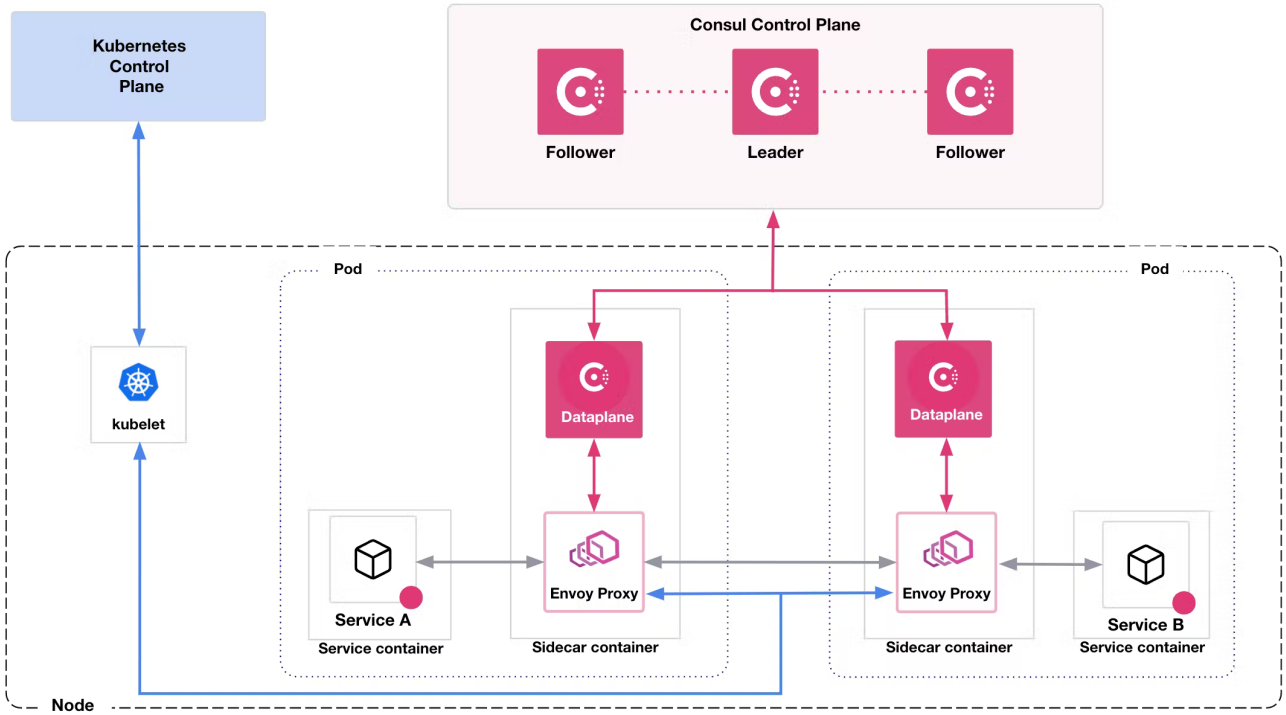

Figure 33. Service discovery with Consul in a kubernetes cluster

In this schema, the consul server is deployed in a kubernetes cluster. The consul client agent is replaced a unique sidecar container, that is deployed in each pod. The sidecar container is responsible to register, deregister and query the services and is made up of two parts: a dataplane

and a envoy proxy.

Consul uses the gossip protocol to share the services states between the client agents and the server agents. The gossip protocol is a peer-to-peer protocol, that means that each node is responsible to share the information with the others. Here is a schema of the gossip protocol:

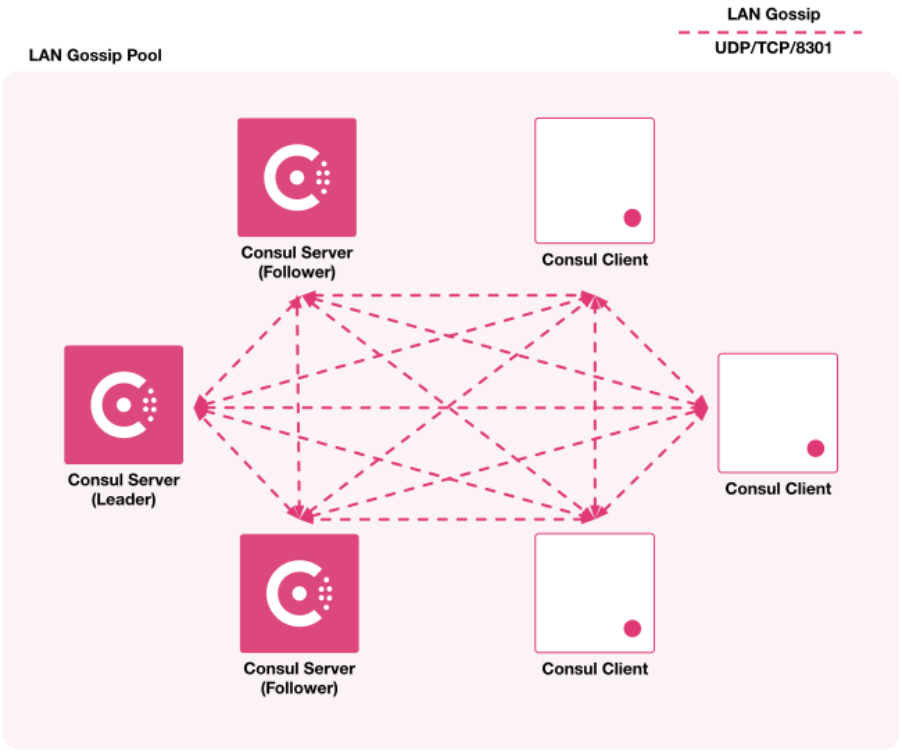

Figure 34. Gossip protocol with consul

Every few seconds, each consul client agent and server will send a heartbeat to one of its peers, chosen randomly. By this way, the global state of the cluster is shared between all nodes of the cluster. This protocol is based on UDP, so it is very efficient, and it is fault tolerant. It is also very scalable, and reduces the load on the network bandwidth.

3.3.2. Client-side discovery

In client-side discovery, the client is responsible for discovering the location of the other services by query the service registry.

The following diagram shows a simple example of client-side discovery:

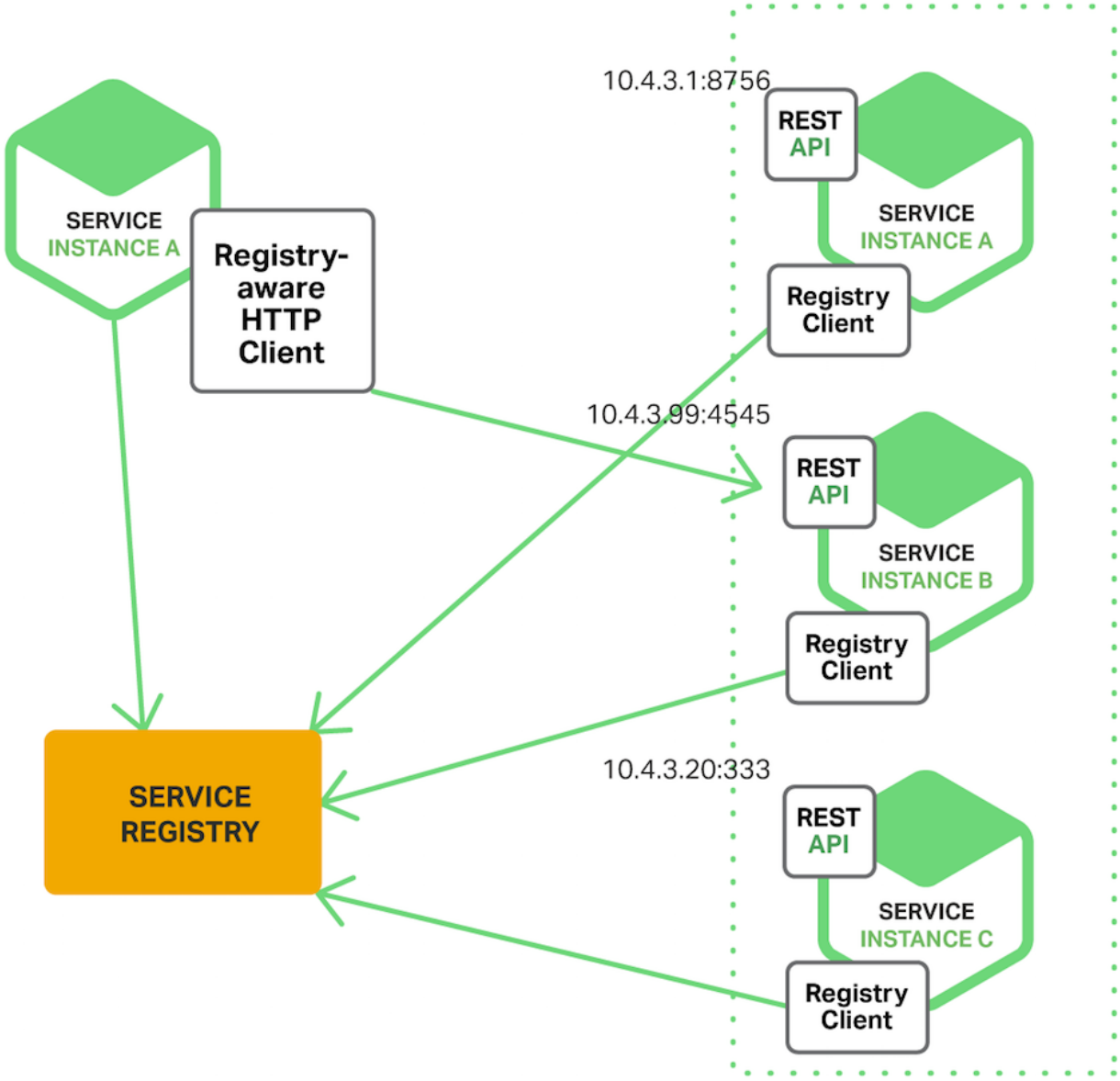

Figure 35. Client-side discovery

in this schema, each service, once started, will register itself to the service registry.

The client will then query periodically the service registry to get availiable locations of the services. If a service is no longer available, the service register will delete its list of available services, then after the next periodic updating of the customer, it will no longer try to connect to this service.

3.3.3. Server-side discovery

In server-side discovery, the server (load balancer) is responsible for discovering the location of the other services by query the service registry. A load balancer is a server that distributes network or application traffic across a number of services.

The following diagram shows a simple example of server-side discovery:

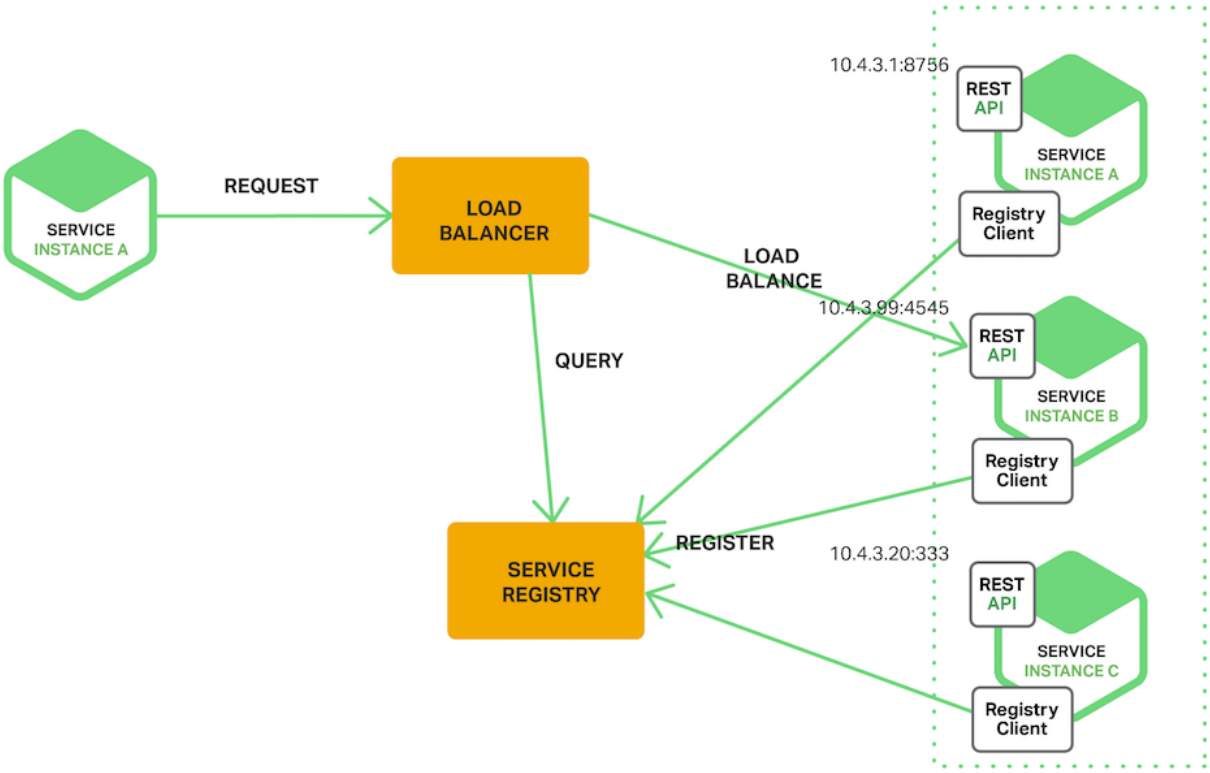

Figure 36. Server-side discovery

In this schema, as well for the client-side discovery, each service, once started, will register itself to the service registry.

The load balancer will then query the service registry periodically to get availiable locations of the services. If a service is no longer available, the service register will delete its list of available services, then after the next periodic updating of the load balancer, it will no longer try to connect to this service.

The server-side discovery has several benefits:

- All details of service discovery is abstracted for the client.
- Client needs only to query one service, the load balancer, to query the required service.
- This pattern is more scalable than the client-side discovery.
- This pattern is easier to use because some softwares already implement this pattern (and for free), such as Nginx, HAProxy, Varnish, etc.

3.4. PolyCode integration

For our application, PolyCode, we have different types of microservices. There is the microservice cutting scheme presented in part 1 of this document.

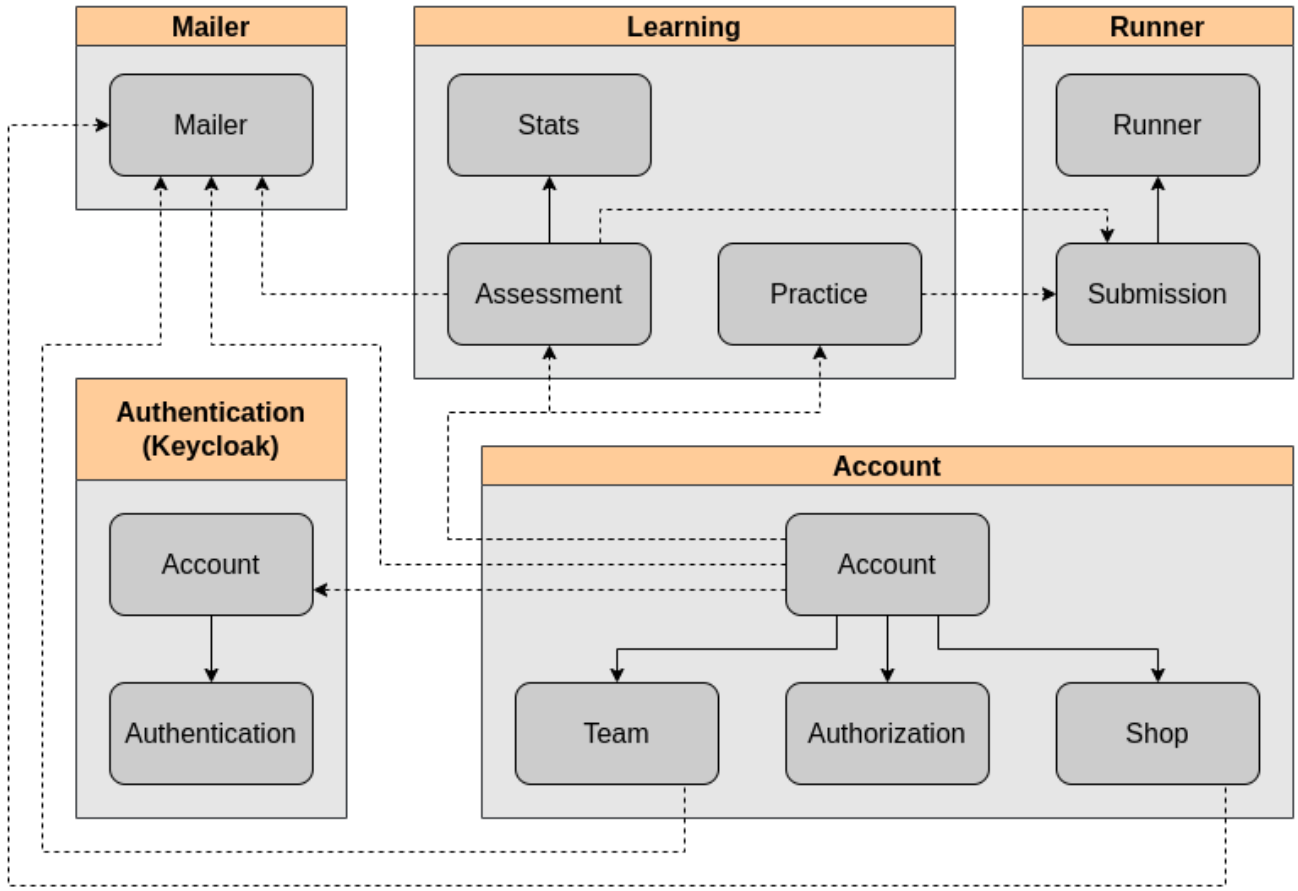

Figure 37. PolyCode microservice architecture schema

The five microservices are:

- `Mailer`
- `Authentication` (Keycloak)
- `Account`
- `Learning`
- `Runner`

In this part, we will see how we will manage the communication and the discovery between those microservices. Let's begin with the mailer microservice.

3.4.1. Mailer service communication

IMPORTANT	In the case of this exercise, we can't use a message queueing system, then we cannot consider the following proposition. In a real case, I think this is the best solution, that is why I present it here. You can skip this part if you want.

The mailer service is responsible to send emails to users or candidates. It is a service that is used by others. For example, when a user is invited in a team, the `Account` service will call the `Mailer` service to send the email.

This use case does not need a response from the `mailer`, because if a letter is not received, there is still a way to send it again. Especially since we would have no guarantee than the mail was well

received and read.

Therefore, we will use asynchronous communication for this service. Since we only have this service that uses it, we will not use pub/sub pattern, but a simple queueing system.

In our previous example, the `Account` service will send a message to the event queue, and the `Mailer` service will consume this message, and send the email.

The benefits of this pattern are:

- The `Account` service does not have to wait for the `Mailer` service to send the email.
- The `Mailer` service can consume the messages in the queue at its own pace.
- We can scale the `Mailer` service easily by adding multiple consumers, depending on the queue size.

The drawbacks of this pattern are:

- If the scale is not well managed, the queue can grow indefinitely and cause some problems, and the mails will not be sent in a suitable time.

The chosen the queueing system RabbitMQ. There is no particular reason for this choice, except that it is the most widely deployed open source message broker.

Here is the schema of the `Mailer` microservice communication with RabbitMQ message broker:

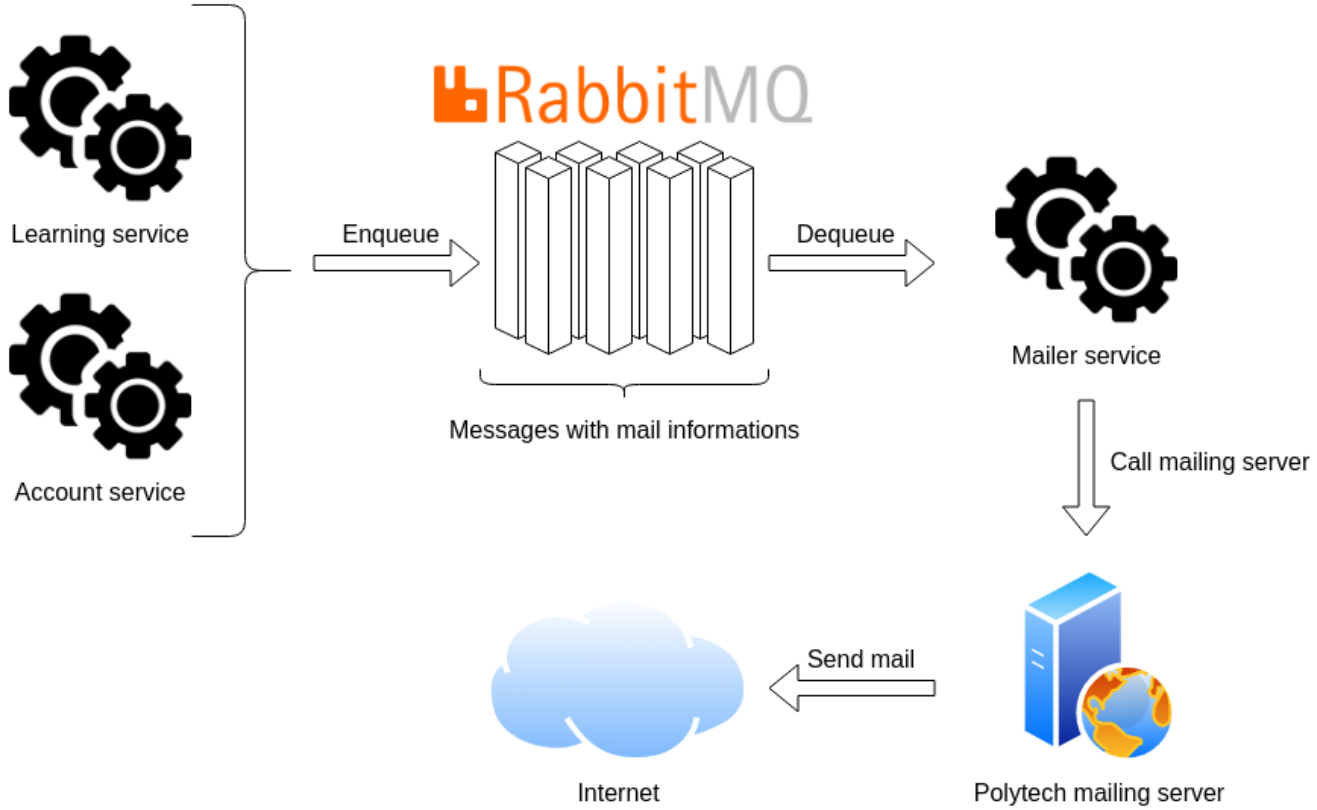

Figure 38. Mailer microservice communication

3.4.2. Others services communication

For the other microservices, we will use synchronous communication. The reason is that we need a response from the service. For example, when a user query his profile data to the `Account` service, a call to the `Authentication` service is made to check token validity. In this case, we need a response before sending the data to the user.

For all our microservices, we will use a REST API. The reason is that it is a well-known pattern, and it is easy to use. We don't need to use a more complex protocol such as gRPC, because we don't need to optimize the communication between our microservices for now.

3.4.3. Service discovery

For the service discovery, we will use the third-party registration pattern. We will use the `Consul` service registry. The reason is that it is a well-known service registry. We will use the self-managed version of Consul (open source) with kubernetes, because the PolyCode application is not deployed in a cloud environment, but on private servers, at Polytech.

Here is the schema of the PolyCode service discovery with Consul:

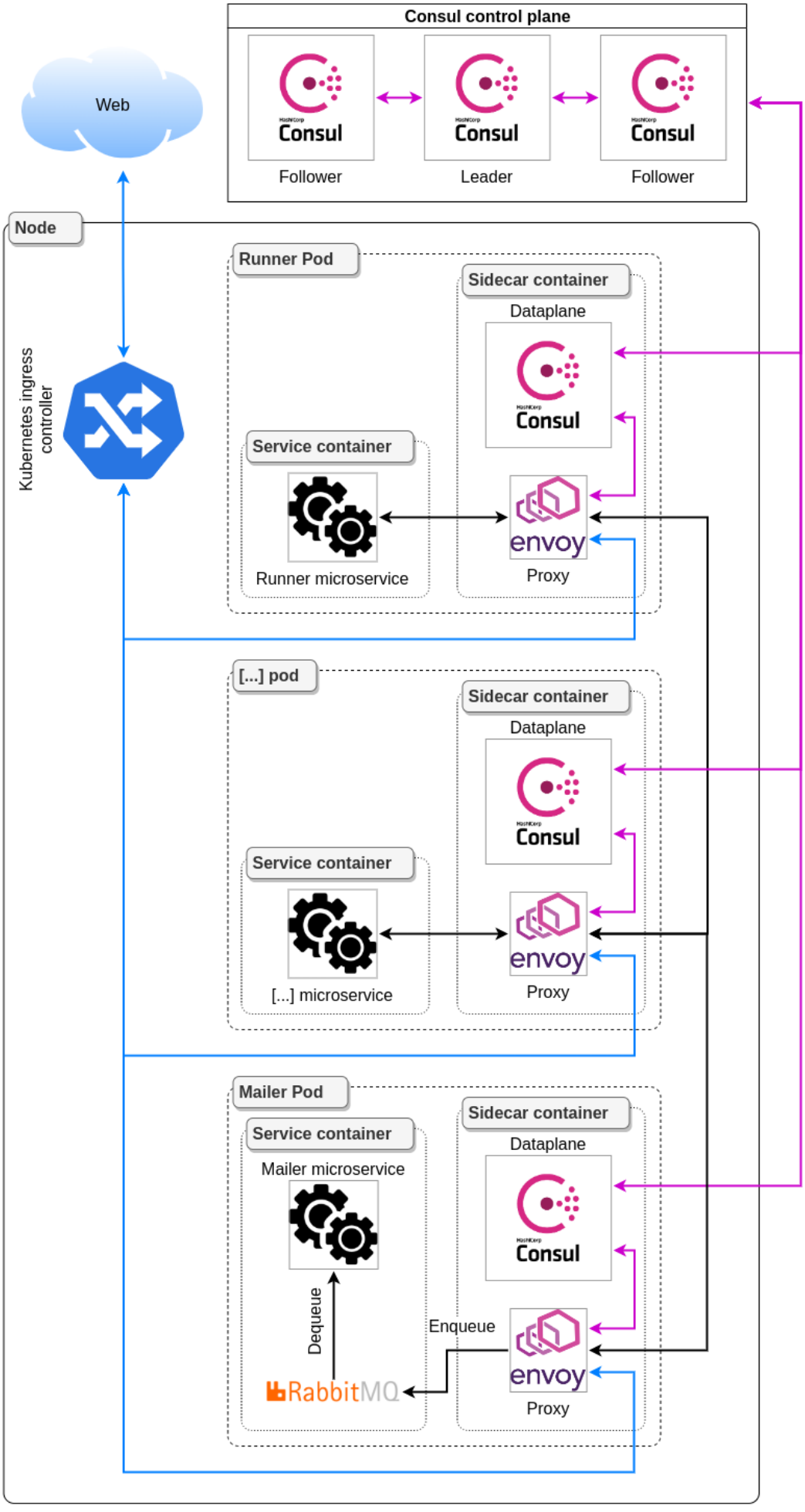

Figure 39. PolyCode service discovery with Consul

In this schema, we can see that each microservice is deployed in a kubernetes pod. Each pod has the microservice and a sidecar container, that is responsible to register, deregister and query the service. The sidecar container is made up of two parts: the dataplane and an envoy proxy. The envoy proxy is responsible to communicate with the service, and make load-balacing, and the dataplane with the consul control plane. The consul control plane is responsible to keep the global state of the cluster.

3.4.4. Sequence diagram of a basic request

Here is a sequence diagram of a basic request, from the `Account` service to the `Authentication` service, to verify the token validity:

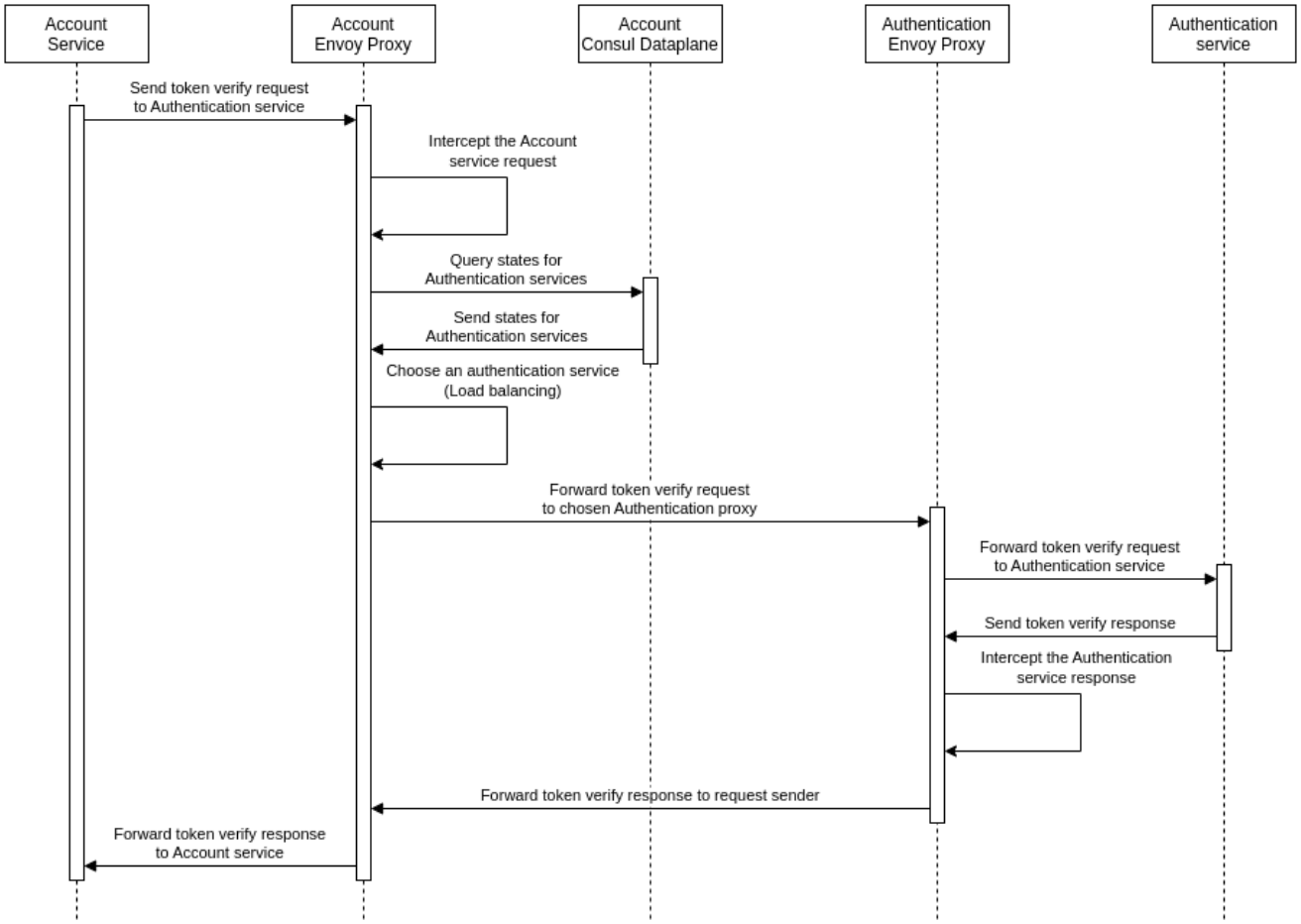

Figure 40. Sequence diagram of a basic PolyCode request

Chapter 4. Data Management

The data management concerns the way the database is managed, and how the data is stored. In microservice architecture, the microservices are independent, and they don't share the same database. Because of this, some new problems can appear and need to be solved. Let's see them, then how to solve them. After that, we will see how the microservice migration affects the data organization.

4.1. Data Consistency

In a traditional monolithic architecture, the data is stored in a single database, and it guarantees the data consistency by using transactions. In a microservice architecture, the data is stored in different databases, and it is not possible to use transactions. This means that the data consistency is not guaranteed.

Here is an example of a flow that cause a data inconsistency:

[Inconsistent Data] | *images/Inconsistent Data.png*

Figure 41. Inconsistent Data

This problem is the most complex problem to solve in a microservice architecture, and can cause a lot of problems, such as data loss, or data corruption.

There are several ways to solve it, such as the Saga pattern, or the Event Sourcing pattern, but they are not allowed in our case. We can limit the problem by following some rules, like validate the data editing only after the resonse of all others services, but we can't solve it completely; A called service can edit data, and the caller service can fail before validating the data. This means that the data is not consistent between this two services.

The best we can do is manage data inconsistencies, which requires additional development.

4.2. Data Backup

An important part of data management is to backup the data. It is necessary to ensure that it is not lost. As said before, in a microservice architecture, the data is stored in different databases, and it is not possible to backup all the data at the same time. This means that it is necessary to backup each database separately.

The backup can be done in several ways, such as a snapshot, or a dump. The snapshot is a copy of the database at a specific time, and the dump is a copy of the database at a specific time, but in a sql file. The snapshot is faster, but the dump is more portable.

The problem here is again data consistency. We need to ensure that the backup is made at the same time for all databases, to ensure that the data is consistent. This can be done by using a tool, such as `pg_dumpall`, which is a tool to dump all databases at the same time, but it is not possible is your databases are not on the same instance.

4.3. PolyCode Data Migration

The PolyCode data organization changes is caused by the microservice migration. This new organization is more complex, and it requires more development.

Unfortunately, this refactoring has not been done yet.

Chapter 5. Tracing

Tracing is the ability to follow the flow of a request at its different steps. It is useful to provide more observability, and to debug the application when a problem occurs.

In a monolithic architecture, all the requests are handled by the same application, then it is easy to trace them. You only need to implement a tracing or a log system, and to store it.

Nowadays, a microservice architecture, the requests are handled by different applications, and it is more difficult to trace them. You need to implement a tracing system in each application, and to send the traces to a central system to store them.

Let’s see how tracing works in a microservice architecture.

Chapter 6. Data Management

The data management concerns the way the database is managed, and how the data is stored. In microservice architecture, the microservices are independent, and they don't share the same database. Because of this, some new problems can appear and need to be solved. Let's see them, then how to solve them. After that, we will see how the microservice migration affects the data organization.

6.1. Data Consistency

In a traditional monolithic architecture, the data is stored in a single database, and it guarantees the data consistency by using transactions. In a microservice architecture, the data is stored in different databases, and it is not possible to use transactions. This means that the data consistency is not guaranteed.

Here is an example of a flow that cause a data inconsistency:

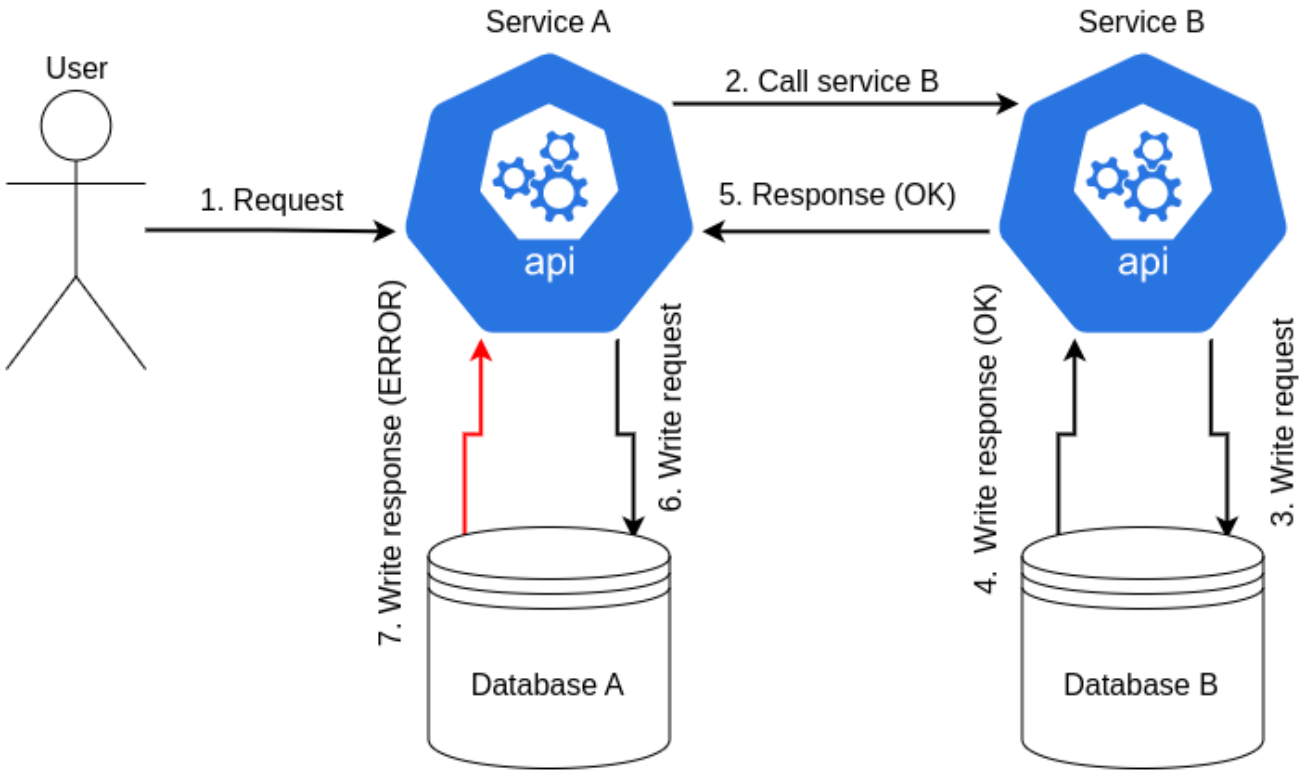

Figure 42. Inconsistent Data

This problem is the most complex problem to solve in a microservice architecture, and can cause a lot of problems, such as data loss, or data corruption.

There are several ways to solve it, such as the Saga pattern, or the Event Sourcing pattern, but they are not allowed in our case. We can limit the problem by following some rules, like validate the data editing only after the resonse of all others services, but we can't solve it completely; A called service can edit data, and the caller service can fail before validating the data. This means that the data is not consistent between this two services.

The best we can do is manage data inconsistencies, which requires additional development.

6.2. Data Backup

An important part of data management is to backup the data. It is necessary to ensure that it is not lost. As said before, in a microservice architecture, the data is stored in different databases, and it is not possible to backup all the data at the same time. This means that it is necessary to backup each database separately.

The backup can be done in several ways, such as a snapshot, or a dump. The snapshot is a copy of the database at a specific time, and the dump is a copy of the database at a specific time, but in a sql file. The snapshot is faster, but the dump is more portable.

The problem here is again data consistency. We need to ensure that the backup is made at the same time for all databases, to ensure that the data is consistent. This can be done by using a tool, such as `pg_dumpall`, which is a tool to dump all databases at the same time, but it is not possible is your databases are not on the same instance.

6.3. PolyCode Data Migration

The PolyCode data organization changes is caused by the microservice migration. This new organization is more complex, and it requires more development.

Unfortunately, this refactoring has not been done yet.

Chapter 7. Security

The security is a very important part in any software development, and that has always been.

In a microservice architecture, unlike a monolith, each service runs in its own process and may have its own unique security requirements. Additionally, because the services communicate with each other over the network, there is a greater risk of data breaches and other security incidents. Therefore, it is important to consider security as early as possible in the development process.

There is many ways to secure a microservice architecture, under several aspects. Here, we will try to cover the most important ones. You have to keep in mind that there is no ultimate solution, and you have to choose the right security solution for your specific needs.

7.1. Application level

The application security level concerns the code itself, and the way it is written. It is important to write secure code, and to follow the best practices. It is necessary to start that a developer will write code containing security failures. We cannot prevent this, but we can limit it in several ways. Let's see some of them.

7.1.1. Code Review

The code review is a very important part of the development process. It is a way to ensure that the code is written in a secure way, and that it follows the best practices. It is also a way to ensure that the code is written in a way that is easy to maintain, and that it is easy to understand. It is always good to have another point of view on written code.

Examination of the code can be carried out by someone in the same company, or by external developers, with security audits.

7.1.2. Dependency Analysis

Additionally, it is necessary to keep an eye on the dependencies used for the development of applications. security failures are frequently discovered, and security updates are then necessary.

There are tools to detect it automatically, such as Dependabot (GitHub), or Snyk.

7.1.3. Security Testing

Security testing is a way to ensure that the code written code works as expected. There are several ways to do it, and we will explain three of them.

Static application security testing (SAST)

SAST is a way to analyze the code statically, without executing it (only with the source code). It is a way to detect security failures in the code, and to ensure that the code follows the best practices. There are several tools to do it, such as Klocwork, or Checkmarx.

Dynamic application security testing (DAST)

DAST is a way to analyze the code dynamically, by executing it (without the source code). It is a way to detect security failures in the code, and to ensure that the code follows the best practices. Arachni, an open source tool, can be used for this purpose.

Here is a diagram showing where the SAST and DAST tests are done on the development process.

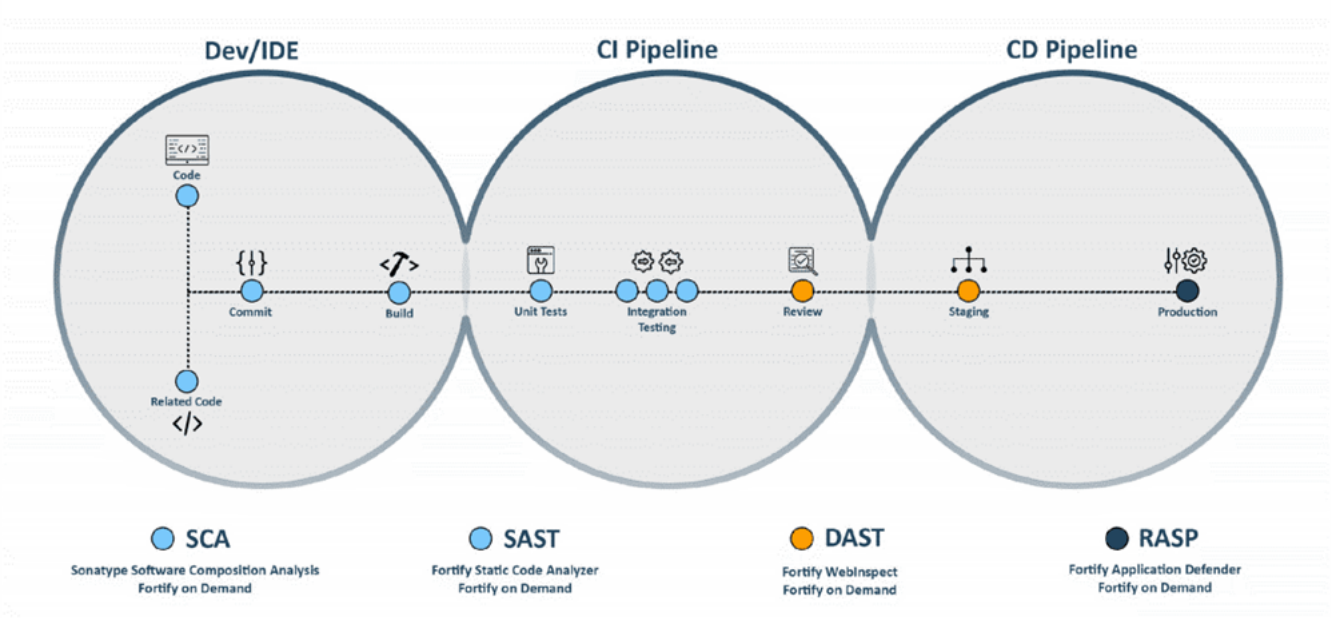

Figure 43. SAST and DAST tests in the development process

Stress Testing

Stress testing is a way to test the code under heavy load. It is a way to ensure that the code can handle a large number of requests. There are several tools to do it, such as WebLoad, LoadNinja, or JMeter.

This type of testing is important, because there is no other way to test the resistance of an application against a huge load.

7.2. Communication level

The communication level concerns the way the services communicate with each other. In a microservice architecture, each microservice communication is done through the network.

There are two types of communication in a microservice architecture: internal communication and external communication. Here is a diagram showing those two different types.

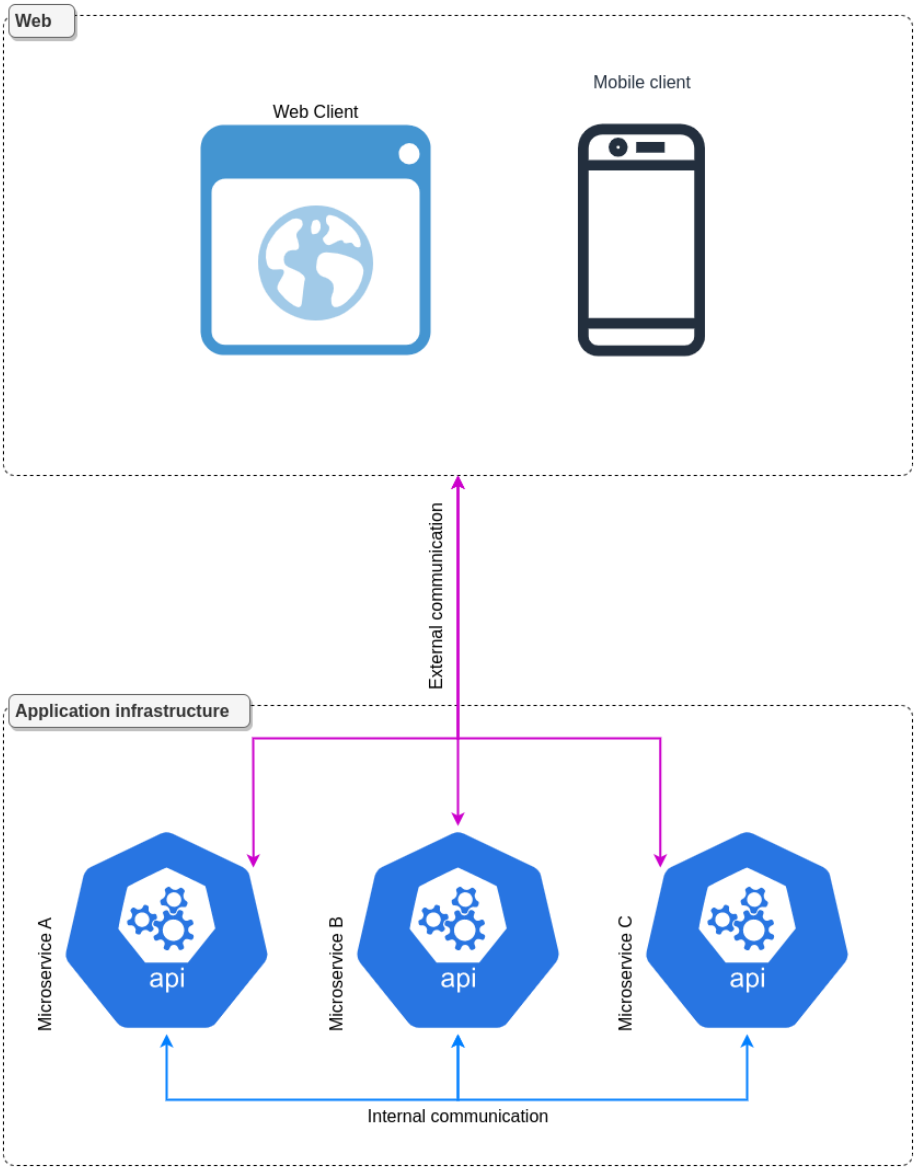

Figure 44. Internal and external communication in a microservice architecture

External communication is the communication between a microservice and the web. If a microservice is exposed to the web, it is strongly likely that he was attacked. Therefore, it is necessary to ensure that the communication between the microservice and the web is encrypted.

Internal communication is the communication between microservices. This communication is not exposed to the web, but it is still better to ensure that it is encrypted, if possible.

Let us first see what layers we can encrypt communication.

7.2.1. Gateway level

If you use a gateway, you can encrypt the communication between the gateway and the web. This is the most easy way to add encryption. You only have to add a TLS certificate to the gateway, then all the outgoing communication will be encrypted. Note that all internal communication will not be encrypted.

Here is a diagram showing the gateway level encryption:

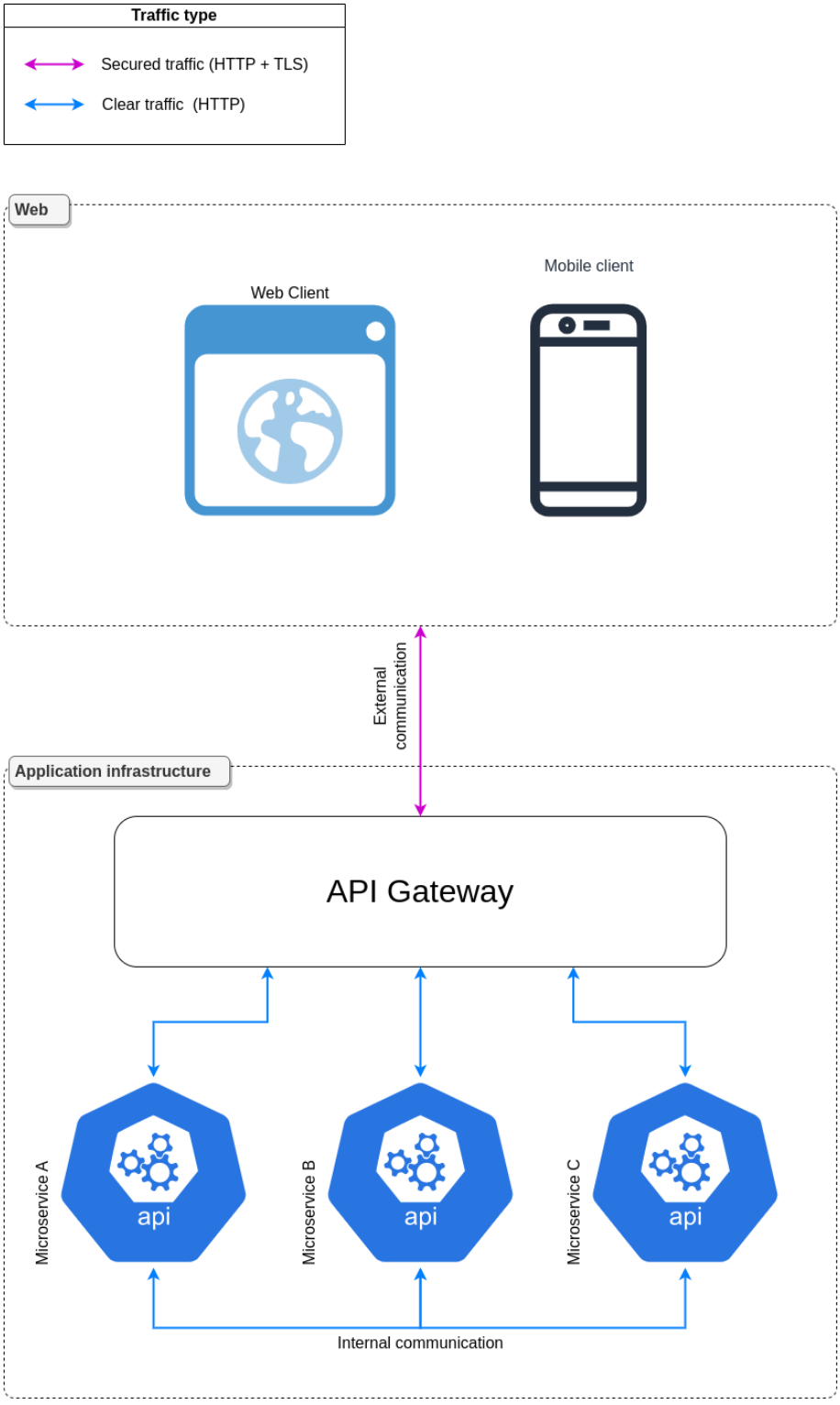

Figure 45. Gateway level encryption

Here, the gateway provide a TLS certificate to the web. The web will then be able to communicate with him in a secure way.

7.2.2. Sidecar level

You can also encrypt the communication between each microservice and the gateway (or the web, if you donc have a gateway). This is a more complex way, but it is also the most secure one.

Here is a diagram showing the sidecar level encryption:

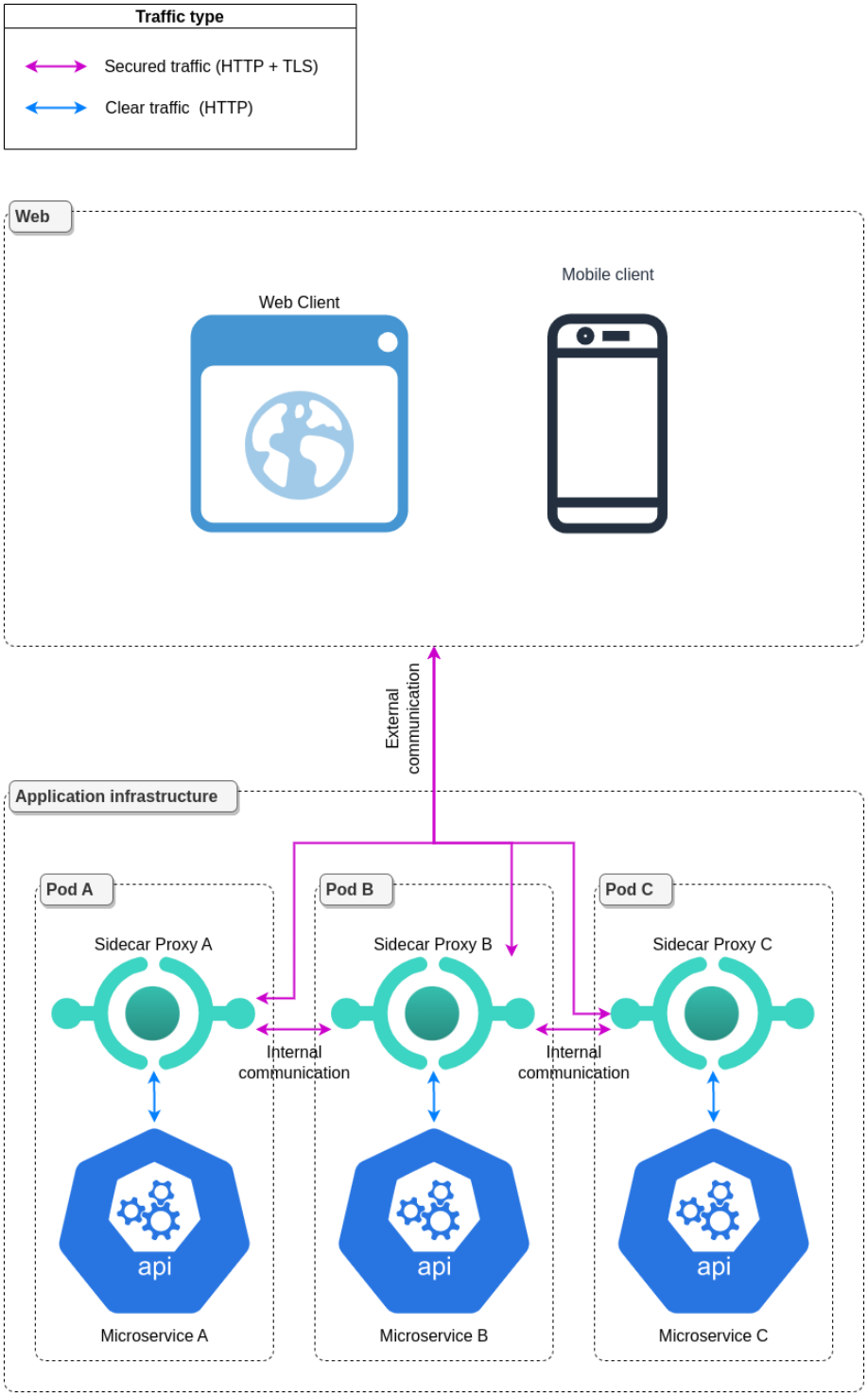

Figure 46. Sidecar level encryption

Here, each microservice have a sidecar that provide a TLS certificate. With this approach, all communication is encrypted (internal and external). The only exception is the communication between the sidecar and the microservice, but this communication is not exposed to any other network.

It's strongly recommended to use this approach, if possible.

7.2.3. Rate limiting

Rate limiting is a way to limit the number of requests that can be sent to a microservice. It is a way to prevent a service from being overloaded, or to prevent a service from being attacked, such as a

DDoS attack, or a brute force attack.

There are several ways to do it, such as the Nginx rate limiting module, or the Envoy rate limiting module.

7.2.4. Service Authentication and Authorization

Service authentication and authorization is a way to ensure that only known and authorized services can communicate with each other.

Authentication

Authentication is a way to ensure that the service is known. Here, we will see how to authenticate a service using OAuth 2.0.

> **NOTE** You can refer to the authentication part of this document to see more details.

A service, such as a user, can authenticate himself to the authorization server, and get an access token to communicate with other services. In this case, it's the Client Credentials Grant type that is used.

Here is a sequence diagram of this flow:

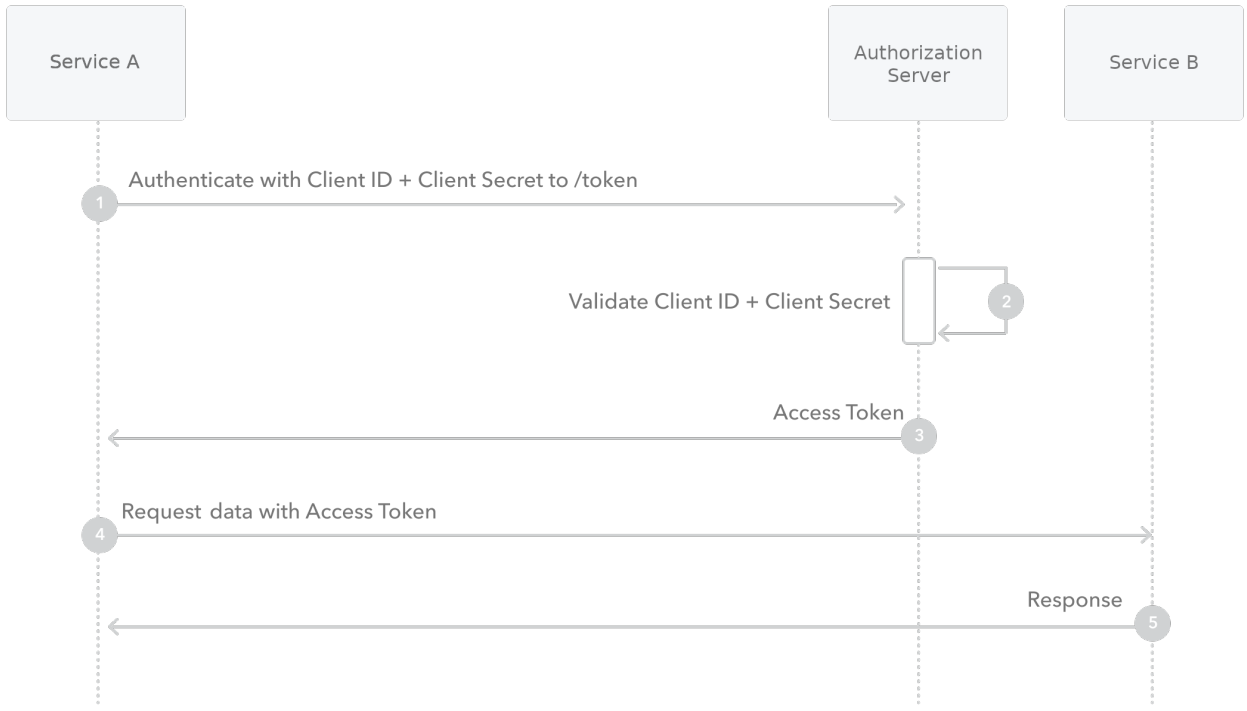

Figure 47. Client Credentials Flow

As you can see, this flow is very simple. Because the communication is encrypted, as explained in the previous section, the client secret is not exposed to the network.

Authorization

Authorization is a way to ensure that a known service has sufficient rights to communicate with another service. To do so, we will use the role based access control (RBAC) model.

This model is based on roles and permissions. A role is a set of permissions, and is assigned to a subject (can be a user, or a service). A permission is an action associated with a resource. For example, a subject can be assigned to the role named `admin`, who own the permission `read` on the resource `user`. This means that this subject can read the resource `user`.

With this model, we can apply the principle of least privilege. This principle is a way to ensure that a subject can only do what he needs to do. It is better to have a problem of lack of permissions, and to have to add it, rather than an excess of not necessary rights.

Generally, the authorization server is the one that will check the permissions of a subject.

7.3. Data Security

The data security concerns the way the data is stored. This is an extremely important part, because it is the most important asset of a company, and can cause a lot of problems about privacy, or even financial problems if they are stolen.

It is necessary to encrypt the sensitive data as much as possible.

First of all, all passwords must be hashed, and that with a strong salt. Basically, this is already done by the authentication server, which is the one that stores the passwords.

To encrypt the sensitive data (who needs to be read, not as a password), the cloud provider can provide a way to do it, or if you manage your own database, you can use its feature (if it exists).

The cons of this approach is that it is not possible to search in this data, and the encryption and decryption process can be very slow.

7.4. Secrets Management

Secrets are sensitive data, such as passwords, tokens, certificates, etc. It is necessary to manage them correctly, and to ensure that they are not leaked. There are several ways to do it.

In a microservice architecture, we need to centralize the secrets management, to ensure that all services have the same secrets, and get them from a secure way, so that they are not leaked. There is some solutions to do it, like HashiCorp Vault, Azure Key Vault, or Kubernetes Secrets.

Here, because the chosen solution to PolyCode is HashiCorp Vault, we will explain how it works.

7.4.1. HashiCorp Vault

HashiCorp Vault is a tool for securely storing and accessing secrets. It provides a centralized platform for managing sensitive information. Vault can also be used to generate dynamic credentials for services and applications, and to control access to those secrets through role-based access controls.

Here is a diagram showing the flow of HashiCorp Vault:

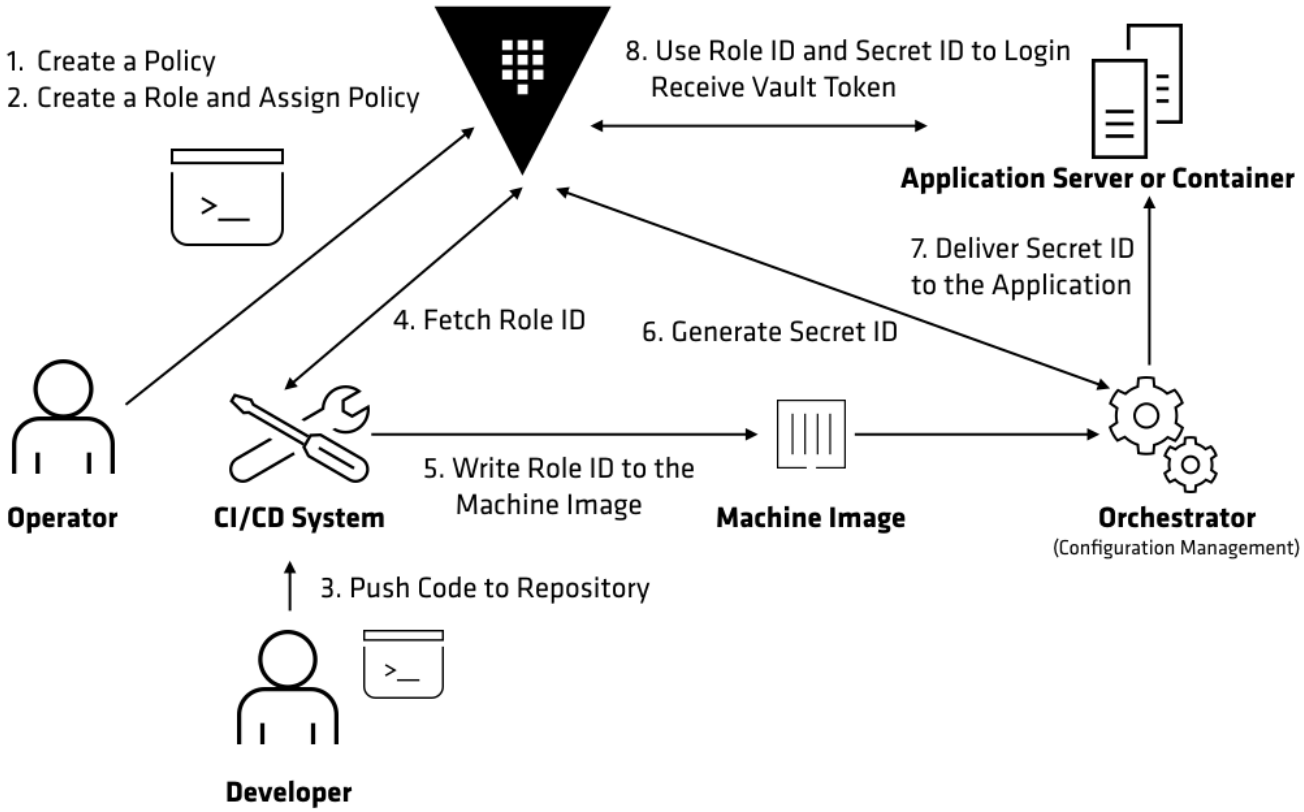

Figure 48. HashiCorp Vault flow

As you can see, a service can get the secrets from Vault, using its own credentials. Vault will then check if the service is authorized to get the secrets with the RBAC, and if it is, it will return them.

7.5. PolyCode Integration

For PolyCode, we will use some of the solutions explained before:

- `TLS everywhere`: We will use the sidecar level encryption, as explained before. This will ensure that all communication is encrypted, and that the data can't be intercepted.
- `Service Authentication and Authorization` : We will use the Client Credentials Grant type, with our own Keycloak authorisation server, describe in the authentication part. This will ensure that only known services can communicate with each other.
- `Service Authorization`: Additionally, we will use the RBAC model, to ensure that the service can only do what it needs to do.
- `Data Encryption`: We will hash the passwords. For the other data, we will not encrypt them, because it will be too complicated and not necessary for now.
- `Secrets Management`: We will use HashiCorp Vault, to manage the secrets.

Glossary

Microservices architecture

An architecture where the system is divided into multiple services that communicate with each other.

Actors

It is a person that interacts with the system. It can be a user, a system, a third party, etc.

User story

Informal, short and clear description of a feature from the perspective of the user. It takes the following format: `As a [actor], I want [feature], so that [benefit]`.

Functionnal quarter

A functionnal quarter is a set of user stories that are related to a same feature.

Bibliography

Authentication

OpenID Connect - curity.io
Oauth 2.0 with PKCE - postman.com
PKCE - loginradius.com
[FR video] Oauth 2.1 simply explained
[EN video] Oauth guide
OAuth 2.1 - fusionauth.io
Short explanation of OIDC - ibm.com
OpenID Connect - connect2id.com
JWT - jwt.io
JWT - ibm.com
Standard claims - openid.net
OpenID Connect endpoints - connect2id.com
Keycloak - heiswayi.nrird.com
Keycloak get started (docker) - keycloak.org
Active Directory - microsoft.com
Active Directory: Provision a user - microsoft.com

IPC

IPC in Microservices - vishnuch.tech
IPC in Microservices - nginx.com
What is a REST API? - Red Hat
REST Architectural Constraints - restfulapi.net
What is gRPC? - gRPC.io
Protocol Buffers - developpers.google.com
RabbitMQ - rabbitmq.com
ActiveMQ - activemq.apache.org
IBM MQ - ibm.com
Circuit Breaker Design Pattern - wikipedia.org
Bulkhead Pattern - dzone.com
Service Mesh - Red Hat
Service Mesh - istio.io
Envoy Proxy - envoyproxy.io
Service Proxy Pod Sidecar - iximiuz.com
Service Discovery - nginx.com
Service Discovery - baeldung.com
RabbitMQ - rabbitmq.com
Consul Architecture - developer.hashicorp.com
Consul Simplified Architecture - developer.hashicorp.com
Consul Consensus - developer.hashicorp.com
What is Gossip Protocol? - educative.io

Data management

Data management

Data Consistency in Microservices Architecture - medium.com
pg_dumpall - postgresql.org

Security

8 Ways to Secure Your Microservices Architecture - okta.com
How to Implement Security for Microservices - medium.com
Securing Microservices - geekflare.com
SAST vs DAST: When to Use Them - circleci.com
Secure at every step - github.com
Static Application Security Testing (SAST) - community.microfocus.com
Client Credentials Flow - auth0.com
Role-Based Access Control (RBAC)? - digitalguardian.com
Transparent Data Encryption in PostgreSQL - arctype.com
Transparent Data Encryption - wiki.postgresql.org
Secrets Management - techmanyu.com
Secrets Manager - cloud.ibm.com
Hashicorp Vault - hashicorp.com