Microservices help you deliver code faster, and make it efficient for teams to split work without conflicting with each other. But what about the security of microservices? In this article, I go over microservices security patterns that will make your application far more secure!

It's very important to take measures to make services secure since a security "hole" in even one service can impact the entire application. There is even more to that, as attested by Adam Gola, TSH's QA Engineer with great personal and professional interest in cybersecurity:

> Implementing microservices requires some considerations in terms of security. Above all, you need to start thinking about architecture security right from the beginning (security by design).
>
> When you are in the development or testing phase, it is already too late to start. You may have huge costs as getting rid of some bugs at this point may sometimes require rewriting the whole module from scratch.
>
> 
> Adam Gola
> QA Engineer

Let's not allow it to happen! Here is a list of 8 **microservices security patterns** that are commonly used for more secure microservices:

Basics of microservices security patterns – start with the code

As befits a software engineer, code is the right place to start if you want to consider how to make your app secure.

At this point, there is not much difference between microservices code and one that you would write in any other architecture.

Your code should be secure and testable.

```ts
import { RequestHandler } from "express";
import { connection } from "./connection";

export interface User {
  email: string;
  password: string;
}

export class UserModel implements User {
  constructor(public email: string, public password: string) {}
}

export const createUser: RequestHandler = async (req, res) => {
  const { email, password } = req.body;

  const user = new UserModel(email, password);

  await connection.save(user);

  res.json({
    result: user,
  });
};
```

request-handler.ts hosted with ❤ by GitHub view raw

Can you spot a security vulnerability in this code?

First of all, there is no validation done, so you don't even know if the user has provided an actual email address, or if they have given us some random characters. You also don't ensure that the password is secure enough. In addition, the code is hard to test due to the imported "connection". Here's a way to fix all this:

```ts
import { RequestHandler } from "express";
import { Connection } from "./connection";
import Joi from "joi";

export interface User {
 email: string;
 password: string;
```

```
}

export class UserModel implements User {
 constructor(public email: string, public password: string) {}


 // Return fields that can be publicly accessible
 get publicData() {
   return {
     email: this.email,
   };
 }
}

const userSchema = Joi.object<User>({
 // Ensure that email is valid and that it is not too long
 email: Joi.string().email().required().max(100),
 // Ensure that password is secure enough
 password: Joi.string().min(5).max(50),
});


// Inject “Connection” here for easier testing
export const makeCreateUser =
 (connection: Connection): RequestHandler =>
 async (req, res) => {
   // Note - this should be done via middleware, but for purposes of this
   // article it’s done here instead
   const validationResult = userSchema.validate(req.body);

   if (validationResult.error) {
     throw validationResult.error;
   }

   // At this point we know that all the values are valid!
   const { email, password } = validationResult.value;

   const user = new UserModel(email, password);

   await connection.save(user);

   res.json({
     result: user.publicData,
   });
 };
```

validation.ts hosted with ❤ by GitHub view raw

This example uses the popular “joi” library for validation. In addition to checking if the email is valid, you ensure that the password has at least 5 and up to 50 characters.

There is also a “publicData” getter in UserModel to ensure that you won’t return fields that should not be public. In this example, it’s a user password. Even though you might encrypt it before saving it to the database, you should avoid returning it in your responses.

Of course, doing that for every object would be quite cumbersome, so you can use GraphQL to declare a public schema that will be returned to the client and keep other fields hidden.

And last but not least, remember, if the user can break something, they eventually will!

Keep track of your dependencies

Did you know that external dependencies can make up even 70% of your code in production?

Often these dependencies have their dependencies, and so on. In order to keep your app secure, you should keep them up to date, because updates often fix security vulnerabilities.

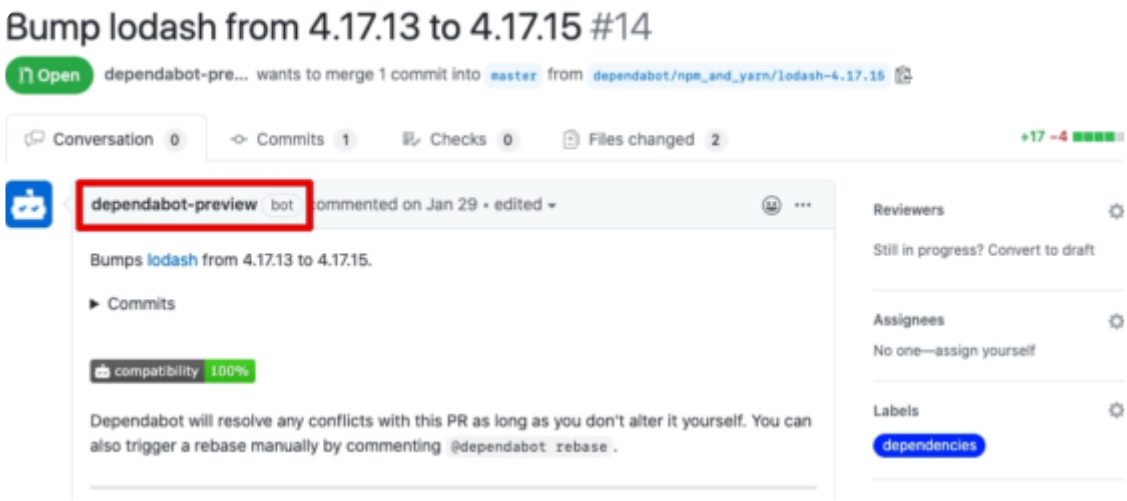
For example, a very popular Node.js library called Lodash introduced a serious security issue in version 4.17.19. The function zipObjectDeep() allowed a malicious user to modify the prototype of an Object if the property identifiers were user-supplied. To be affected by this issue, developers would have to be zipping objects based upon user-provided property arrays.

There are various tools that help you achieve that. The npm audit tool is a good choice if you work with Node.js:

```
# npm audit report

lodash  <4.17.21
Severity: high
Command Injection - https://npmjs.com/advisories/1673
fix available via `npm audit fix --force`
Will install lodash@4.17.21, which is outside the stated dependency range
node_modules/lodash

1 high severity vulnerability

To address all issues, run:
  npm audit fix --force
```

Example npm audit output

You can also use GitHub dependabot to update your outdated dependencies automatically.

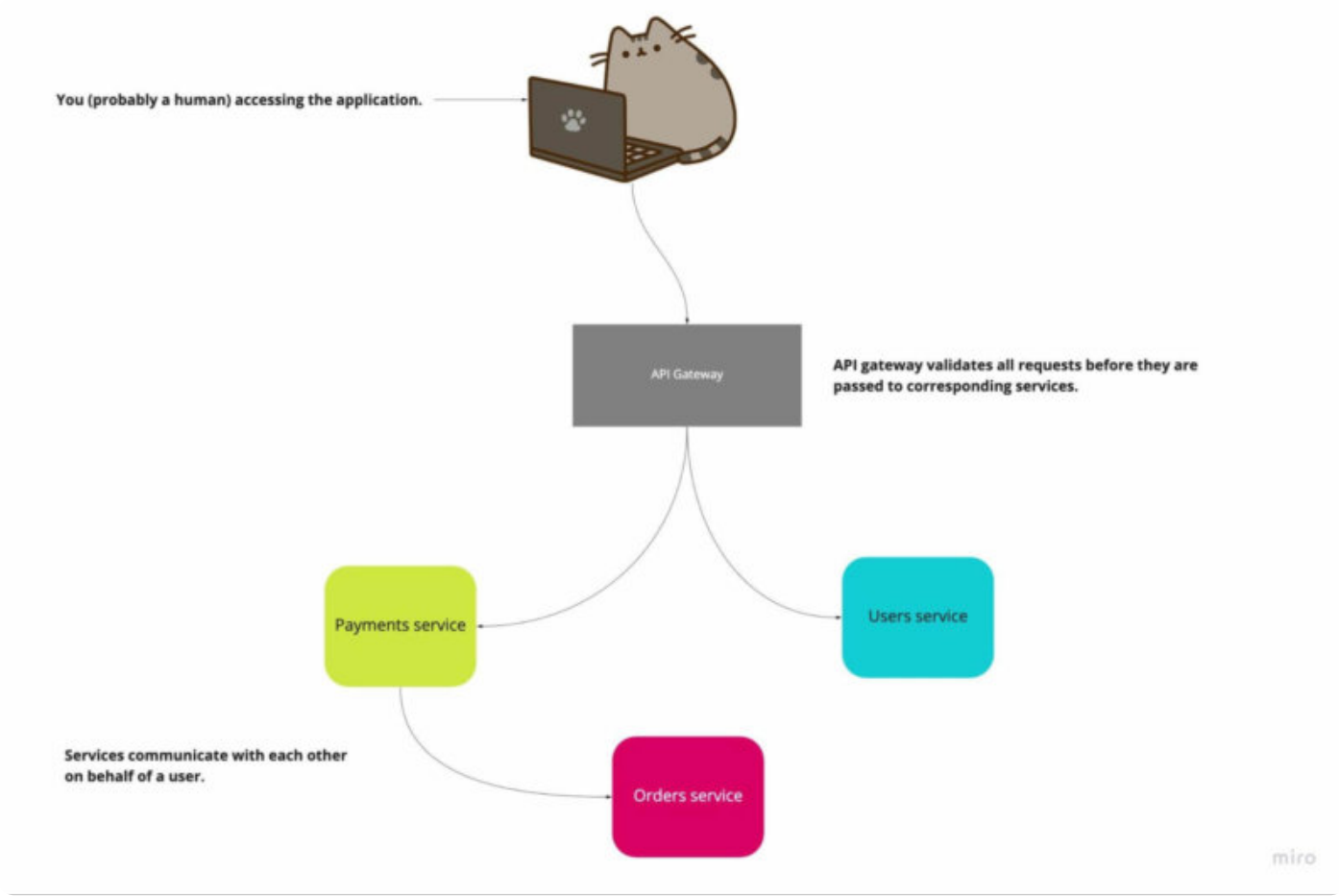


Dependabot can scan your repository and automatically create merge requests with updated dependencies. (service)

If your project is not hosted on Github, you can use other tools such as Snyk. It has integrations with the majority of popular git providers (BitBucket, GitLab etc.).

Use API Gateway

Usually, microservices applications contain many services that are accessible by different clients and systems. It exposes them to many security risks since it's hard to monitor every single service.



Microservices architecture simplified

That's why you can deploy API Gateways that handle, monitor and scan all incoming traffic before it is passed to the target service. It will act as a point of entry for all requests

Secure your communication

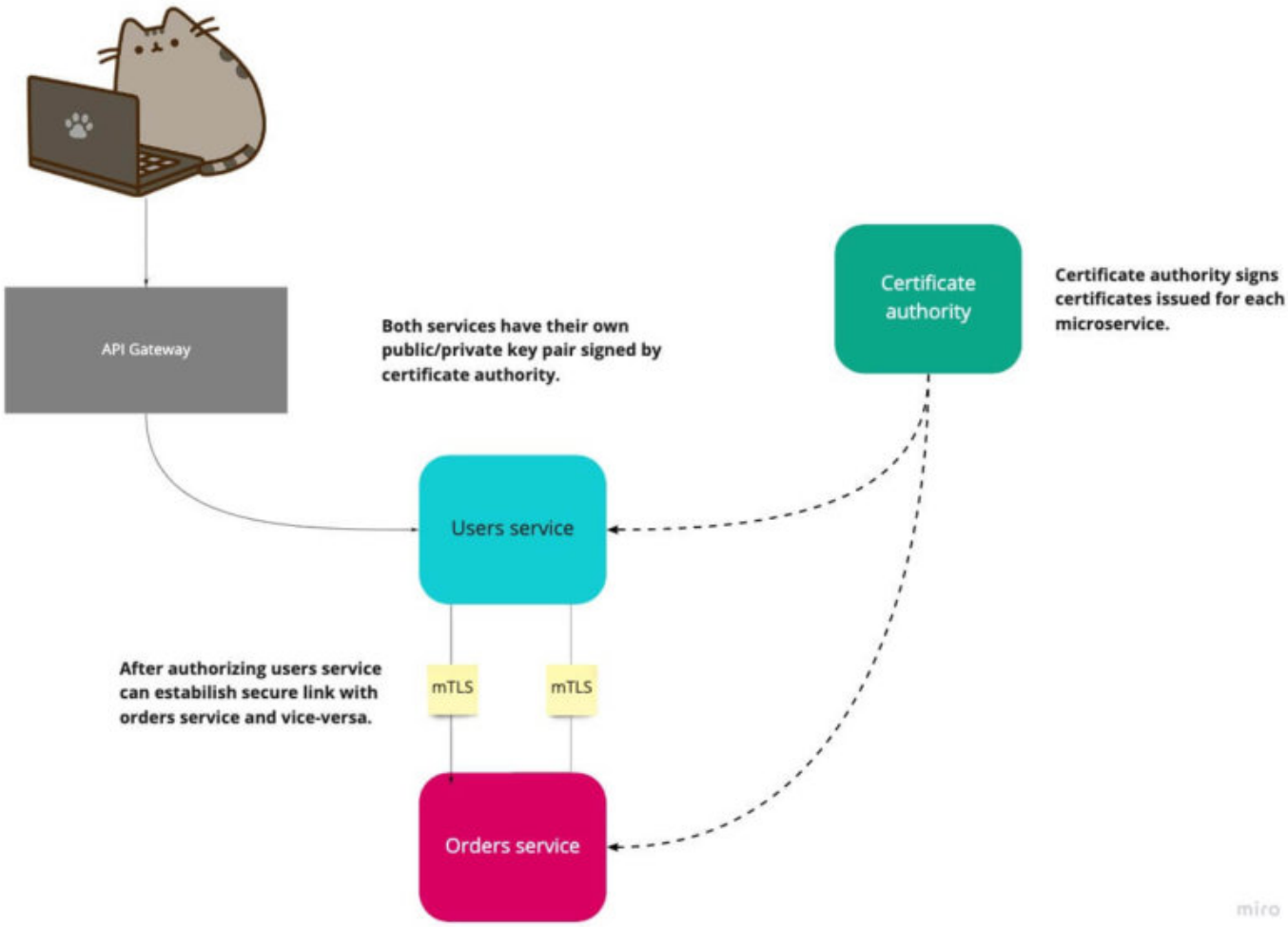
Secure communication in microservices is a very important topic. Various testing methods are used such as dynamic application security testing, when the tester tries to entre the app through the frontend. Still, there are some more basic things you can do to implement security in your communication.

> **By default, you should enforce the use of HTTPS in your entire application. When sending sensitive information, such as client credentials, passwords, keys or secrets, encrypt it as soon as possible,**

and decrypt it as late as possible. Never send this kind of information in plain text.

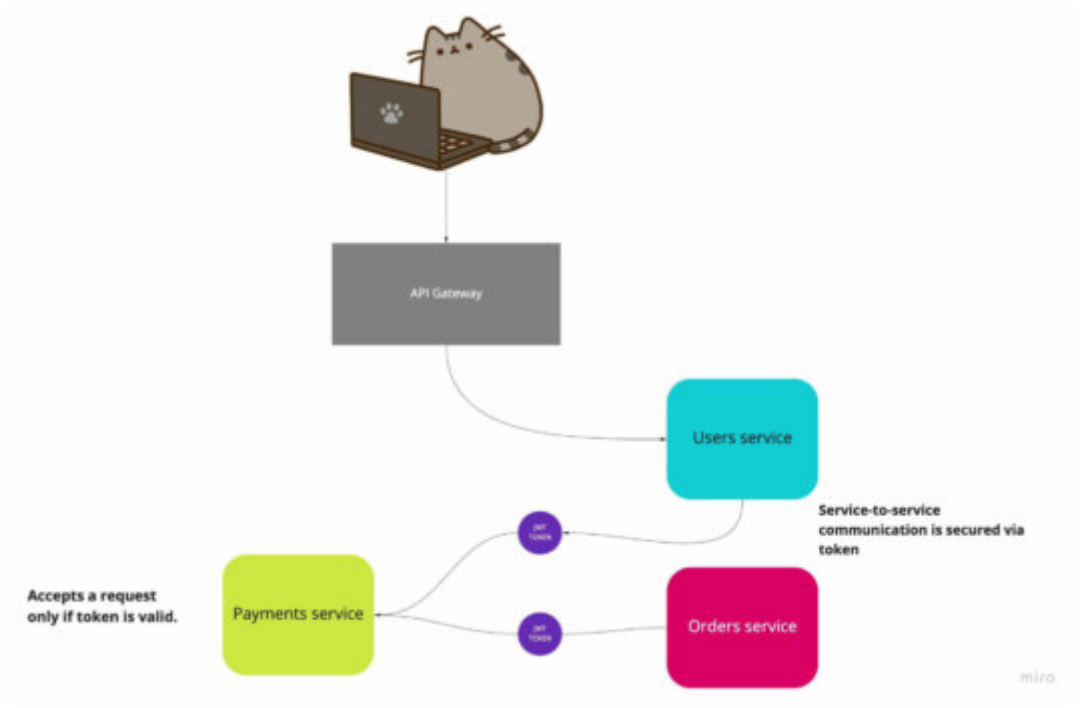
There are several ways of how you can secure communication between services. However, these two are the most common: **Mutual Transport Layer Security** (mTLS) and **Json Web Token** (JWT).

mTLS – each microservice has a public/private key pair issued by a trusted certificate authority. The client then uses the key-pair to perform basic authentication through the mTLS.



mTLS communication

JWT: each microservice has its own unique JWT token. Clients must know this token in order to access it (access tokens).



JWT Microservices security pattern for secure communication

More expert content on microservices: read up

An introduction to Microservices for CTOs: microservices pros and cons for business

Communication in event-driven microservices architecture

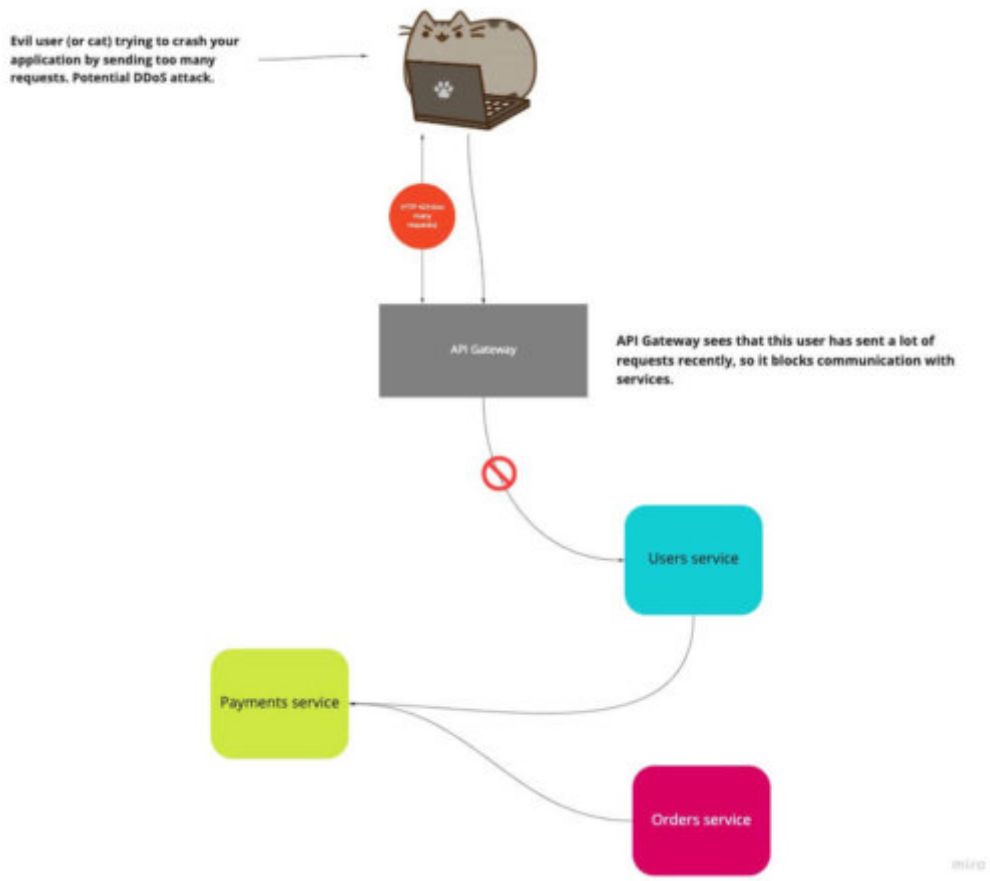
Design patterns in microservices for Devs and CTOs: Direct, API Gateway, Backend for Frontend

Gentle introduction to microservices, federated GraphQL and Apollo Federation

How to automate deployment and SSL certification of your Kubernetes microservices with cert-manager and Garden

Use rate limiting

Limiting traffic prevents one of the most common attacks: denial of service (DoS). In addition, it prevents certain services from using most of the application bandwidth.



Blocking requests in microservices

There are several ways you can apply it, but the most common one is to monitor requests per IP, or time and act accordingly.

To achieve that, you can use packages such as express-rate-limit if you are using Express, or fastify-rate-limit for Fastify.

```ts
import rateLimit from "express-rate-limit";

const limiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 100 // limit each IP to 100 requests per windowMs
});

// apply rate limiter
app.use(limiter);
```

limiter.ts hosted with ❤ by GitHub view raw

On the client-side, you can use exponential-backoff to automatically retry requests that have failed due to reaching the rate limit after a random period of time.

```ts
import { backOff } from "exponential-backoff";

function getUsers() {
  return fetch("/api/users");
}

async function main() {
  try {
    // If request fails (ex. we are blocked by rate limit) it will be
    //retried automatically after a random period of time!
    const response = await backOff(() => getUsers());
    // process response
  } catch (e) {
    // handle error
  }
}

main();
```

backoff.ts hosted with ❤ by GitHub view raw

If you are especially worried about distributed denial-of-service (DDoS) attacks, you can check if your cloud provider provides any additional security against it. For example, in the case of AWS, it is AWS Shield.

Secure your containers

Microservices are typically deployed inside containers. Because of that, ensuring that your containers are secure – both internally and externally – is a good step towards making your whole application more secure. To that end, there are a couple of good practices to follow:

- don't run your services using "sudo" or administrator accounts,
- provide only a minimum of permissions,
- don't store secrets inside containers,
- ensure that your container image is downloaded from a secure source (ex. Docker Hub),
- keep your container image up-to-date with the latest security updates,
- work on access control to limit access to available resources (ex. memory and CPU),
- perform regular security and vulnerability scanning (ex. in Docker docker scan my-image). You can also scan your Dockerfile – docker scan –file Dockerfile.

There are also platforms such as Anchore that provide in-depth inspection and reporting capabilities for your containers that you can integrate with your CI/CD process

Keep your secrets secret

This is a practice that applies not only to microservices but is important enough to mention it here. You should keep your secrets (logins, passwords, API Keys etc.) in a secure place and never commit them with your code.

You can store your secrets in an external store, e.g. AWS Parameter Store, Vault, AWS Secrets Manager or AWS KMS.

Monitor your services

Since it's hard to monitor every single service due to its distributed nature, it's especially important to have a good tool that will make it as painless as possible. Some of them include AWS Cloudwatch, Grafana, Prometheus and DataDog.

You should find a tool that is most appropriate for your application. Some of the best practices for monitoring include:

> Logging in an application layer (we often use winston in our applications) that should be verbose enough to provide you with all the necessary information about the state of your application. With enough logs, you can more easily determine what went wrong, and find potential network security issues before they escalate.
>
> Watch for trends in metrics, such as CPU, Memory or response times. Unusual spikes in these activities often indicate security problems or even a potential attack.

You should dedicate extra time to optimizing the monitoring process. Experience usually proves that it is time well spent!

To learn more about app optimization and monitoring, check out this Grafana and Prometheus tutorial.

Keep tabs on further developments in microservices security patterns!

Microservices bring a lot of benefits for your organization – faster deployments, easier scalability and more flexibility. It's important though not to forget about security as we speed up and scale our applications. There are more interesting patterns and topics to follow such as security challenges, defense in depth, scan dependencies or access token.

The microservices security patterns mentioned above should give you an overall idea of how to make your microservices more secure.

To be even better prepared, follow the general security trends as well. A good way to do this is check out this OWASP TOP 10 article written by Adam Gola that goes over the latest security trends and vulnerabilities.