DATA SECURITY

Alessandro Bocci

name.surname@unipi.it

Advanced Software Engineering (Lab)

13/11/2025

What will you do?

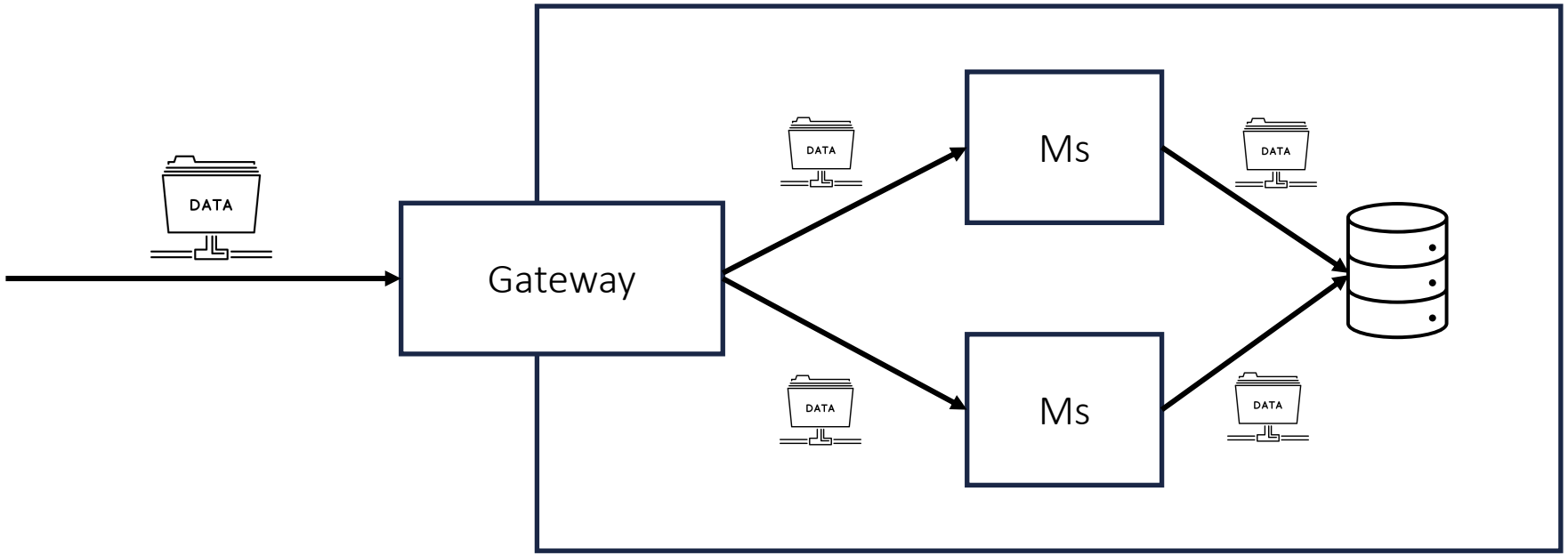
Employ mechanisms to protect a microservice architecture data.

In particular:

- Data in transit
- Data in use
- Data at rest

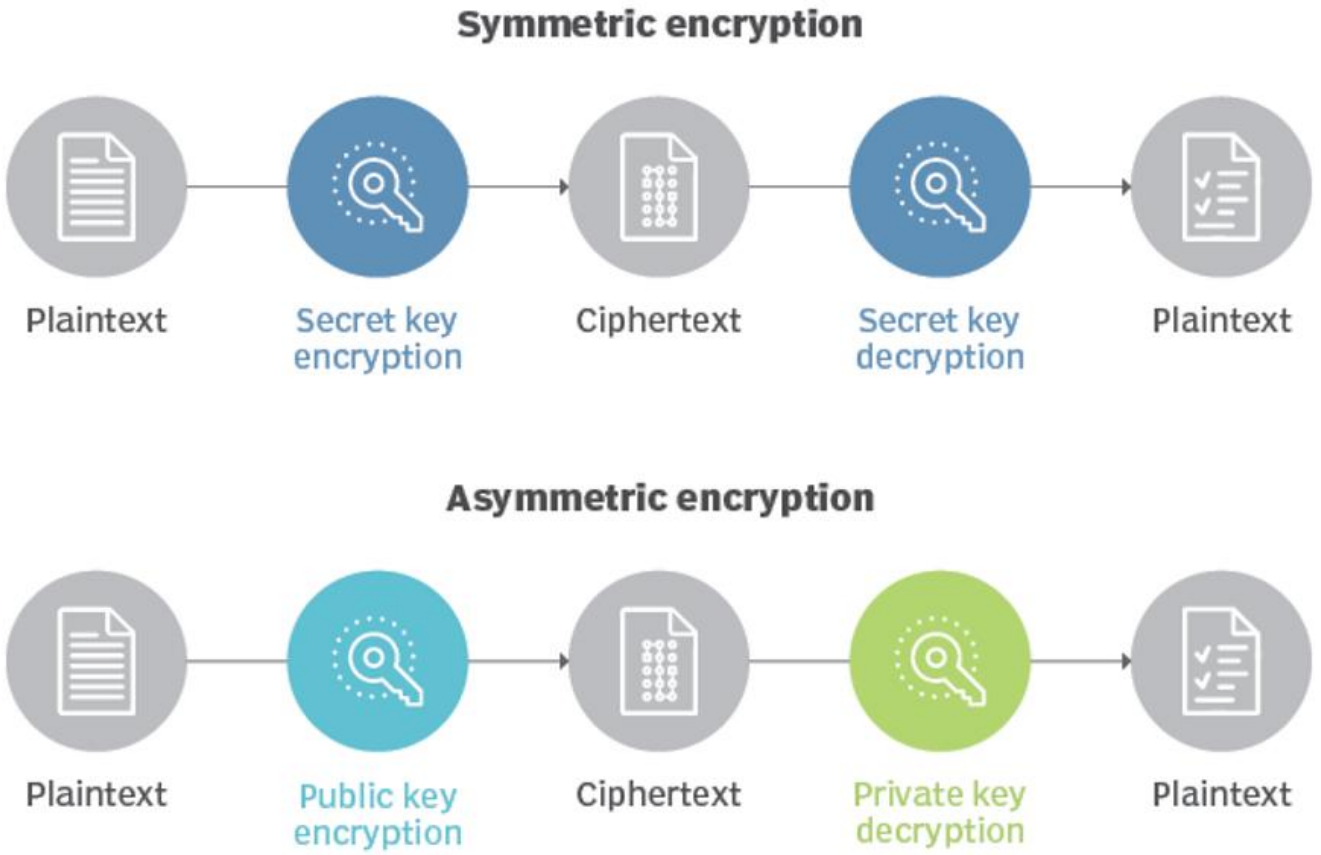
Data in transit

Data transmitted from a point to another.



Data in transit

Basic defense: encrypt the communication channel



Data in transit

With services is not practical and safe to use symmetric encryption.

To implement asymmetric encryption are used certificates to avoid impersonification.

HTTP + TLS Certificate = HTTPS

(TSL = SSL updated version)

One-way TLS vs Mutual TLS

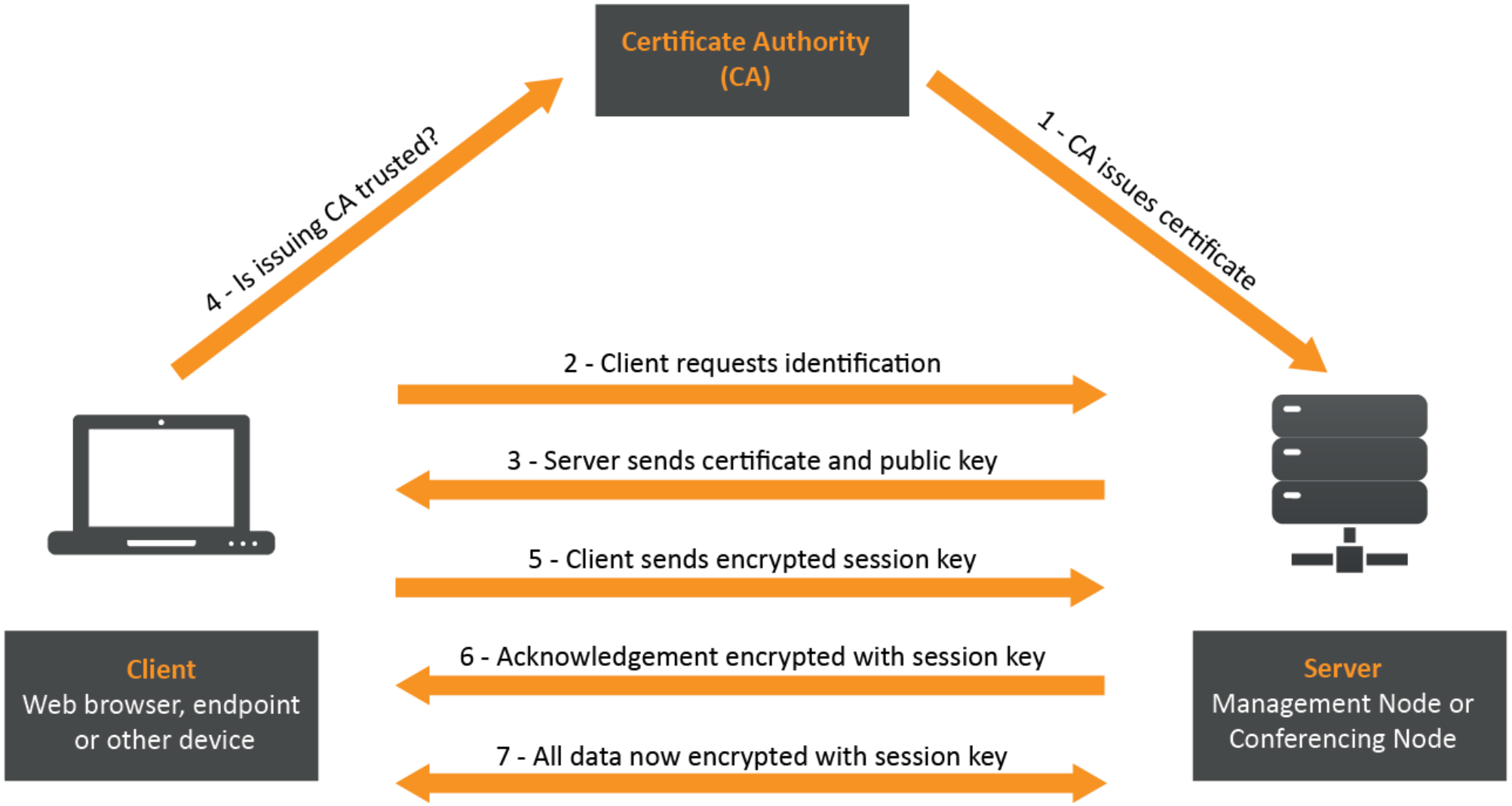
One-way TLS:

- Client verifies the server's certificate (CA-issued).
- Encrypts the channel; assures server identity only.
- Common for browsers & most public APIs.
- Client identity handled by app layer (tokens/keys/passwords).

Mutual TLS:

- Both client and server present & verify certificates.
- Adds strong client authentication at the transport layer.
- Great for service-to-service, zero-trust, and regulated environments.

One way TLS

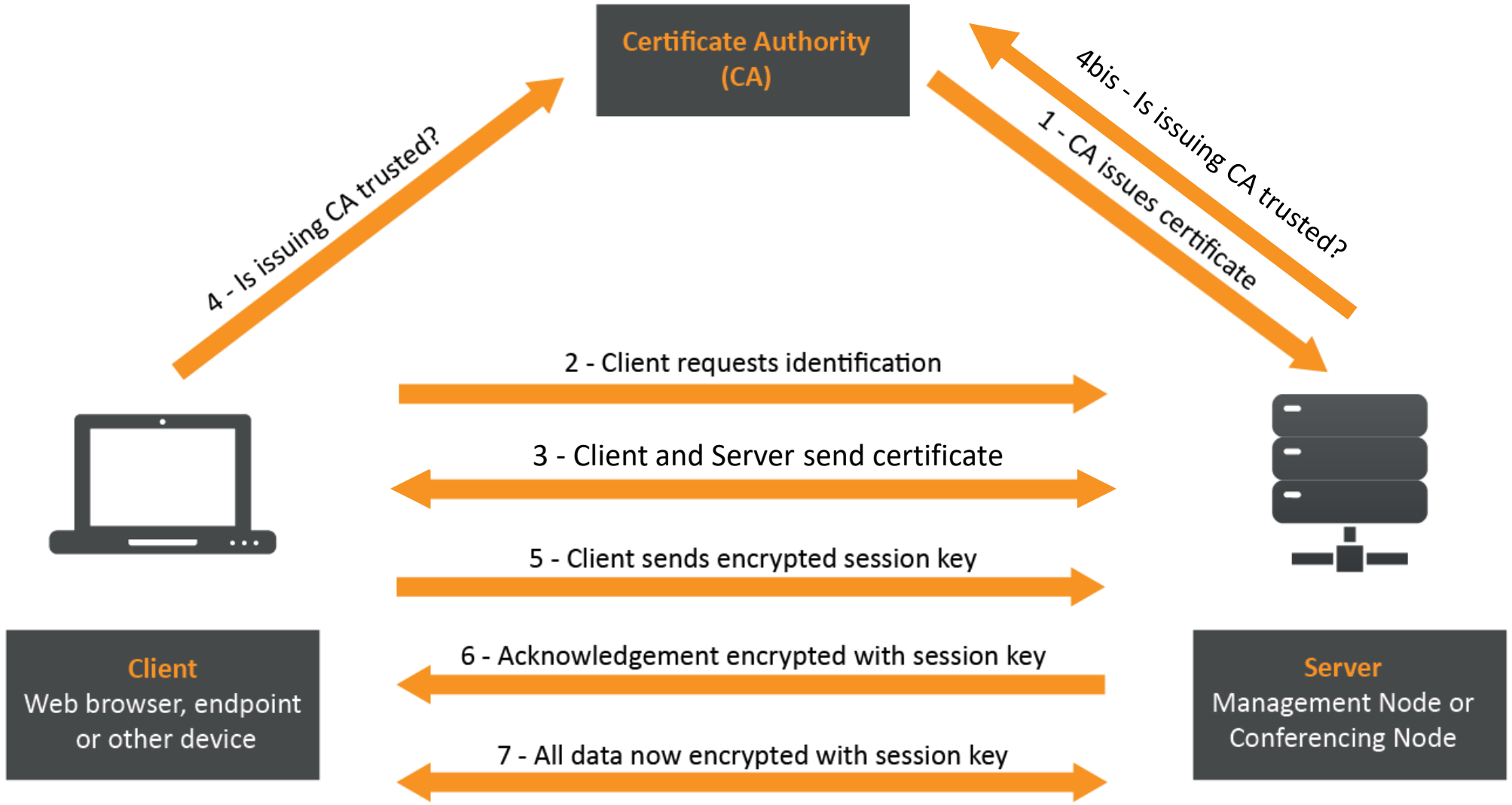

Certificate Authority
(CA)
4 - Is issuing CA trusted?
1 - CA issues certificate
2 - Client requests identification
3 - Server sends certificate and public key
5 - Client sends encrypted session key
6 - Acknowledgement encrypted with session key
7 - All data now encrypted with session key
Client
Web browser, endpoint or other device
Server
Management Node or Conferencing Node

Mutal TLS

Has parallel (3) and (4) steps:

- The client sends its certificate as well.
- The server contacts the CA to verify its validity.

Mutual TLS


Certificate Authority (CA)
4 - Is issuing CA trusted?
4bis - Is issuing CA trusted?
1 - CA issues certificate
2 - Client requests identification
3 - Client and Server send certificate
5 - Client sends encrypted session key
6 - Acknowledgement encrypted with session key
7 - All data now encrypted with session key
Client
Web browser, endpoint or other device
Server
Management Node or Conferencing Node

TLS Certificates

A TLS certificate contains:

- Public key (for encryption/decryption).
- Subject (information about the entity the certificate is issued to).
- Issuer (the Certificate Authority that issued the certificate).
- Validity period (start and expiry dates).
- Digital signature (from the CA, to confirm the certificate's authenticity).

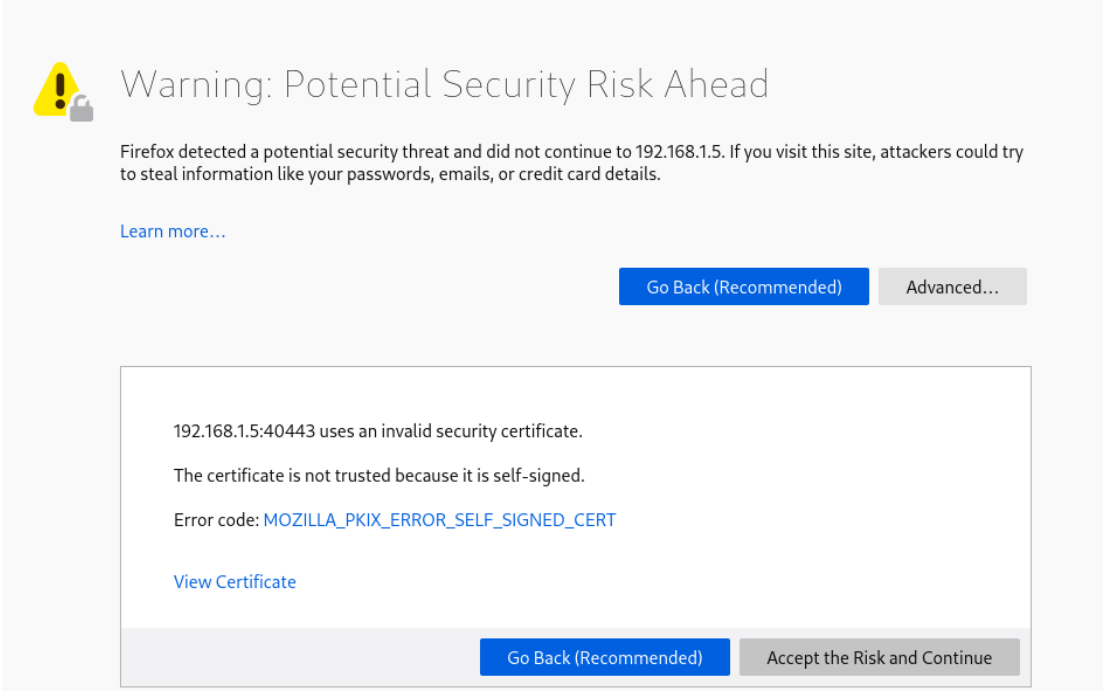
TLS Certificates

For a real certificate (generally) you have to pay.

For development we can you self-signed certificates = No CA involved.

But is less secure, obviously.

Browsers warning you about them.



Self Signed Certificates

Use OpenSSL to create a self-signed certificate.

```
openssl req \
  -x509 \ #Generates a x509 certificate
  -newkey rsa:4096 \ #Generates a private key using RSA with 4096bit modulus
  -nodes \ #Avoids encrypting the key (and asking a passphrase)
  -out    cert.pem \ #Name of the output file of the certificate in pem format
  -keyout key.pem \ #Name output file of the key in pem format
  -days 365 \ #Validity period (365 days in this example)
  -subj "/CN=ms" \ #Sets the Subject to Common Name = service
  -addext "subjectAltName=DNS:ms,DNS:localhost,IP:127.0.0.1"
  #Adds Subject Alternative Names for hostname/IP verification
```

This command generates a self-signed certificate and its private key for a service named **ms**, which expires in one year.

Activate HTTPS – Flask

There are several ways to pass the certificate to a Flask Service, the most common one is to execute it as

```
flask run --cert=cert.pem --key=key.pem …
```

Or use them in the code as

```
if __name__ == "__main__":
    app.run(ssl_context=('cert.pem', 'key.pem'))
```

Activate HTTPS – Docker

Docker compose can manage secrets using specific entries in the docker compose file.

```
services:
  myapp:
    image: myapp:latest
    secrets:
      - my_secret
secrets:
  my_secret:
    file: ./my_secret.txt
```

In the `myapp` container, `/run/secrets/my_secret` is a file set to the contents of the file `./my_secret.txt`.

Activate HTTPS – Docker

You need to:

- Create secrets for all certificates and keys (in the docker compose file).
- Pass secrets needed by each microservice (in the docker compose file).
- Allow Flask access certificates and keys (in one of the ways explained before).

Activate HTTPS – Python requests

We need to change the requests to HTTPS

Example of HTTPS GET:


```
requests.get('https://other:5000/endpoint', verify='run/secrets/other_cert')
```

Our Flask service needs 2 certificates:

- Its own (and its private key).
- The certificate of the service to invoke (**other**).

The 2 certificates are exchanged and verified during the GET request.

Testing with HTTPS

What is changing?

The protocol used and the certificate exchange.

With Postman, we have to

- Add `https://` in the URL.
- Disable the certificate verification (Settings->General->SSL certificate verification: OFF) **[Development mode only!!!]**

With a browser, you will receive a warning about the self-signed certificate; you can skip the warning.

Testing in isolation with HTTPS

Also for testing a microservice in isolation you will need to use HTTPS.

If you need to run the container stand-alone with **`docker build`** + **`run`** you can change the testing Dockerfile:

```
FROM python:3.12-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install -r requirements.txt

COPY app*.py .
#Here I copy certificate and key in the same path as docker compose
COPY cert.pem /run/secrets/certificate
COPY key.pem /run/secrets/key

EXPOSE 5000

#The command use the same path for certificate and key
CMD ["flask", "--app", "app_test", "run", "--host=0.0.0.0", "--port=5000",
"--cert=/run/secrets/certificate", "--key=/run/secrets/key"]
```

Testing in isolation with HTTPS

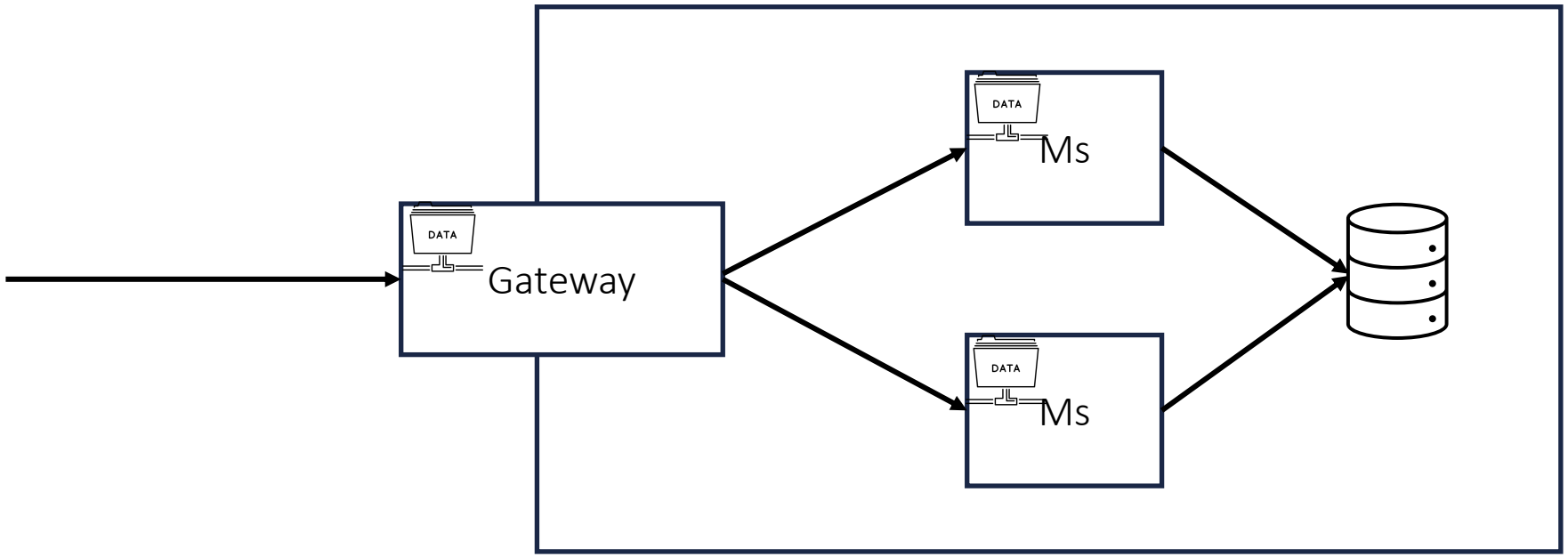
In this example, the `COPY` puts the certificate and the key in the same path as using docker compose secrets (`/run/secrets/`).

Doing like this allow us to use the same code for testing mode without changing paths of secrets.

Data in use

If the system is compromised, memory and CPU data can be read or tampered.

Before processing data, we have to be careful with injection attacks.



Data in use

Advanced defense:

- Utilizing hardware-based protections (like TEEs or enclaves).
- Encrypting sensitive data in memory or while being processed.
- Implementing advanced cryptographic techniques (like homomorphic encryption or SMPC).

We will focus on basic defence before processing data.

Data in use

Basic defense: input sanitization

Input sanitization

Protect the system by malicious input and injection attacks.

In general: remove special characters by input strings.

You can do it “by hand” or using specialized libraries.

Example:

Run `microase` with docker compose and check the logs after sending a GET to

```
http://localhost:5000/str/reduce?op=upper&lst=print("ciao",flush=True)
```

Sanitize input

There are tons of way to sanitize input.

The basic strategy is to avoid unnecessary characters in the input and avoid to build output strings with unchecked input.

Easiest way: type checking and regular expression

```python
import re

def sanitize_username(input_str):
    # Only allows alphanumeric characters, underscores, and hyphens
    sanitized_str = re.sub(r'[^a-zA-Z0-9_-]', '', input_str)
    return sanitized_str

# Example usage
user_input = "user@name!#$"
safe_input = sanitize_username(user_input)
print("Sanitized input:", safe_input) #Output: username
```

Sanitize input

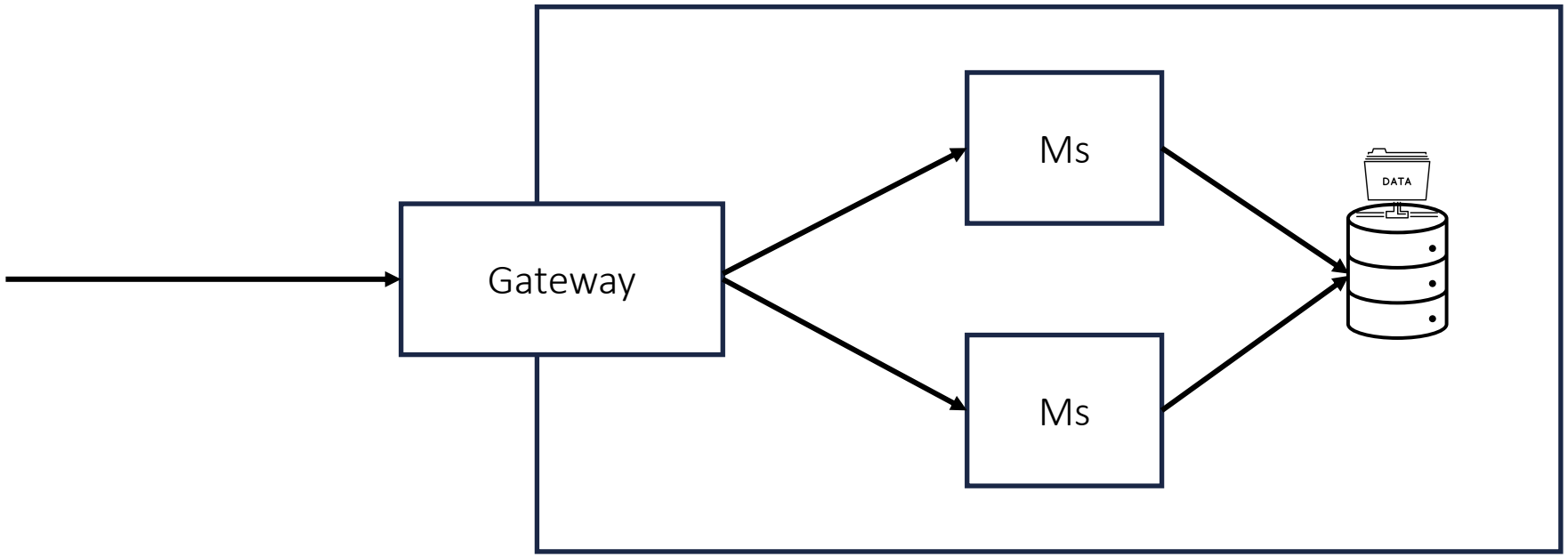
Check all the routes of your microservices and be careful when you accept input from query strings or payloads.

Sanitize inputs by checking the type and excluding unnecessary characters.

Avoid using or outputting unchecked data to other services or databases.

Data at rest

If the system is compromised, databases data can be read and tampered with.



Data at rest

Basic defense: access control and encryption (again)

- Follow the Least Privilege Principle to give access to only who need it.
- You can encrypt at different level (db, table, field, etc.)

  The bigger the encrypted part the slower are performance.

  E.g. Encrypting the database means to en/decrypt data at every write/read

Data at rest

We have to choose carefully the data at rest to encrypt:

- Most sensible data?
- Less used data?

It depends on the situation.

Security vs Performance is always a trade-off without a win-win solution.

Data at rest

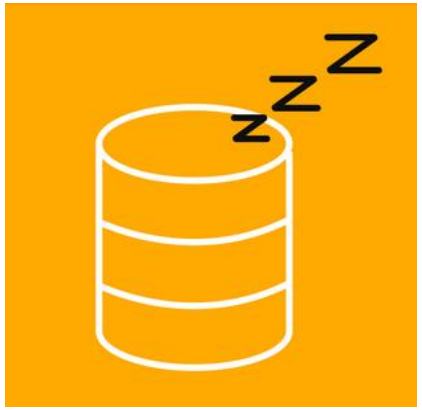
Database technologies (Redis, Mongo, MySQL etc.) provide support to access control and encryption.

- Access with credentials or API keys.
- HTTPS for communication.
- Native encryption of data.

DB: Access Control and Encryption

This part really depends on the DB technology used.

You have to check the documentation to how enable credentials and encryption (it is also needed the certificate and the secret key).



Example: MongoDB

You can see how secrets are used to pass credentials and TSL configuration.

The command is changed to enable authorization and HTTPS.

```
services:
…
  db:
    image: mongo:latest
    ports:
      - '27017:27017'
    volumes:
      - dbdata:/data/db
      - configdata:/data/configdb
    environment:
      MONGO_INITDB_ROOT_USERNAME: root
      MONGO_INITDB_DATABASE: admin
      MONGO_INITDB_ROOT_PASSWORD_FILE: /run/secrets/mongodb_password
    command: mongod --auth --quiet --logpath /dev/null
      --tlsAllowConnectionsWithoutCertificates
      --tlsMode preferTLS
      --tlsCertificateKeyFile /run/secrets/mongodb_certkey
      --tlsCAFile /run/secrets/mongodb_certkey
    secrets:
      - mongodb_password
      - mongodb_certkey

volumes:
  dbdata:
  configdata:

secrets:
  mongodb_certkey:
    file: mongo.pem
  mongodb_password:
    file: mongo_password.txt
```

Example: MongoDB

In the example, you can see:

- Authorization and credentials (--auth, env variables and secrets)
- TLS (MongoDB wants a single file with certificate and private key)

Also, here (like with Postman) we disable certificate verification for teaching purposes, as MongoDB does not accept self-signed certificates.

Example: DB Manager

```
from pymongo import MongoClient

username = "root"
host = "db"
port = "27017"
db = "admin"
database = "my_database"

with open('/run/secrets/db_password', 'r') as file:
    password = file.read().strip()


uri =
f"mongodb://{username}:{password}@{host}:{port}/{database}?authSource={db}&tls=true&
tlsAllowInvalidCertificates=true"

client = MongoClient(uri)
```

Here the certificate and the password are taken from secrets and the URI contains TLS configuration.

Obviously this conifiguration must match the db one.

DB: Encrypt data

First, you have to decide which are the important data.

Remember the trade-off between performance and security.

Example:

- Users' privacy info, Users' credentials, Users' payment info…

(We will see Users' credentials management next Lab)

DB: Encrypt data

How to encrypt data depends on what and where you encrypt:

- Fields: encrypt/decrypt data in the DB manager before sending them to the DB.
- Tables and database: exploit the DB features where available (check their docs).

MongoDB does not support collection encryption natively.

You have to encrypt/decrypt everything before the DB.

Example: Encrypt/Decrypt data

Python example using Fernet as library for symmetric encryption.

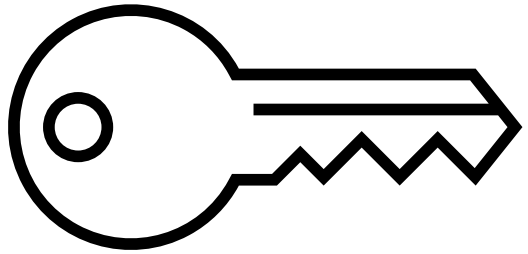
Ideally, we should used a KMS to managed keys.

```python
from flask import Flask
from cryptography.fernet import Fernet

app = Flask(__name__)

# Generate a key for encryption and decryption
# This example creates a key every run
# We should read the key from secrets as before
key = Fernet.generate_key()
cipher_suite = Fernet(key)

@app.route('/save_data', methods=['POST'])
def save_data():
    data = request.json.get('data')
    # Encrypt the data
    encrypted_data = cipher_suite.encrypt(data.encode())
    # Store encrypted data
    db[data_id] = encrypted_data
    # Answer to the client on success
@app.route('/get_data/<data_id>', methods=['GET'])
def get_data(data_id):
    # Fetch encrypted data from the "database"
    encrypted_data = db.get(data_id)
    # Decrypt the data
    decrypted_data = cipher_suite.decrypt(encrypted_data).decode()

    return jsonify({"data": decrypted_data}), 200
```

Key Management Systems (KMS)

Systems to manage the cryptographic keys.

- Keys Exchange.
- Keys Storage.
- Keys Usage.
- Keys Rotation.



Encryption is useless if you do not care about your keys.

Not required for the project.

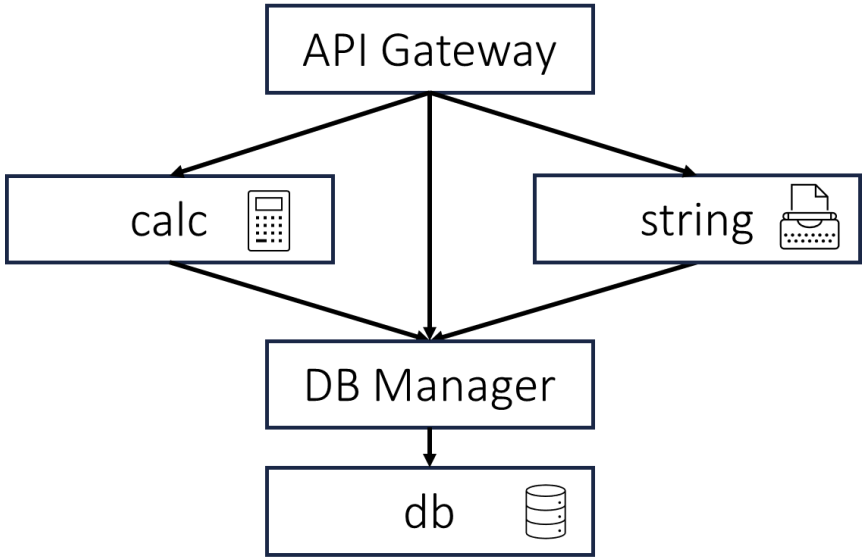
In real case scenarios…

- You will have real certificates, released by Certification Authorities.
- Avoiding to validating them is not secure.
- We saw some basic mechansisms, generally you will use something more sophisticate.
- The management of private keys is vital for security involving encryption.

Today's Lab

Work on your project code or use **`microase`** (your own or from Moodle).

- Activate HTTPS for all microservices with self-signed certificate.
- Change the HTTP requests to HTTPS.
- Sanitize data in input.
- Activate access control and encryption of the database.
- Encrypt the logs sent to the db.



Recap

1. Generate TLS certificates and keys for each microservice.
2. Modify the docker compose with secrets for them.
3. Modify microservices' Dockerfiles or code to read them.
4. Sanitize inputs.
5. Activate TLS and credentials in the DB(s).
6. Modify microservices' code to interact with DB(s).
7. Encrypt data sent to the db from the DB manager.

Lab take away

- ❑ Learn how to secure data in microservice architectures.
- ❑ Basic strategies for data in transite, in use and at rest.
- ❑ Encryption without a secure management of keys is weak.
- ❑ Performance vs Security balance depends on your case.

Project take away

- ❑ The project microservices must use HTTPS when communicating.
- ❑ Encrypt important data at rest of your project.
- ❑ Sanitise inputs before using it.