NGINX tutorial for securing a REST API

First of all, why would we secure a REST API?

Securing a REST API is crucial for safeguarding sensitive information and preventing unauthorized access. APIs often handle personal data, authentication tokens, and user details, making them attractive targets for malicious actors. Proper authentication and authorization mechanisms ensure that only authorized users or applications can access and manipulate resources, enhancing overall data protection.

Additionally, securing the API helps maintain data integrity during transmission through encryption, preventing tampering or manipulation. It also mitigates the risk of denial-of-service attacks by implementing measures such as rate limiting. Compliance with industry regulations, monitoring, and logging of API activities further contribute to maintaining user trust and preventing potential security incidents. In summary, a secure REST API not only protects against unauthorized access but also helps uphold brand reputation, comply with regulations, and ensure the overall reliability of the API and the organization it serves.

Now we will present an example of how this can be achieved. When looking for an authorization method multiple options come up: API keys, OAuth, Basic Auth, Bearer Tokens, Json Web Tokens, etc. In this tutorial we are going to use Json Web Tokens with the Bearer Token Authorization.

In order to let users acces our API we need a way for creating a set of credential and a way to verify them. For This we provided two public endpoints: /auth/login, /auth/register.

Lets first take a look at the register endpoint. As we expect the user must pass an email and a password. The important aspect for this process of creating a user in our database is storing the password in a secure way. This is achieved by hashing the password like in the following code spippets:

```
const sha512 = function (password, salt) {
  const hash = crypto.createHmac('sha512', salt)
  /** Hashing algorithm sha512 */
  hash.update(password)
  const value = hash.digest('hex')
  return {
    salt: salt,
    passwordHash: value
  }
}

export function getSaltHashPassword (password: string) {
  return sha512(password, passwordSalt)
}
const passObject = getSaltHashPassword(password)
const userObject = User.build({
  name,
  email,
  password: passObject.passwordHash
})
```

Now lets take a look at the login endpoint. As expected the user need to pass in the request body their email and password. Because we store the hashed value of the password in our database we need to compare it with the result of hashing the password we received in the request body:

```
export const verifyPassword = (password, hashedPassword)
=> {
  return sha512(password, passwordSalt).passwordHash ===
hashedPassword
}

const userObject = await User.findOne({ where: { email } })
if (!userObject) {
  throw new WrongLoginInfoException()
}

const isVerified = verifyPassword(password,
userObject.password)
```

In the case that the email and password that were passed in the request are correct we need to send in the response of the request an JSON Web Token that will be used as the Authorization method for making requests to other endpoints. Here is an example of how this can be done:

```
export const passwordSalt = process.env?.JWT_SALT || “our
big secret”

export function getJWTToken (payload) {
  return jwt.sign(payload, passwordSalt, { expiresIn:
'3y' })
}

const isVerified = verifyPassword(password,
userObject.password)
if (isVerified) {
  return { token: getJWTToken({ id: userObject.id }) }
} else {
  throw new WrongLoginInfoException()
}
```

Now this token will be sent in the other requests made to private endpoints.
But how do we add the authorization step before accessing the endpoint?
For this NGINX comes in handy. It allows us to add an intermediary auth request before the actual request to the secured endpoint. Lets take a look at how this works:
In our example we have a private path called /ticket/ which has to be private. To acieve this we created the /check-token endpoint in the auth service which just checks that the token passed is valid.

```
authRoute.get('/check-token', authHandler, function
(req, res) {
  res.setHeader('User-Id', req.user.id)
  res.sendStatus(200)
})
export const authHandler = (req: Request, res:
Response, next: NextFunction) => {
  const authHeader = req.headers?.authorization
  const token = authHeader && authHeader.split('
')[1]

  jwt.verify(token, passwordSalt, (err: any, user:
any) => {
    if (err) {
      return res.sendStatus(401)
    }
    req.user = user
    next()
  })
}
```

If the token is valid it also returns the userId. Now we can configure the /check-token in NGINX as follows:

```
location /check-token {
  add_header 'Access-Control-Allow-Headers' 'Authorization' always;
  internal;
  proxy_pass          http://service-auth:3000/check-token;
  proxy_redirect      off;
  proxy_set_header    Host $host;
  proxy_set_header    X-Real-IP $remote_addr;
  proxy_set_header    X-Forwarded-For $proxy_add_x_forwarded_for;
  proxy_set_header    X-Forwarded-Host $server_name;
  proxy_set_header    Authorization $http_authorization;
  proxy_set_header    Content-Length "";
  proxy_pass_request_body off;
}
```

What this does is that it routes the requests made to /check-token to go to the auth service for which we showed the implementation. Now we can ad a auth request for the /ticket requests like this:

```
location /ticket {
  add_header 'Access-Control-Allow-Headers' 'Authorization' always;
  auth_request /check-token;
  auth_request_set $user_id $sent_http_user_id;

  rewrite ^/ticket/(.*) /$1 break;

  proxy_pass         http://service-ticket:3000;
  proxy_redirect     off;
  proxy_set_header   Host $host;
  proxy_set_header   X-Real-IP $remote_addr;
  proxy_set_header   X-Forwarded-For $proxy_add_x_forwarded_for;
  proxy_set_header   X-Forwarded-Host $server_name;
  proxy_set_header   User-Id $user_id;
}
```

What this does is that when a request is made to /tickets NGINX makes a request to the /check-token endpoint from the auth service. If it's successful it makes the request to the /tickets endpoint from the ticket service. Additionally it passes the user id in the header which can be used inside the ticket endpoint logic. In some cases CORS needs to be configured, and spoiler alert: it can be very painful. In my specific situation I had to add custion handling for the OPTIONS requests like this:

```
location /ticket {
   if ($request_method = OPTIONS) {
        add_header 'Access-Control-Allow-Origin' * always;
        add_header 'Access-Control-Allow-Methods' 'GET, POST, OPTIONS'
always;
        add_header 'Access-Control-Allow-Headers' 'Authorization, Content-
Type' always;
        add_header 'Access-Control-Max-Age' 86400 always;
        add_header 'Content-Length' 0 always;
        return 200;
    }
      …
}
```