# Protegendo a web API

Orientadora: Kelly Joany de Oliveira Santos

Reprograma: Segurança - semana 15

# Para que serve a autenticação?

- Controlar acessos
- Somente usuários autorizados/autenticados.

# O que é um Hash?

Os algoritmos de soma hash são algoritmos que retornam um resultado de **comprimento fixo** independente do tamanho da entrada, e esse resultado é **quase** exclusivo para os dados de entrada. Os algoritmos mais conhecidos são: MD5, SHA-1 e SHA-2.

# Exemplo de Hash

| Mensagem | Hash |
| --- | --- |
| Oi | lxFc |
| Olá, como vai? | R+bU |
| Bem-vindo | IB84 |

# Criptografia Simétrica

Os algoritmos de criptografia simétrica utilizam apenas uma chave para criptografar um dado qualquer, que pode ser uma mensagem, etc. Os algoritmos mais conhecidos são: DES, TripleDES, AES, RC4 e RC5.

A principal vantagem da criptografia simétrica é que são muito rápidos, o que se traduz em baixa latência e pouco uso de CPU. Já a principal desvantagem é que por utilizar a mesma chave para criptografar quanto para descriptografar, a chave precisa ser compartilhada com o receptor.

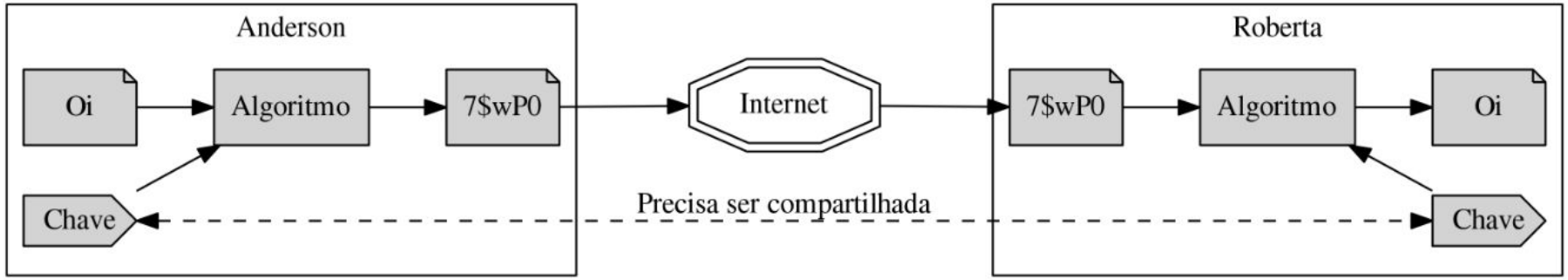



*Figura 1. Envio de Dados com Criptografia Simétrica*

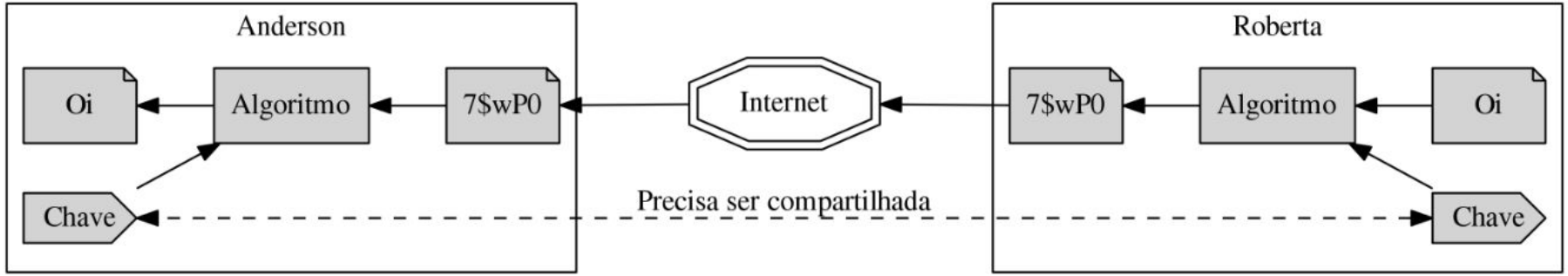



*Figura 2. Recebimento de Dados com Criptografia Simétrica*

# Criptografia Assimétrica

Os algoritmos de criptografia assimétrica utilizam duas chaves complementares para criptografar e descriptografar. Uma das chaves é guardada em segredo e não é revelada ninguém e outra pode ser publicada a qualquer um livremente. Os algoritmos mais conhecidos são: RSA e ECDSA.

Um grande diferencial dessa classe de algoritmos é que um dado criptografado com uma chave pode apenas ser descriptografado com outra e vice-versa. Essa característica permite que estranhos mantenham uma comunicação segura mesmo que o meio de comunicação não o seja, e além disso, não há a necessidade de um meio seguro para que a troca de chave ocorra.

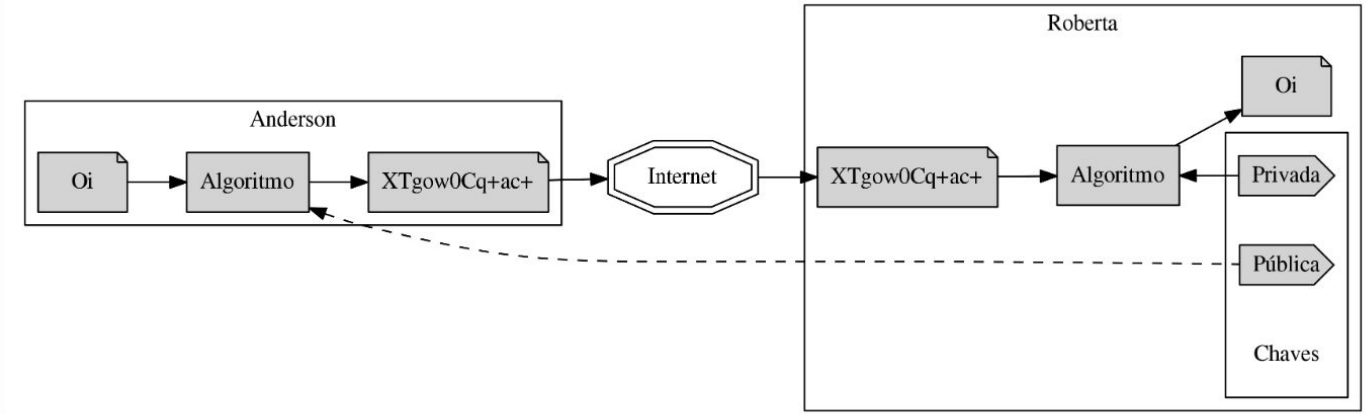



*Figura 3. Envio de Dados com Criptografia Assimétrica*

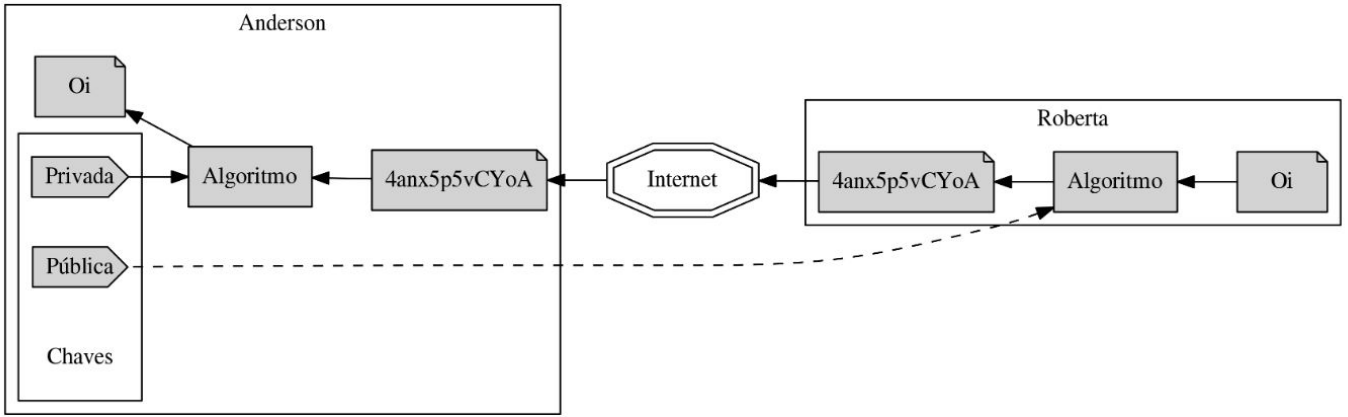



*Figura 4. Recebimento de Dados com Criptografia Assimétrica*

# Desvantagens da criptografia assimétrica

Algoritmos de criptografia assimétrica são muito custosos em termos de CPU, por esse motivo as comunicações, normalmente, os utilizam como meio de troca de chave simétrica. Diminuindo, assim o tempo e recursos da CPU.

# Assinaturas

Há também outro uso muito comum para a criptografia assimétrica, além de ser utilizada para garantir privacidade, também é utilizada em assinaturas para garantir identidade.

Quando queremos apenas confirmar identidade o dado não é privado, pois a chave pública está disponível a qualquer um, o que permite que os mesmos acessem os dados. Assim, uma maneira eficiente de alcançar o mesmo objetivo, com quase a mesma eficiência, é gerar uma soma Hash (Checksum) do dado e criptografar esse resultado. Então a confirmação de identidade passaria a ser da seguinte maneira: gerar uma soma Hash do dado recebido, descriptografar a assinatura recebida e por fim comparar se os resultados são iguais.

# Assinaturas

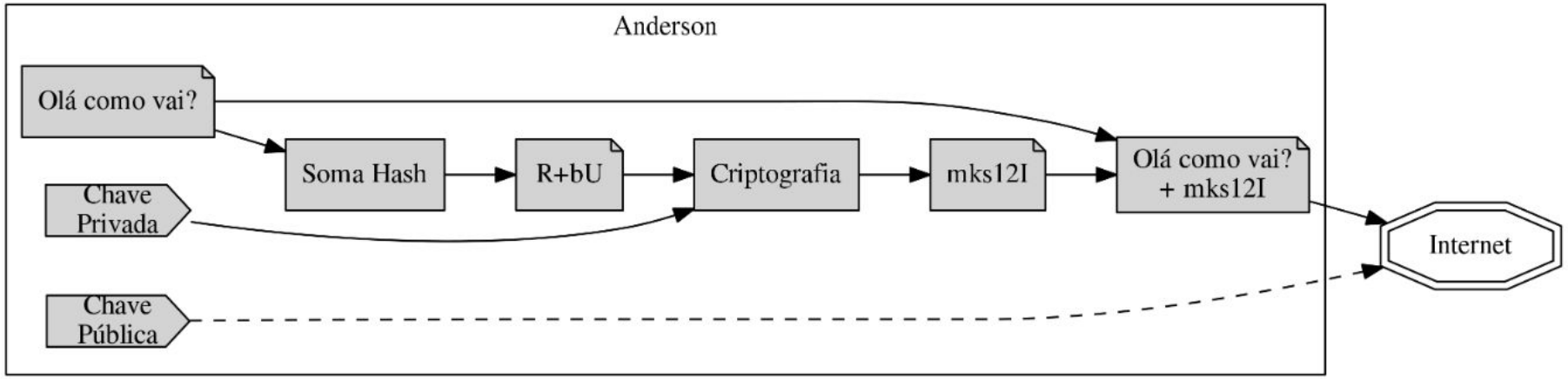



*Figura 5. Assinatura de Dados com Criptografia Assimétrica*

# JWT - Json web token

O padrão JWT permite as informações sejam assinadas tanto com criptografia simétrica (com o algoritmo HMAC) quanto com criptografia assimétrica (com os algoritmos RSA e ECDSA).

Os JWTs são muito utilizados no processo de autenticação permitindo que o processo de autorização de acesso a recursos seja mais rápido e escalável. Mais rápido porque por ser independente retira da equação o tempo de latência de acesso ao banco de dados ou outro mecanismo de cache. E mais escalável pois permite que serviços totalmente independentes compartilhem a mesma autenticação sem necessitar de comunicação entre os mesmos.

# Composição do JWT

- Header
- Payload
- Signature

O cabeçalho é codificado utilizando o algoritmo Base64Url, antes de compor um JWT.

# Header

O Header é um objeto JSON que define informações sobre o tipo do token (typ), nesse caso JWT, e o algorítmo de criptografia usado em sua assinatura (alg), normalmente HMAC SHA256 ou RSA.

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Header

# Payload

O payload é um objeto Json com as Claims (informações) da entidade tratada, normalmente o usuário autenticado.

- Reserved claims: atributos não obrigatórios (mas recomendados) que são usados na validação do token pelos protocolos de segurança das APIs.

```
sub (subject) = Entidade à quem o token pertence, normalmente o ID
do usuário;
iss (issuer) = Emissor do token;
exp (expiration) = Timestamp de quando o token irá expirar;
iat (issued at) = Timestamp de quando o token foi criado;
aud (audience) = Destinatário do token, representa a aplicação que
irá usá-lo.
```

Geralmente os atributos mais utilizados são: **sub**, **iss** e **exp**.

- Public claims: atributos que usamos em nossas aplicações. Normalmente armazenamos as informações do usuário autenticado na aplicação.

```
name
roles
permissions
```

- Private claims: atributos definidos especialmente para compartilhar informações entre aplicações.

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

Payload

*Por segurança recomenda-se não armazenar informações confidenciais ou sensíveis no token.*

## Signature

A assinatura é a concatenação dos hashes gerados a partir do Header e Payload usando base64UrlEncode, com uma chave secreta ou certificado RSA.

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret
) secret base64 encoded
```

Signature

Essa assinatura é utilizada para garantir a integridade do token, no caso, se ele foi modificado e se realmente foi gerado por você.

## Resultado final

O resultado final é um token com três seções (header, payload, signature) separadas por "."—*ponto*.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJz
dWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gR
G9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab3
0RMHrHDcEfxjoYZgeFONFh7HgQ
```

Token JWT

# VULNERABILIDADES

Se a biblioteca aceita que um *token* seja validado sem especificar o algoritmo esperado, outra vulnerabilidade grave é aberta. Exatamente no caso esperarmos que o *token* use uma criptografia assimétrica e o atacante utiliza uma criptografia simétrica.

O problema com essa lógica é que o atacante pode obter a chave pública e assinar um *token* qualquer utilizando um algoritmo simétrico (HMAC) e indicar no cabeçalho o mesmo algoritmo. Assim quando um recurso protegido utilizar o mesmo algoritmo e a mesma chave o token será considerado válido, pois a **assinatura gerada** será igual a **assinatura do token**.

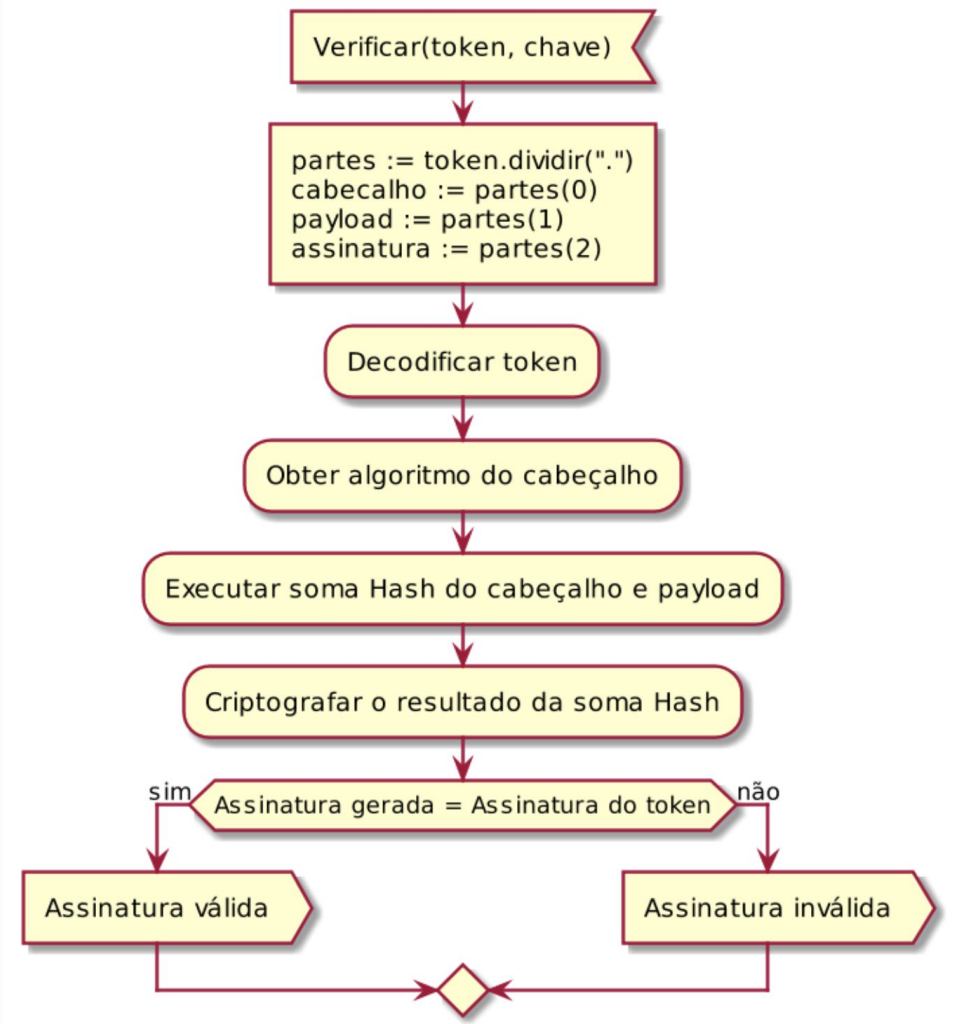



*Figura 6. Lógica de validação de assinatura JWT*

# VULNERABILIDADES

Lembrando que nesse caso como os tokens válidos estão sendo assinados com a chave privada os mesmos devem ser validados com a chave pública. Por isso o atacante terá sucesso, pois tem a certeza que o token está sendo validado com a chave pública.

# RECOMENDAÇÕES

Desenvolvedores deveriam exigir que o algoritmo utilizado para validação seja passado como parâmetro. Assim garante-se que será utilizado o algoritmo apropriado para a chave fornecida.

Caso seja necessária a utilização de mais de um algoritmo com chaves diferentes, a solução é atribuir um identificador para cada chave e indicá-la no campo kid do cabeçalho (*key identifier*, em inglês). Assim será possível inferir o algoritmo de acordo com a chave utilizada. Dessa maneira o campo alg não terá utilidade alguma além de, talvez, validar se ele indica o algoritmo esperado.

# RECOMENDAÇÕES

Ao utilizar uma implementação do padrão JWT, você deve auditar de maneira consistente se ela rejeita efetivamente algoritmos além do esperado. Assim a possibilidade de sucesso em ataques dessa natureza estarão quase nulos.

# Hands On

```
C:\Users\Avell
λ cd C:\MongoDB\bin

C:\MongoDB\bin
λ mongoimport --db reprograma --collection alunas --file alunas.json --jsonArray
2020-11-12T16:36:20.482-0300    connected to: mongodb://localhost/
2020-11-12T16:36:20.599-0300    34 document(s) imported successfully. 0 document(s) failed to import.
```

https://github.com/kellyjoany/tokenReprograma


src > config > auth.js > ...

```
1   module.exports = {
2     secret: 'ae074a5692dfb7c26aae5147e52ceb40',
3     expiresIn: '7d',
4   };
```

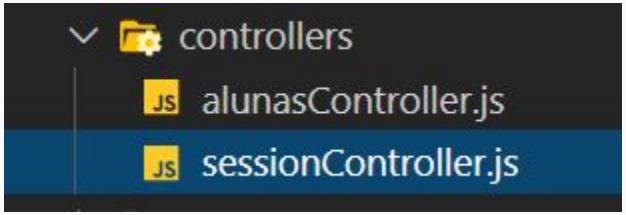
```
const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/reprograma',
{
    useNewUrlParser: true,
    useUnifiedTopology: true,
});

//rotas
const index = require("./routes/index")
const alunas = require("./routes/alunasRoute")
const sessions = require("./routes/sessionRoute")


app.use(express.json());

app.use(function(req, res, next) {
  res.header("Access-Control-Allow-Origin", "*")
  res.header(
    "Access-Control-Allow-Headers",
    "Origin, X-Requested-With, Content-Type, Accept"
  )
  next()
})

app.use("/", index)
app.use("/alunas", alunas)
app.use("/sessions", sessions)
```




src > controllers > sessionController.js > checkPassword

```
1   const jwt = require('jsonwebtoken');
2   const authConfig = require ('../config/auth');
3   const bcrypt = require("bcrypt");
4   const Alunas = require('../model/Alunas');
5
6   function checkPassword(passwordEntry, password) {
7     return bcrypt.compareSync(passwordEntry, password);
8   }
9
```

```
exports.accessToken = (req, res) => {
  try {
    const { name, password: passwordEntry } = req.body;

  } catch (e) {
    return res.status(401).json({ error: 'erro' });
  }
}
```

```
Alunas.findOne({nome: name})
  .then((user) => {
      const {id, nome, hashPass } = user;

  })
  .catch((e) => {
    return res.status(401).json({ error: 'user not found' });
  });
```

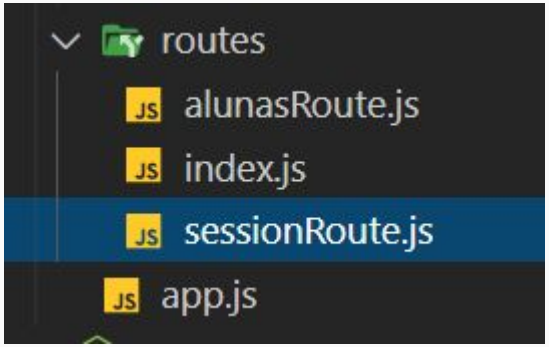
```
.then((user) => {
    const {id, nome, hashPass } = user;

    try {
      checkPassword(passwordEntry, hashPass);
    } catch(e) {
      return res.status(401).json({ error: 'password does not match' });
    }

    try {
      return res.json({
        user: {
          id,
          nome,
        },
        token: jwt.sign({ id }, authConfig.secret, {
          expiresIn: authConfig.expiresIn,
        }),
      });
    } catch (e) {
      return res.status(401).json({ error: 'erro' });
    }
```

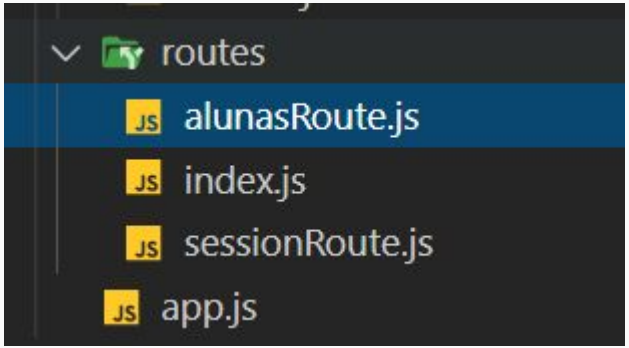

src > middlewares > auth.js > ...

```
1   const jwt = require('jsonwebtoken');
2   const authConfig = require ('../config/auth');
3   const { promisify } = require('util');
4
5   module.exports = async (req, res, next) => {
6     const authHeader = req.headers.authorization;
7     if (!authHeader) {
8       return res.status(401).json({ error: 'Token not provided' });
9     }
10
11    const [, token] = authHeader.split(' ');
12
13    try {
14      const decoded = await promisify(jwt.verify)(token, authConfig.secret);
15      req.userId = decoded.id;
16      return next();
17    } catch (err) {
18      return res.status(401).json({ error: 'Token invalid' });
19    }
20  };
21
```




src > routes > JS sessionRoute.js > <unknown>

```
1   const express = require("express")
2   const router = express.Router()
3   const controller = require("../controllers/sessionController")
4
5   router.post("/", controller.accessToken)
6
7   module.exports = router
```




src > routes > alunasRoute.js > ...

```
1   const express = require("express")
2   const router = express.Router()
3   const controller = require("../controllers/alunasController")
4   const authMiddleware = require("../middlewares/auth")
5
6   router.get("/", controller.get)
7   router.post("/", controller.post)
8   router.use(authMiddleware);
9   router.get("/nasceuSp", controller.getSp)
10  router.get("/:id", controller.getById)
11
12  module.exports = router
13
```