<WA/>
2024

Authorization

Stateless Authorization Mechanisms

Enrico Masala
Antonio Servetti

https://flaviocopes.com/jwt/

https://stackabuse.com/authentication-and-authorization-with-jwts-in-express-js/

Can you do the requested operation?

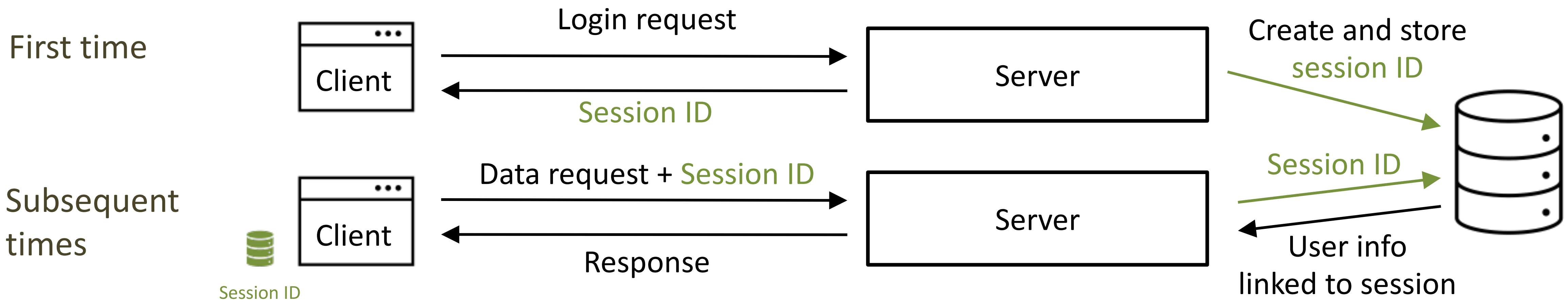
AUTHORIZATION IN WEB APPLICATIONS

Authorization after authentication

- Two approaches to handle authorization after authentication:
  - Stateful server
  - Stateless server

Stateful Server

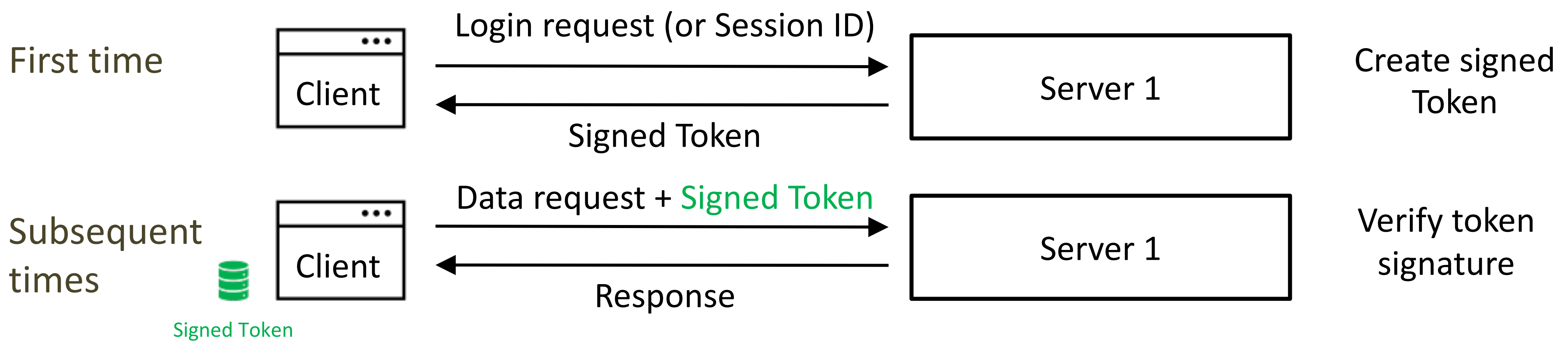
- The server **remembers** the valid session IDs and the associated user info (after login)
- Associated info cannot be maliciously altered: the trusted version is only in the **server**
- Each time a request arrives for a restricted resource, the server **retrieves** the info associated with the session and decides if the associated user is authorized or not
- Works best with a single server that manages everything



https://stackabuse.com/authentication-and-authorization-with-jwts-in-express-js/

Stateless Server

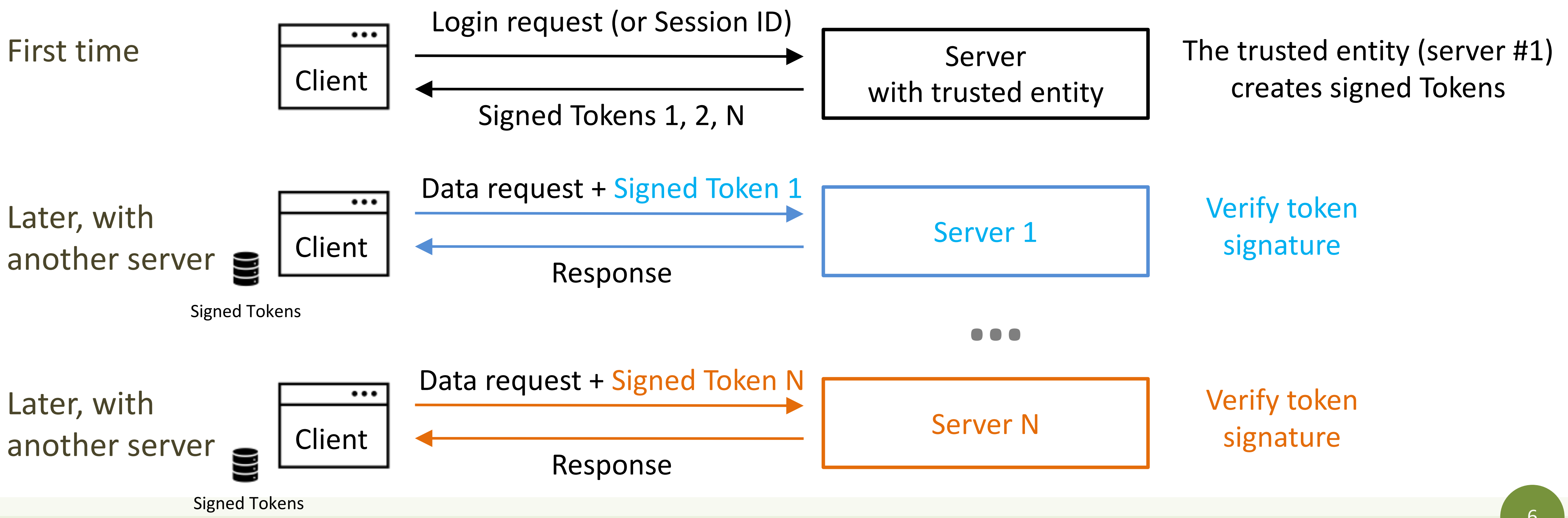
- A trusted entity **signs a payload** which contains information about what can be accessed, the user info (if needed), and when the authorization expires
- The client gets the signed payload and **stores** it
- Each time a request arrives for a restricted resource, the receiving **server verifies the signature**, extracts and uses the information
  - which is trusted because it is signed



https://stackabuse.com/authentication-and-authorization-with-jwts-in-express-js/

Stateless Server

- Works best where there are multiple servers which cannot easily share session information (note: tokens can also be the same for all servers)


First time
Client
Login request (or Session ID)
Signed Tokens 1, 2, N
Server with trusted entity
The trusted entity (server #1) creates signed Tokens
Later, with another server
Signed Tokens
Client
Data request + Signed Token 1
Response
Server 1
Verify token signature
Later, with another server
Signed Tokens
Client
Data request + Signed Token N
Response
Server N
Verify token signature

Note on Stateless Servers

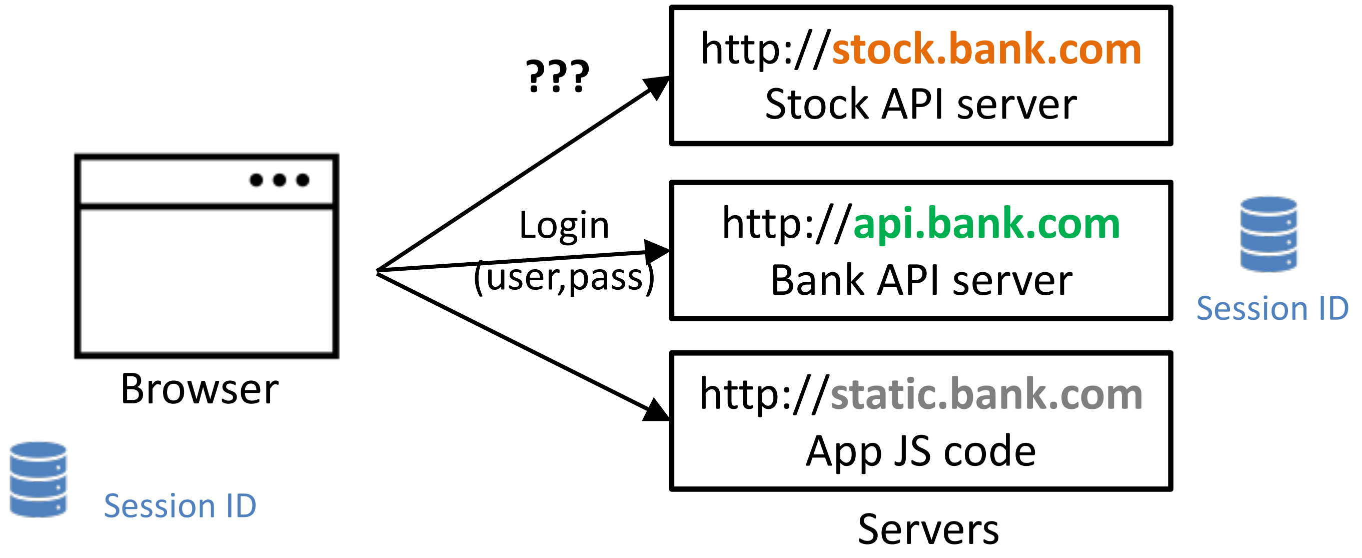
- Many schemes exists to implement complex authorization flows, for different purposes (access permission, Single Sign On, etc.)
- Examples:
  - OAuth
  - SAML
  - OpenID

$\left.\begin{array}{l}\text{OAuth}\\ \text{SAML}\\ \text{OpenID}\end{array}\right\}$ *Out of scope of this course*

- For this course: a simple example is provided where a token allows to access restricted information without an active session

Example

- SPA uses an API server to perform its operations (e.g., bank transactions) and would like to use a second API server to retrieve additional user information (e.g., non-free stock exchange data)

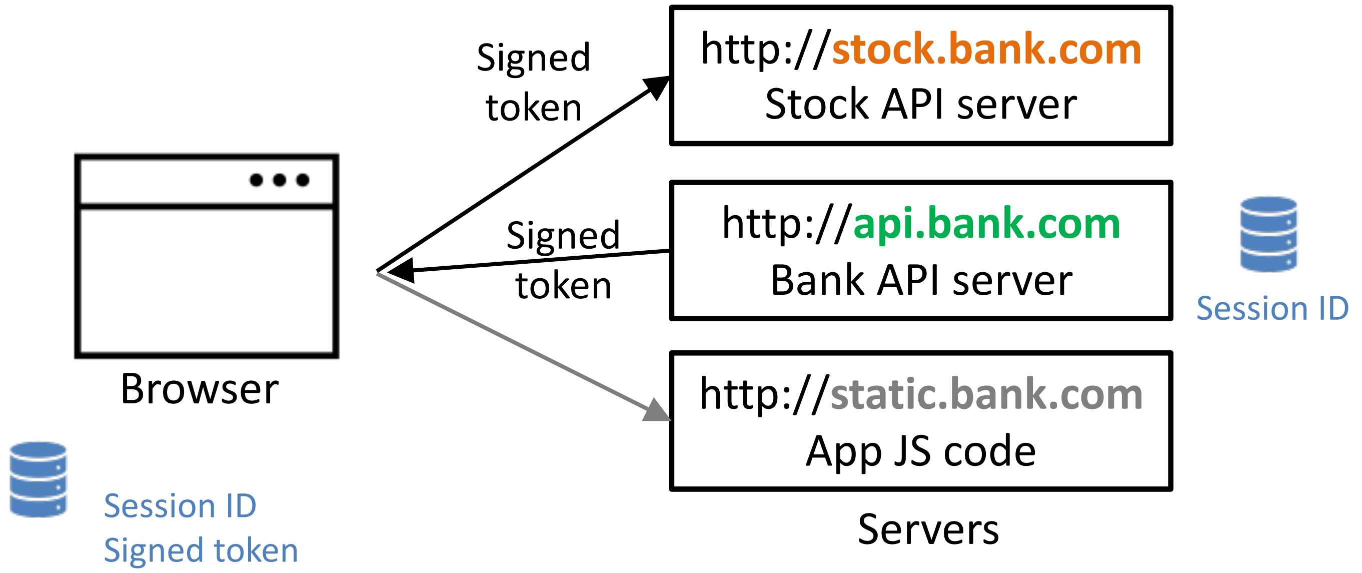


1. Client loads SPA code
2. User perform login with api.bank.com and operates there as usual
   - Balance, money transfers etc.
3. Client would like to access and show non-free stock data **without having the user to login again** with stock.bank.com

Note: actual implementations may be more complex (e.g., additional servers to share secrets etc.)

Solution

- After authentication, the server api.bank.com provides a signed token that authorizes the client to access non-free data from stock.bank.com



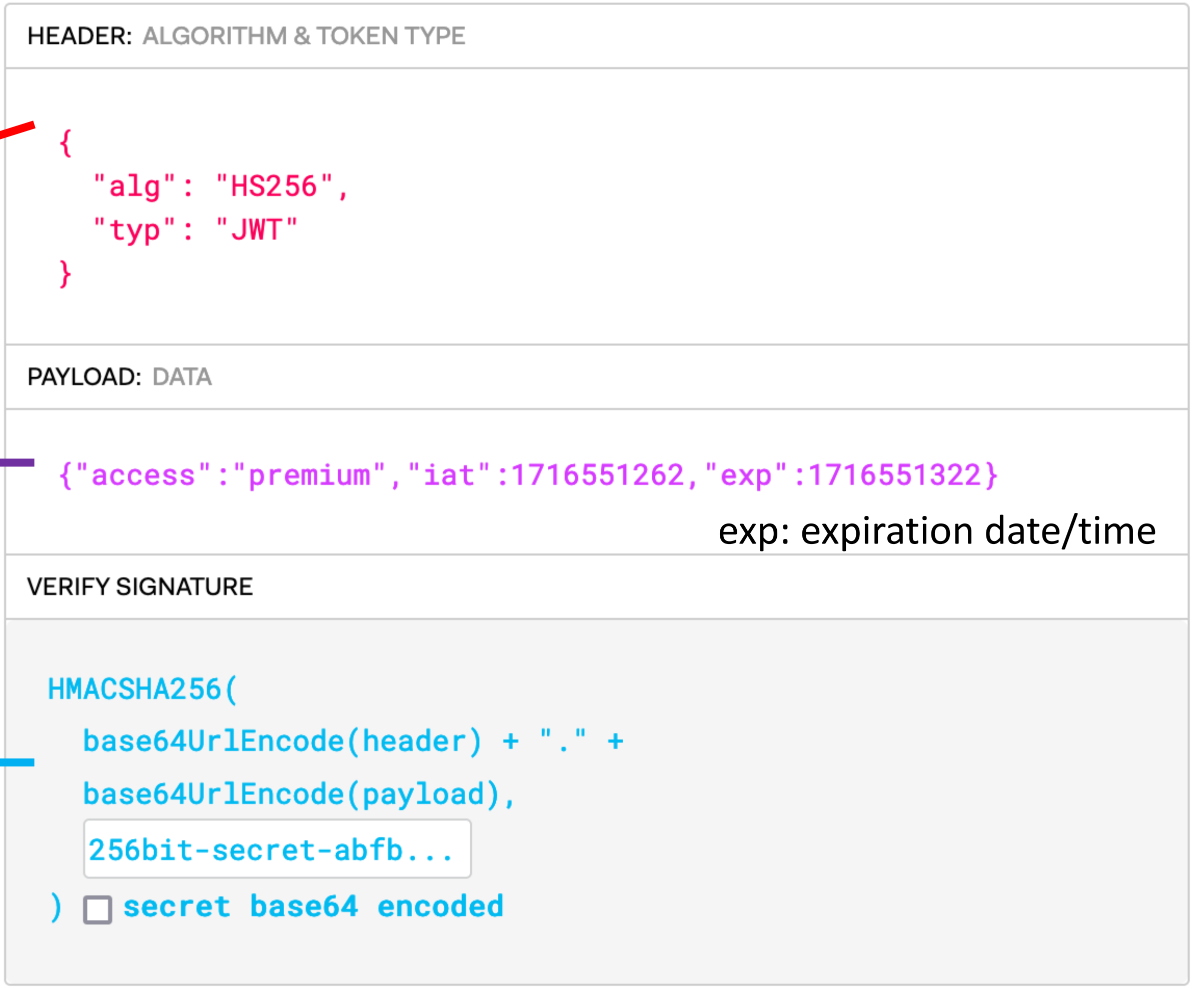
- No need to talk between stock.bank.com and api.bank.com at runtime
- They only share a sign/verify mechanism (e.g., a secret)

JSON Web Token: a mechanism for authorization

- Standardized in RFC 7519
- In short, JSON Web Tokens (JWTs) are digitally signed JSON payloads, encoded in a URL-friendly string format
- A JWT can contain any payload in general
- A common use case is to store authorization levels and user info
- JWTs used for authorization should contain at least:
  – info about the permissions
  – an expiration timestamp

JWT Example

```
eyJhbGciOiJIUzI1NiIsInR5cCI6
IkpXVCJ9.
eyJhY2Nlc3MiOiJwcmVtaXVtIiwi
aWF0IjoxNzE2NTUxMjYyLCJleHAi
OjE3MTY1NTEzMjJ9.
W9loagJ6ClHveg6aIVYeE5tF1zlF
SkR8_FUiW8m677k
```



HEADER: ALGORITHM & TOKEN TYPE

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

PAYLOAD: DATA

```
{"access":"premium","iat":1716551262,"exp":1716551322}
```

exp: expiration date/time

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  256bit-secret-abfb...
) ☐ secret base64 encoded
```

https://jwt.io/ (also useful to read the content of header and payload, which is cleartext, not encrypted)

JWT Pros

- To confirm JWT payload validity, **validating the signature is enough**
  - No need for further interaction with username, password, etc.
  - Very well suited for stateless APIs that receive a single request and must provide an answer
- No need to contact the entity that provided the JWT at runtime (e.g., the authentication server)
  - Of course, verification could go to the entity and ask, but this would make the token-based approach pointless
- No need to keep the token in server memory nor in the server storage (files, DBs etc.) between HTTP requests

JWT Cons

- Difficult to make JWT invalid sooner than the expiration date/time
  - Change secret used for the signature (but this invalidates **all** tokens)
  - Keep a list of blacklisted/whitelisted token (but this requires stateful server)
- For this reason: better to have a short expiration date/time
  - If authorization is still needed after expiration date/time?
  - a new token must be requested before expiration
    - This is a complex issue to manage, actual implementations typically involve a second token (refresh token) as in, e.g., OAuth2 *(out of the scope of this course)*

JWT in practice

- It must be kept secret, as any other authentication/authorization token (cookies included)
  - sent over HTTPS only, at least in production, so that it cannot be sniffed and/or intercepted
- Must be sent with each request to the API server
- The server receiving the token must have a method to validate the legitimacy of the JWT
  - Depends on how the signature is implemented

JWT Signing Algorithms

- Many
- Two important categories
  - Single secret key (Hash-based)
  - Public / private key (RSA, ECDSA)

- HMAC + SHA256
- RSASSA-PKCS1-v1_5 + SHA256
- ECDSA + P-256 + SHA256
- …

https://auth0.com/blog/json-web-token-signing-algorithms-overview/

Keys for Signing

Single key

- Key is the same between authentication server and verifying server
- Key must be long enough (at least as the hash length, i.e. 256 bits = 32 bytes or characters)
- Key must be duly protected
  - Can be used to forge JWT tokens

Public/private keys

- Private key is used **only** by the authentication server to initially sign the JWT token
- API servers can be many and only need the public key: better security
  - Public keys cannot be used to forge JWT tokens

https://stackabuse.com/authentication-and-authorization-with-jwts-in-express-js/

Using JWT tokens in practice

JWT IN PRACTICE

Recommendations

- Create an endpoint that, after checking the authorizations of the user, returns the authorization token
- Receive the signed JWT and store it in the application memory
  - For instance, in a React State
- Send it with the requests that need it
  - Not as cookie: usually it is sent to another domain where a cookie cannot be sent

- Note: a session will NOT be established
  - It is possible to do it in passport using a JWT as cookie, but it is not the purpose of the presented example

JWT in express.js

- Several libraries are available
- Most frequently used:
  - `express-jwt` https://github.com/auth0/express-jwt
  - `jsonwebtoken` https://github.com/auth0/node-jsonwebtoken/

- `npm install express-jwt` (*middleware*)
- `npm install jsonwebtoken` (*utilities to encode info and sign JWTs*)

express-jwt

https://github.com/auth0/express-jwt

- Configuration through an object `jwt( { … config props … } );`
- Most important properties are:
  - `secret`: sufficiently long random string needed to verify signature
  - `algorithms`: a set of algorithms, e.g., `["HS256"]`
    - Note: do not use both symmetric and asymmetric algorithms
  - `credentialsRequired`: (optional) if false, allow access to unauthorized users (e.g., for logging or other purposes)
  - `getToken()`: (optional) function to extract token from the HTTP request (e.g., needed in case the token must be retrieved from a cookie)
- Token is automatically extracted from the HTTP `Authentication:` header

jsonwebtoken

https://github.com/auth0/node-jsonwebtoken/

- Used to **create the JWT** with a specified sign method
- `jwt.sign(payload, secretOrPrivateKey, [options, callback])`
  - Can be used synchronously or asynchronously (providing a callback)
  - Main options:
    - `expiresIn`: seconds from now when the token will expire
    - `algorithm`: the algorithm to be used for signature
    - `noTimestamp`: used not to include, in the payload, the timestamp when the token is issued
    - … others to include standard fields in the payload (issuer, audience, subject, etc.)
- Other methods (verify, decode) are present but directly used by the previous middleware

Import and Headers

Server 1

```
// In the server that issues the token
const jsonwebtoken = require('jsonwebtoken');

const jwtSecret = '6xvL4xkAAbG49h_a_long_random_secret__min_256bit';
// must be the same as the other server
```

- The secret must be the same in both servers!

Server 2

```
// In the server that verifies the token
// import expressjwt as jwt
const { expressjwt: jwt } = require('express-jwt');

const jwtSecret = '6xvL4xkAAbG49h_a_long_random_secret__min_256bit';
// must be the same as the other server
```

Route that Generates the Token

Server 1

```
const expireTime = 60; //seconds

app.get('/api/auth-token', isLoggedIn, (req, res) => {
  // Payload to adapt to the use case
  // in this case the .level was retrieved from the DB with the user info
  const payloadToSign = { access: req.user.level, user: req.user.id, authId: 1234, … };
  const jwtToken = jsonwebtoken.sign(payloadToSign, jwtSecret, {expiresIn: expireTime});
  res.status(200).json({token: jwtToken});
});
```

- The same route can generate different tokens depending on user role
- It could even generate a generic access token for non-authenticated requests
  - In general, this is less useful

Getting the Token in the Application

```
API.getAuthToken()
   .then((resp) => { setAuthToken(resp.token); } );
```

Client

```
// In API.js
async function getAuthToken() {
  const response = await fetch(URL+'/auth-token', {
    credentials: 'include'
  });
  const token = await response.json();
  if (response.ok) {
    return token;
  } else {
    throw token;  // e.g., an object with the error coming from the server
  }
}
```

Client

Sending the Token in a Request

Client

```
async function getExternalInfo(authToken) {
  // retrieve info from an external server
  // where info can be accessible only via JWT token
  const response = await fetch('http://localhost:3002/api/stock-quotes', {
    headers: { 'Authorization': `Bearer ${authToken}` }
  });
  const info = await response.json();
  if (response.ok) {
    return info;
  } else {
    throw info;  // expected to be a json object (coming from the server)
                 // with info about the error
  }
}
```

Protecting APIs

Server 2

```
// In the server that verifies the token
const { expressjwt: jwt } = require('express-jwt'); // import expressjwt as jwt

app.use(
  jwt({
    secret: jwtSecret,
    algorithms: ["HS256"],
  })
);
// After this app.use(…), the APIs in the rest of the code will require authentication
…
// APIs
app.get('/api/stock-quotes', …
 // Without a valid token, an error will be raised automatically in the HTTP response
…
```

Note that the token is automatically extracted from the HTTP header:

`Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpX…`

Unauthorized Requests

- The JWT middleware throws an exception if not authorized
- To handle the error, you may provide a custom middleware function

Server 2

```
app.use( function (err, req, res, next) {
    if (err.name === 'UnauthorizedError') {
       // Example of err content generated by the middleware:
       //   {"code":"invalid_token","status":401,"name":"UnauthorizedError",
       //    "inner":{"name":"TokenExpiredError","message":"jwt expired",
       //    "expiredAt":"2024-05-23T19:23:58.000Z"}}
       res.status(401).json(  // can be adapted as appropriate
         { errors: [{ 'param': 'Server', 'msg': 'Authorization error', 'path': err.code }] });
    } else {
      next();
    }
});
```

Accessing the JWT payload

- The JWT payload is accessible in **`req.auth`**

Server 1

```
// This was done in server 1
// … const token = jsonwebtoken.sign({ access: 'premium'}, jwtSecret, {expiresIn: …});
```

Server 2

```
app.get('/api/stock-quotes', (req, res) => {
  // Extract payload from JWT payload
  const level = req.auth.access;    // 'premium'
  // Do whatever is required with the info extracted from the JWT payload
  dao.retrieveInfo(level)
    .then((data) => res.json(data))
    .catch((err) => res.status(503).json(dbErrorObj));
});
```

Authorization is a complex problem

- Never invent your own mechanism!
- Use standardized, well tested, ones!

License