JSON Web Tokens Deconstructed

/swapnil-bhattacharjee-ghy

JSON Web Tokens aka JWT is an open standard for securely transmitting data b/w parties as a JSON object. It is defined in RFC 7519.

The most common use case for JWT is Authorization in Web Applications.

In applications using JWT auth, the client sends the issued JWT token on every request in the Authorization header.

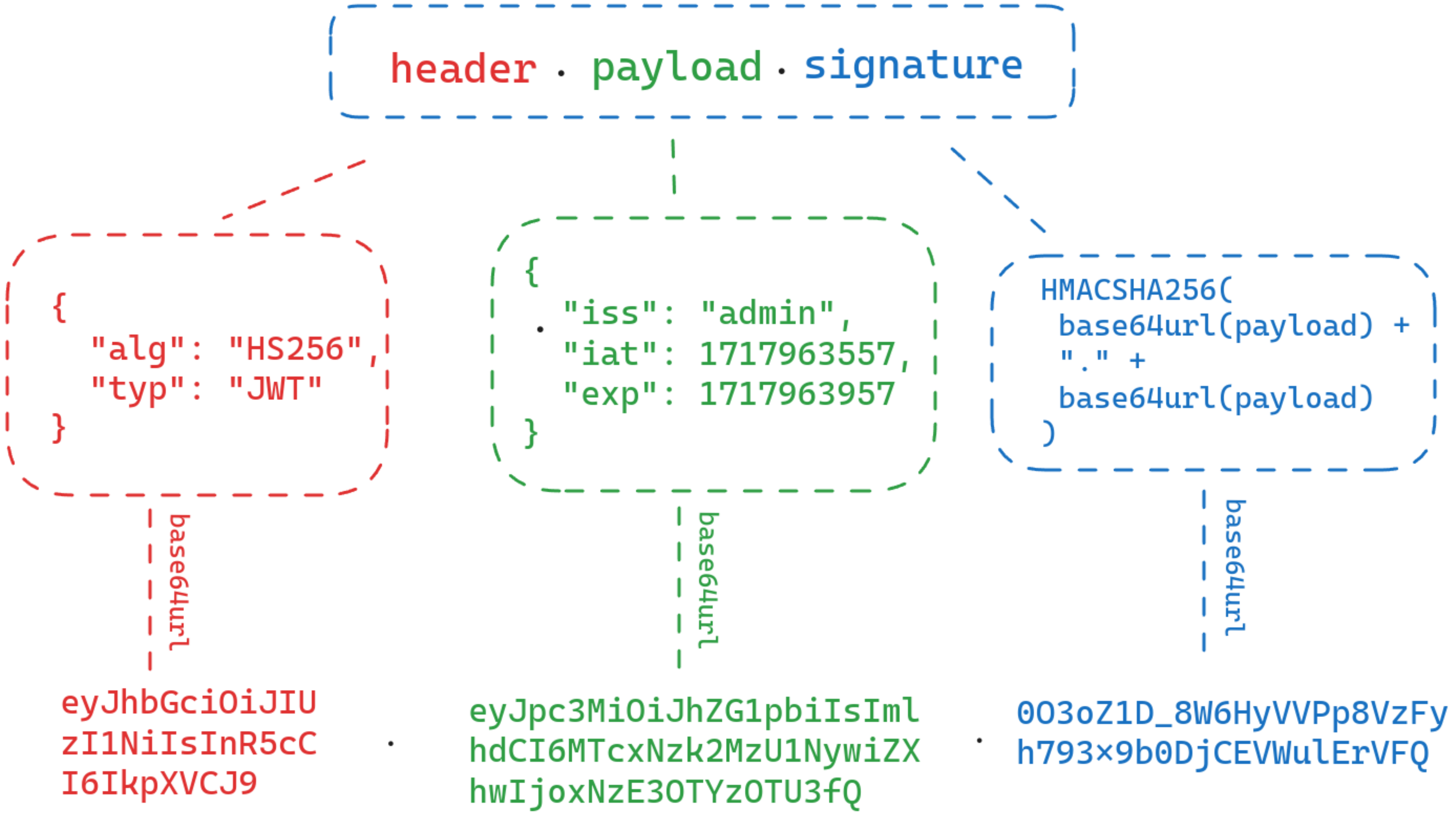
```
Authorization: Bearer <JWT>
```

In this post, we will understand the structure of JWTs - their makeup, and how we can implement functions to sign and verify JWTs from scratch with Node.js and TypeScript

JSON Web Tokens Deconstructed

/swapnil-bhattacharjee-ghy

A JWT is just three base64 url encoded strings delimited by the period '.' character.



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

```
{
  "iss": "admin",
  "iat": 1717963557,
  "exp": 1717963957
}
```

```
HMACSHA256(
 base64url(payload) +
 "." +
 base64url(payload)
)
```

```
eyJhbGciOiJIU
zI1NiIsInR5cC
I6IkpXVCJ9
.
eyJpc3MiOiJhZG1pbiIsIml
hdCI6MTcxNzk2MzU1NywiZX
hwIjoxNzE3OTYzOTU3fQ
.
0O3oZ1D_8W6HyVVPp8VzFy
h793x9b0DjCEVWulErVFQ
```

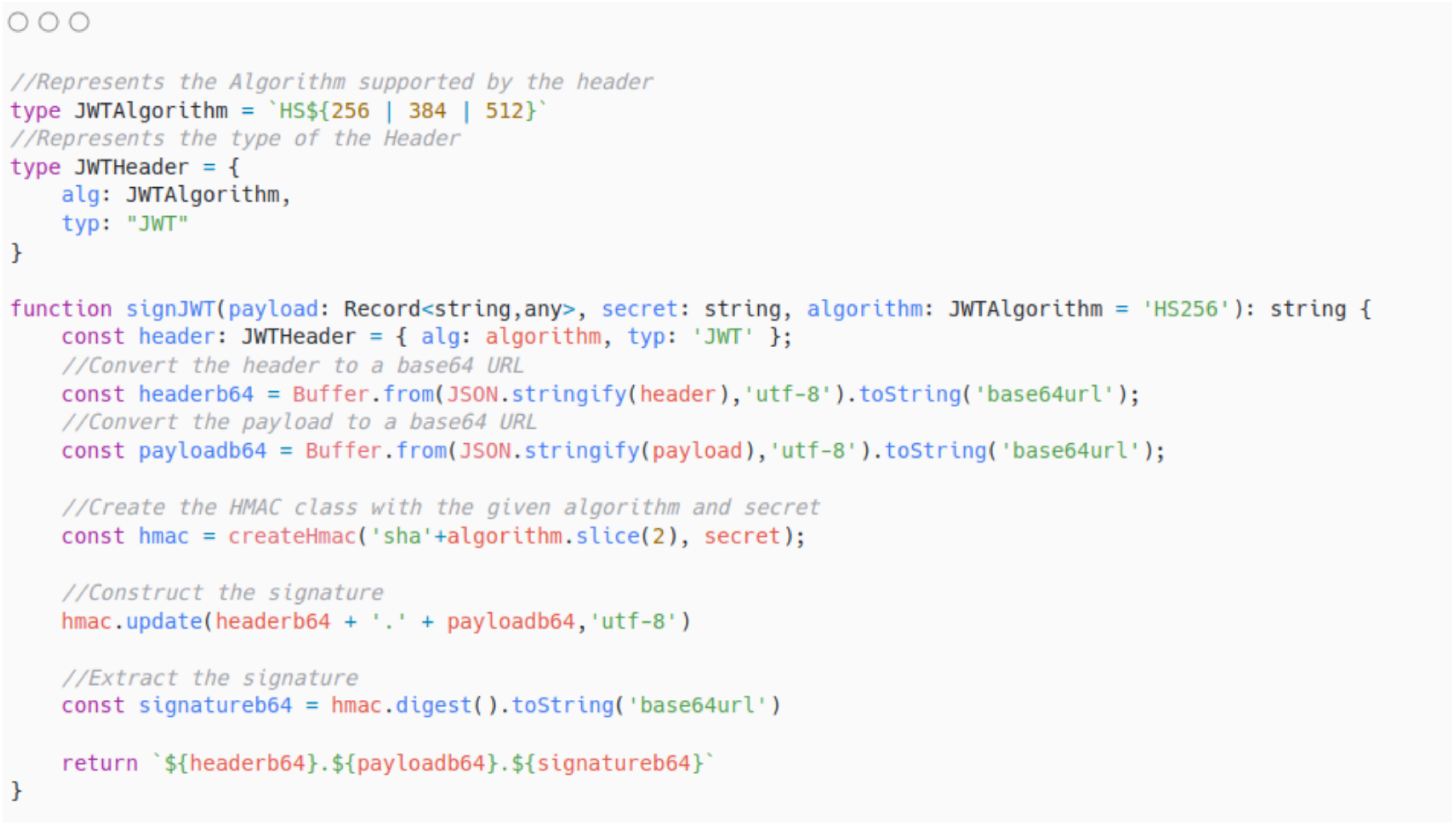
/swapnil-bhattacharjee-ghy

Things to know about JWT in the context of auth

- The payload and header are not encrypted. You shouldn't place any sensitive information in the payload.
- JWTs are signed so that the server can verify the issuer of the token
- JWTs use base64Url encoding, not base64 which are slightly different encoding schemes. Check out RFC 4648 to learn more
- Since JWTs are based on JWA and JWE standards, they are meant to be shared over URIs, that's why URL encoding is used instead of plain old base64
- RFC 7519 lists the following algorithms that can be used for signing JWTS - HMAC, RSA, and ECD with SHA 256, 384 & 512 algorithms
- The payload contains registered claims that are well defined in RFC 7519 Section 4.1. These are used for communication b/w parties
- a few such claims are issuer (is), iat (issued at) & exp (expiry)
- The date claims are defined as the NumericDate type, which expresses the number of seconds, not milliseconds, since the UNIX epoch

Implementing the Signer in Node.JS

/swapnil-bhattacharjee-ghy



```
//Represents the Algorithm supported by the header
type JWTAlgorithm = `HS${256 | 384 | 512}`
//Represents the type of the Header
type JWTHeader = {
    alg: JWTAlgorithm,
    typ: "JWT"
}

function signJWT(payload: Record<string,any>, secret: string, algorithm: JWTAlgorithm = 'HS256'): string {
    const header: JWTHeader = { alg: algorithm, typ: 'JWT' };
    //Convert the header to a base64 URL
    const headerb64 = Buffer.from(JSON.stringify(header),'utf-8').toString('base64url');
    //Convert the payload to a base64 URL
    const payloadb64 = Buffer.from(JSON.stringify(payload),'utf-8').toString('base64url');

    //Create the HMAC class with the given algorithm and secret
    const hmac = createHmac('sha'+algorithm.slice(2), secret);

    //Construct the signature
    hmac.update(headerb64 + '.' + payloadb64,'utf-8')

    //Extract the signature
    const signatureb64 = hmac.digest().toString('base64url')

    return `${headerb64}.${payloadb64}.${signatureb64}`
}
```

Implementing the Verifier in Node.JS

/swapnil-bhattacharjee-ghy

```
//Return type of the Verifier
type VerifyJWTReturn = {
    valid: boolean,
    payload: Record<string,any> | null
}

//Regular expressions for the supported algorithms and the JWT
const AlgorithmRegex = /^HS(256|384|512)$/
const JWTStringRegex = /^([\w\d-_]+)\.([\w\d-_]+)\.([\w\d-_]+)$/

function verifyJWT(jwt: string, secret: string): VerifyJWTReturn {
    //If the string doesn't match the Regex, return
    if(!JWTStringRegex.test(jwt)) return { valid: false, payload: null}
    try {
        //Get the components by splitting on the period character
        const [headerb64, payloadb64, signature] = jwt.split('.') as [string, string, string];

        //Convert the base64urls to JavaScript objects
        const header = JSON.parse(Buffer.from(headerb64,'base64url').toString('utf-8')) as JWTHeader;
        const payload = JSON.parse(Buffer.from(payloadb64,'base64url').toString('utf-8')) as Record<string,any>;

        //Test whether the signing algorithm is supported
        if(!AlgorithmRegex.test(header.alg)) return { valid: false, payload: null };

        //Create the HMAC class with the given algorithm and secret
        const hmac = createHmac('sha'+header.alg.slice(2), JWT_SECRET);

        //Construct the server signature
        hmac.update(headerb64 + '.' + payloadb64,'utf-8');
        const computedSignature = hmac.digest().toString('base64url');

        //Check if the client and server signature match. If they match the JWT is verified to have been issued
        //by this server
        if(signature === computedSignature) return {valid: true, payload }

        //otherwise the token cannot be trusted
        else return { valid: false, payload: null };

    } catch(e) {
        return { valid: false, payload : null}
    }
}
```

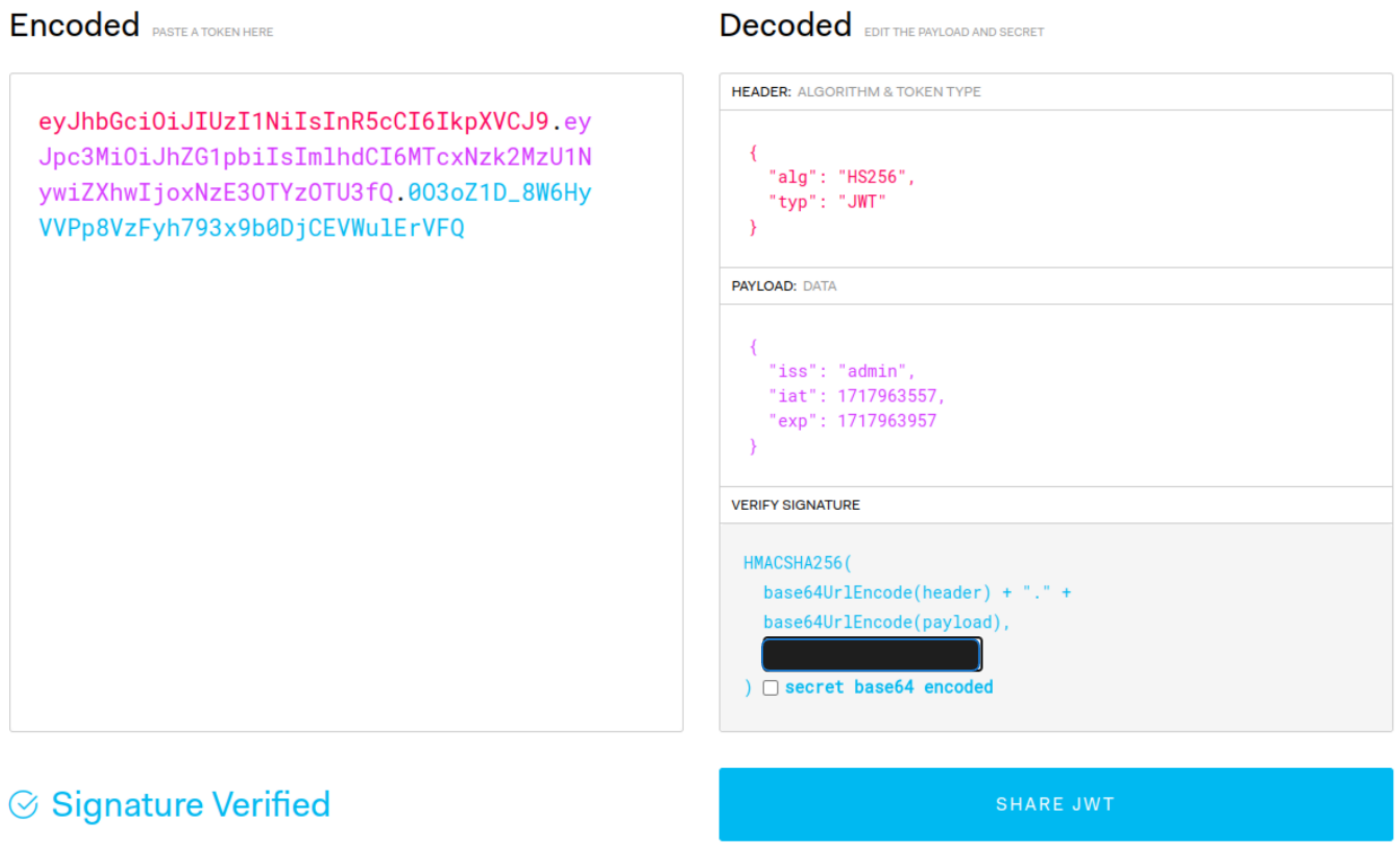
Validating our implementation

/swapnil-bhattacharjee-ghy

```
signJWT({"iss":"admin","iat":1717963557,"exp":1717963957},"<SECRET>")
```

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJhZG1pbiIsImlhdCI6MTcx
Nzk2MzU1NywiZXhwIjoxNzE3OTYzOTU3fQ.0O3oZ1D_8W6HyVVPp8VzFyh793×9b0DjCE
VWulErVFQ
```

Let's Plug this into jwt.io



Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ey
Jpc3MiOiJhZG1pbiIsImlhdCI6MTcxNzk2MzU1N
ywiZXhwIjoxNzE3OTYzOTU3fQ.0O3oZ1D_8W6Hy
VVPp8VzFyh793x9b0DjCEVWulErVFQ
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

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

PAYLOAD: DATA

```
{
  "iss": "admin",
  "iat": 1717963557,
  "exp": 1717963957
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  [redacted]
) ☐ secret base64 encoded
```

Signature Verified

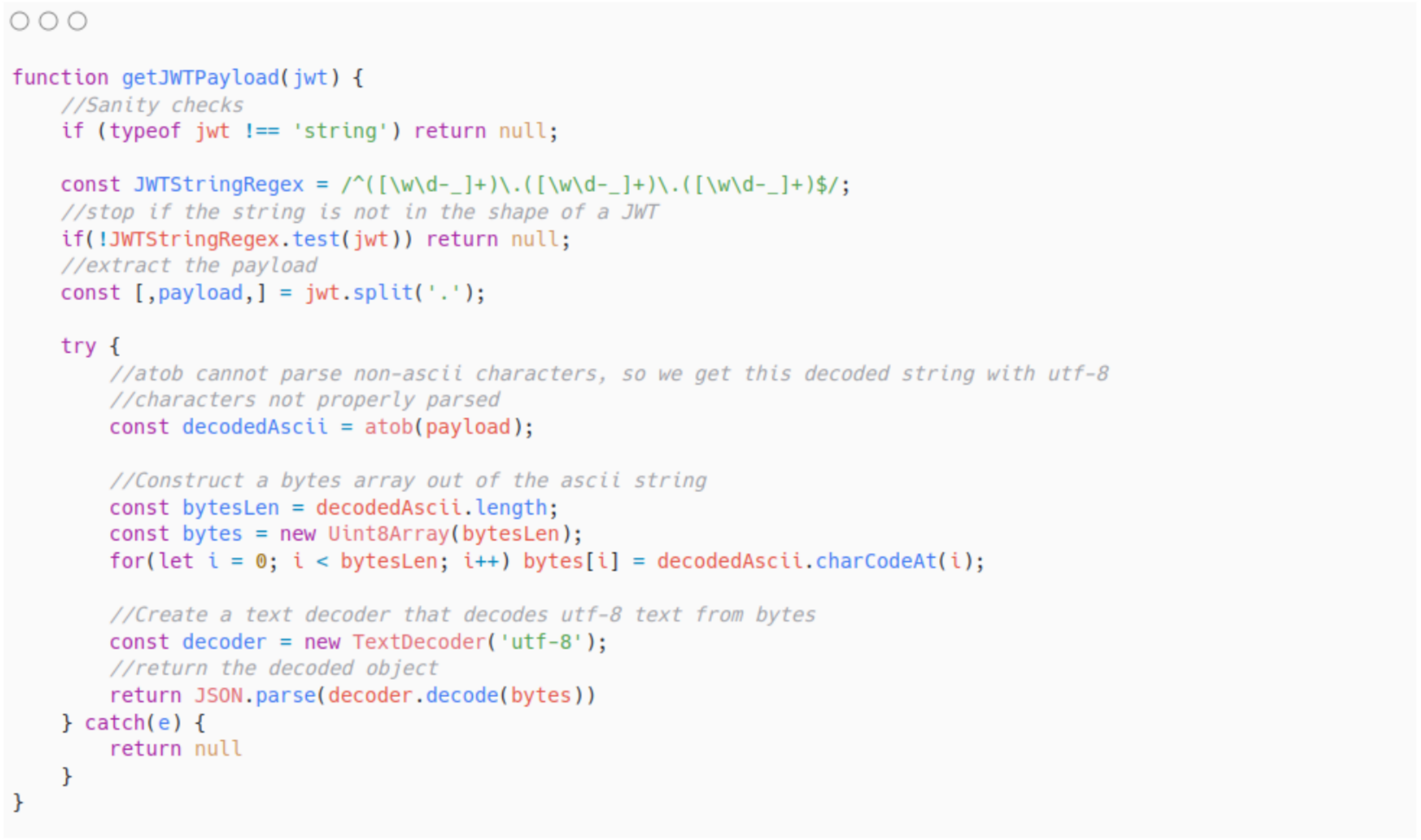
SHARE JWT

Now we have a compliant implementation to use in our web application

Working with JWTs in the Browser

/swapnil-bhattacharjee-ghy

Working with JWTs in the browser is a bit more involved as Client Side JS doesn't have Buffer or its methods. So, we have to write custom code to extract the data out of JWT payloads. Here's one such implementation.



```javascript
function getJWTPayload(jwt) {
    //Sanity checks
    if (typeof jwt !== 'string') return null;

    const JWTStringRegex = /^([\w\d-_]+)\.([\w\d-_]+)\.([\w\d-_]+)$/;
    //stop if the string is not in the shape of a JWT
    if(!JWTStringRegex.test(jwt)) return null;
    //extract the payload
    const [,payload,] = jwt.split('.');

    try {
        //atob cannot parse non-ascii characters, so we get this decoded string with utf-8
        //characters not properly parsed
        const decodedAscii = atob(payload);

        //Construct a bytes array out of the ascii string
        const bytesLen = decodedAscii.length;
        const bytes = new Uint8Array(bytesLen);
        for(let i = 0; i < bytesLen; i++) bytes[i] = decodedAscii.charCodeAt(i);

        //Create a text decoder that decodes utf-8 text from bytes
        const decoder = new TextDecoder('utf-8');
        //return the decoded object
        return JSON.parse(decoder.decode(bytes))
    } catch(e) {
        return null
    }
}
```