<WA1/>

2020

Authentication

For some, but not for all

Enrico Masala

Fulvio Corno

Luigi De Russis

Outline

- The need for authentication
- HTTP sessions
- The JWT approach
- Implementation in React

https://flaviocopes.com/cookies/

Who are you?

AUTHENTICATION IN WEB APPLICATIONS

Authentication vs authorization

- Authentication
  - Verify you are who you say you are (identity)
  - Typically done with credentials (e.g., username, password)
  - Allows a personalized user experience

- Authorization
  - Decide if you have permission to access a resource
  - Granted authorization rights depends on the identity as established during authentication

Both are often used in conjunction to protect access to a system

Authentication and authorization

- Developing authentication and authorization mechanisms:
  - is complicated
  - is time-consuming
  - is prone to errors
  - may require interacting with third-party systems (log-in with google, facebook, …)
- Involve both client and server
- Rely on advice by security experts!

HTTP sessions

- Because **HTTP is stateless**, each HTTP request is independent and must be self-contained
- While communication, it is often desirable that information about previous interactions is maintained
- A **session** is an established communication between entities that may involve one or more messages in each direction, and typically at least one party keep some state information
- Example: client authenticates with the server, the server establish a session in which the client is considered authenticated

Session ID

- Basic mechanism to maintain session
- Upon authentication, the client receives from the authentication server a session ID that allows to recognize subsequent HTTP requests as authenticated
- Such information must be stored on the client side
- Such information must be sent by the client every time it sends a request which is part of the session
- Typical implementation in HTTP: **cookie**

https://en.wikipedia.org/wiki/HTTP_cookie
https://tools.ietf.org/html/rfc6265

Cookie

- RFC 6265
- Automatically handled by the browser. Uses HTTP headers:
  - `SetCookie:` server → client; the browser stores the cookie locally
  - `Cookie:` client → server
- Properties
  - name, value, domain (including port), path, secure, httpOnly, expiration date (optional)
- Stored by the browser in its cookie storage
- Always sent automatically by the browser when sending requests to the **domain** and **path** to which the cookie belong

Security of the session ID

- Any session ID must always travel on encrypted connections (HTTPS) to avoid being intercepted
- By stealing session ID, one can impersonate an authenticated client

- Several attack mechanisms are possible in a web application
  - XSS: Cross-site scripting: malicious JS code stealing the session ID
  - CSRF: Cross-site request forgery: make the browser perform an unwanted action by inducing the user to click on links (phishing email, social media post)
    `<a href="https://api.bank.com/sendMoney">Go to homepage</a>`

Mitigation techniques for XSS and CSRF

- XSS: Prevent any JS code (legit or not) from accessing the session ID
  - Use cookie with the httpOnly attribute

- CSRF: Do not rely only on cookie to authenticate requests on server side
  - Use additional headers derived by a shared secret and added via JS
  - Specific libraries exist for this purpose
  - Some web frameworks incorporate CSRF protection by default

- Rely on advice by security experts!

Alternatives to cookies

- From client to server
  - Separate Custom HTTP header
  - Hidden query parameters
- From server to client
  - Request body
  - Separate Custom HTTP header
- Handling by JS code in the browser, specific code on the server side
- Store in HTML5 `localStorage` or `sessionStorage`
  - Values accessible by JS but only from the same origin (schema, domain, port)

Cookies vs alternative approach: pros and cons

	Cookies	Other approaches
Pros	• Automatically handled by the browser • Can be made inaccessible to JS code (httpOnly option) to prevent access from malicious JS code (XSS) • Can be sent on secure connections only (secure option)	• More flexibility (not restricted to cookie APIs) • Sent only when required by JS
Cons	• Cannot be sent by third parties • Always sent for any request: expose to CSRF attack	• Need to explicitly manage storage (localStorage, sessionStorage) • Can be accessed by any (malicious) script in the page (XSS)

https://flaviocopes.com/jwt/

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

Can you do the requested operation?

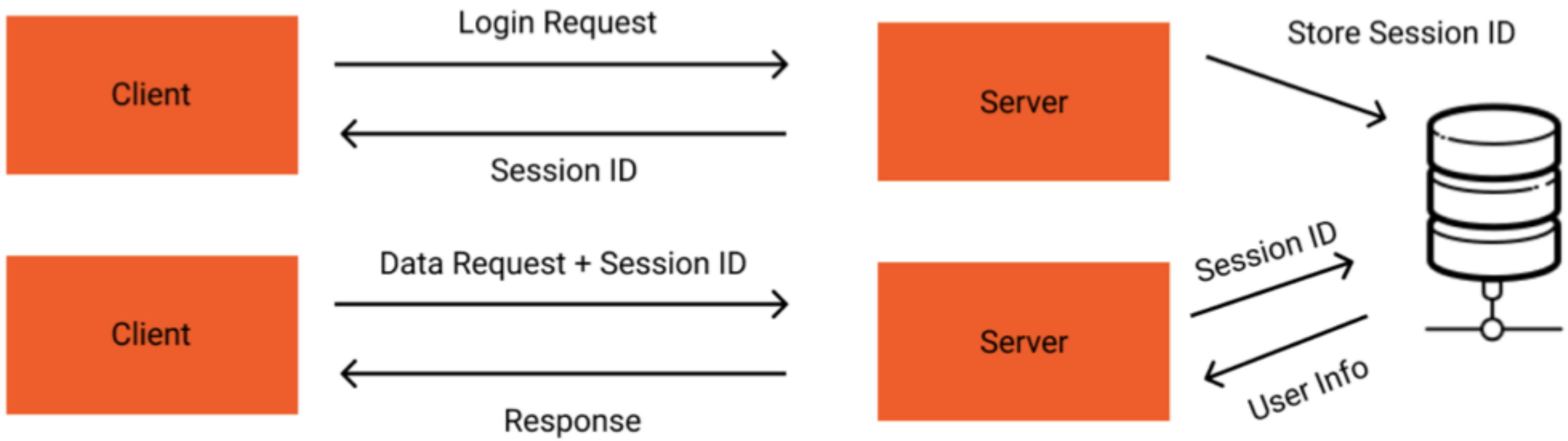
AUTHORIZATION IN WEB APPLICATIONS

Authorization after authentication

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

Stateful server

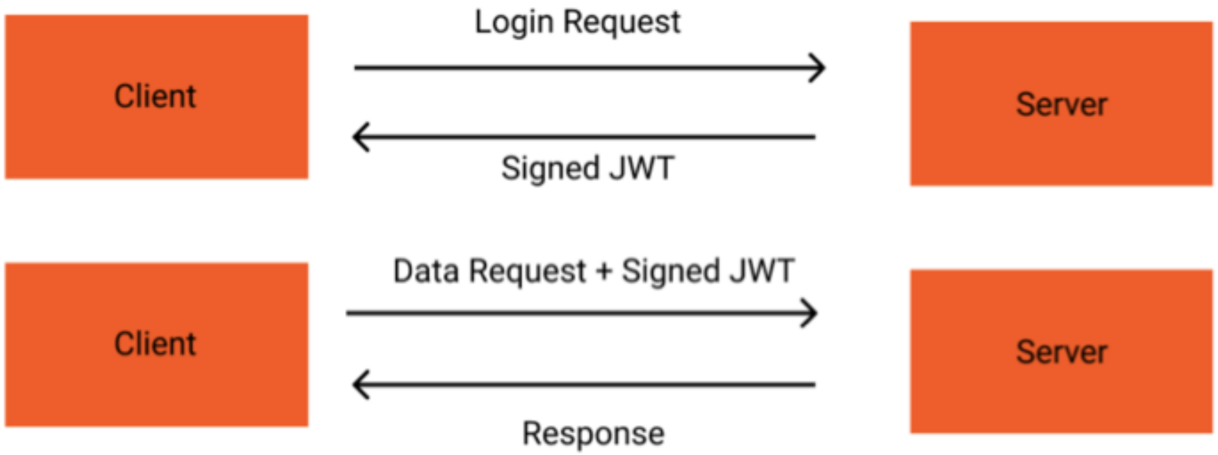
- The server actively **remembers** the still valid session IDs and the associated user info
- Such information cannot be maliciously altered since it **never leaves the server**
- Each time a request arrives for a restricted resource, the server **retrieves** the info associated with the session and decides if the user is allowed or not
- Works best with a single server that manages everything



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

Stateless server

- The **server signs a payload** which contains information about user info, what can be accessed, and when the authorization expires
- The server **sends** the signed payload to the client, that **stores** it
- Each time a request arrives for a restricted resource, the receiving **server verifies the signature**, extracts and uses the information
- Works best where there are multiple servers which cannot easily share session information, often employed for REST API servers in single page applications (SPA)



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

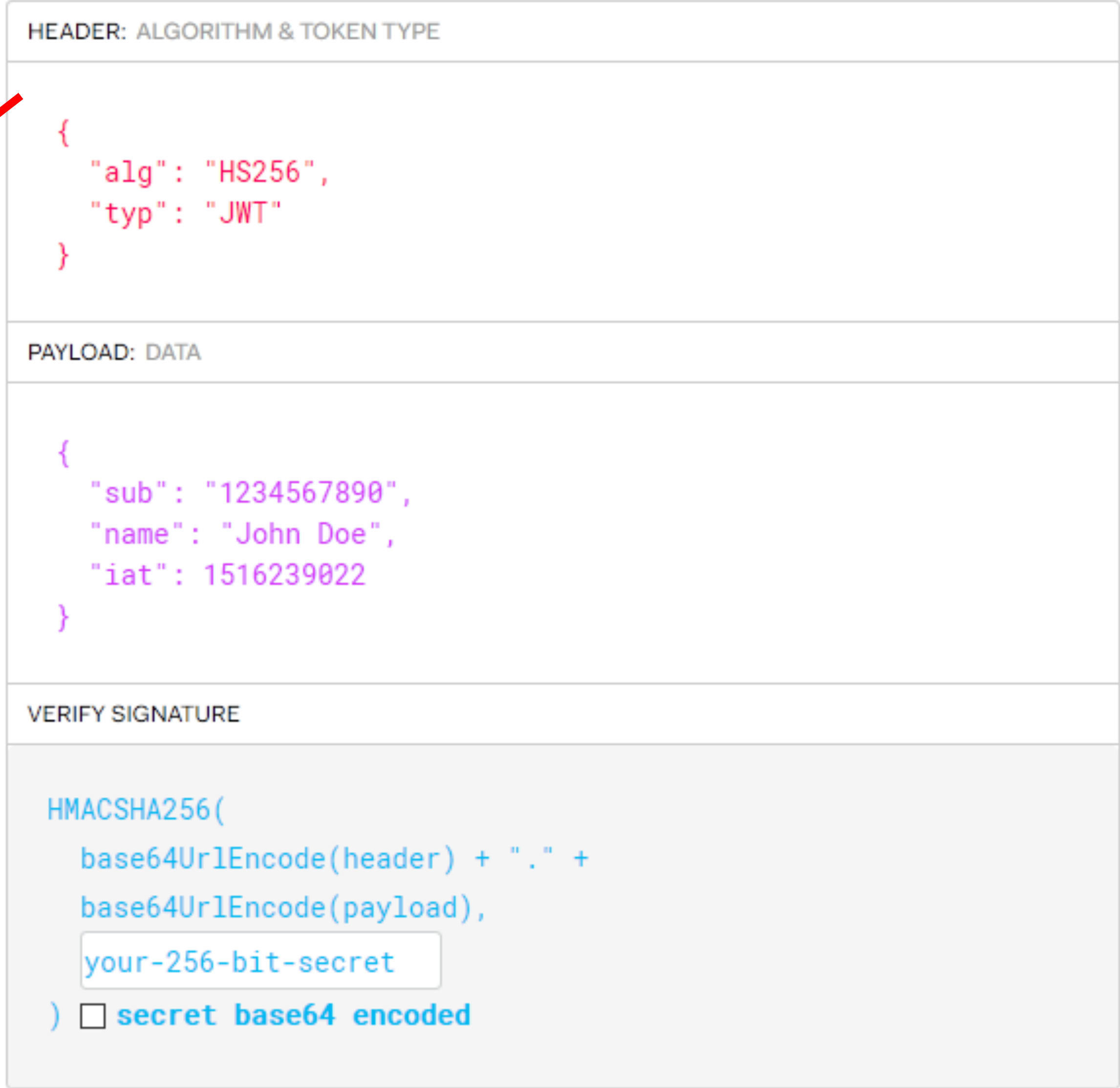
JSON Web Token: a mechanism for authorization

- 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, but the most common use case is to use the payload to define a user session
- JWTs used for authentication should contain at least:
  - a user ID
  - an expiration timestamp

https://blog.angular-university.io/angular-jwt-authentication/
https://tools.ietf.org/html/rfc7519

JWT Example

```
eyJhbGciOiJIUzI1NiIsInR5cCI6
IkpXVCJ9.
eyJzdWIiOiIzNTM0NTQzNTQzNTQz
NTM0NTMiLCJleHAiOjE1MDQ2OTky
NTZ9.
zG-2FvGegujxoLWwIQfNB5IT46D-
xC4e8dEDYwi6aRM
```



HEADER: ALGORITHM & TOKEN TYPE

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

PAYLOAD: DATA

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

VERIFY SIGNATURE

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

https://blog.angular-university.io/angular-jwt-authentication/

https://jwt.io/

JWT Pros and Cons

- To confirm JWT validity, **validating the signature is enough**
- No need to contact the authentication server that provided the JWT
- No need to keep the token in server memory nor server storage (files, DBs etc.) between HTTP requests

- Difficult to make JWT invalid sooner than the expiration time
  - Change secret (invalidates all tokens), list of blacklisted/whitelisted token (requires stateful server)
- For this reason, better to have a short expiration time
  - Requires generating a new JWT token while the old is still valid

JWT in practice

- As any other authentication token, it must be kept secret
  - sent over HTTPS only
- Must be sent with each request to the server (e.g., REST API server)
- The server receiving the token (e.g., REST API server) 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/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://medium.com/@ryanchenkie_40935/react-authentication-how-to-store-jwt-in-a-cookie-346519310e81

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

JWT in practice

JWT IN PRACTICE

Our recommendations

- Create a login page which collects username/password and sends a POST request to the authentication server
- Receive the signed JWT from the authentication server
  - As a **cookie, invisible to JS ("http only" option)** to protect from XSS
- The browser will always send the JWT in the HTTP cookie header to any API that requires authentication
  - Use proxy mechanism for API server: cookie cannot be sent to other domains/ports
- Protect the system from CSRF using a standard library
  - Necessary due to the use of cookie

Client login form: use standard practice

- Create it as React component, with local state, and validation if required

```
<LoginForm userLogin={this.userLogin}/>

class LoginForm extends React.Component {
    constructor(props) {
        super(props);
        this.state = { username: '', password: '' };
    }
    doLogin = (event) => {
        event.preventDefault();
        if (this.form.checkValidity()) {
            this.props.userLogin(this.state.username, this.state.password);  // Make POST request to authentication server
        } else {
            this.form.reportValidity();
        }
    }
...
```

Server

- Decide a sufficiently long *random secret*
  - Keep it secret (don’t commit to GitHub…)
- Decide where to keep the JWT (recommended: cookie)
- Develop an authentication server which receives credentials (username/password), and upon successful authentication it sends the JWT as response body
  - For instance, add `POST /login` route in API server
- Verify the JWT signature for each API call for which authorization is needed
  - Typically through a middleware that does it automatically

JWT in express.js

- Several libraries are available
- Most frequently adopted ones:
  - `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
  - `getToken()`: extract token from the request (req => req.cookies.token)
  - `credentialsRequired`: if false, allow access to unauthorized users for logging or other purposes

Protecting REST APIs

```
app.use(cookieParser());

app.use(
  jwt({
    secret: jwtSecret,
    getToken: req => req.cookies.token
  })
);

// All the following APIs will require authentication
...
//REST APIs
//app.post('/api/exam', ...
...
```

Unauthorized requests

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

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

```
app.use(function (err, req, res, next) {
  if (err.name === 'UnauthorizedError') {
    res.status(401).json(authErrorObj);
  }
});
```

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

```
const jwt = require('express-jwt');
const jsonwebtoken = require('jsonwebtoken');
const cookieParser = require('cookie-parser');

const jwtSecret =
'6xvL4xkAAbG49hcXf5GIYSvkDICiUAR6EdR5dLdwW7hMzUjjMUe9t6M5kSAYxsvX';
```

Login route

```
const expireTime = 1800; //seconds

app.post('/api/login', (req, res) => {
    dao.checkUserPwd(req.body.username, req.body.password)
    .then((userID) => {
      const token = jsonwebtoken.sign({ user: userID }, jwtSecret, {expiresIn: expireTime});
      res.cookie('token', token, { httpOnly: true, sameSite: true, maxAge: 1000*expireTime });
      res.end()
    }).catch(
      // Delay response when wrong user/pass is sent to avoid fast guessing attempts
      () => new Promise((resolve) => { setTimeout(resolve, 1000) }).then(
           () => res.status(401).end()
      )
    );
});
```

Password storage and check

- **Never** save plaintext passwords anywhere (e.g., DB)
- Save the result of a strong hash function applied to the plaintext password
  - So that even if hashed passwords are leaked, they cannot be used directly
  - Note: passwords cannot be recovered, but users can set a new password if needed
- When checking authentication, compare password hashes
- Use well-established libraries, for instance `bcrypt`
  - Provides methods to create and check hashes
  - `bcrypt.hash(myPlaintextPassword, saltRounds, function(err, hash) { /* store hash */ }`
  - `bcrypt.compare(myPlaintextPassword, hash, function(err, result) { /* result == true? */ }`

https://www.npmjs.com/package/bcrypt

Password hash check

```
exports.checkUserPass = function (user, pass) {
    return new Promise((resolve, reject) => {
        const sql = 'SELECT username, passwordhash, name FROM user WHERE username = ?';
        db.all(sql, [user], (err, rows) => {
            if (err) { reject(err); return; }
            if (rows.length === 0) { reject(null); return; }

            // bcrypt.compare might be computationally heavy
            // thus the compare method provides a callback function which is called when computation ends
            bcrypt.compare(pass, rows[0].passwordhash, function (err, res) {
                if (err) { reject(err); }
                if (!res) { reject(null); }
                else { resolve({
                        userID: rows[0].username,
                        name: rows[0].name,
                    }); return;
                }
            });
        });
    });
}
```

Accessing the JWT payload

- For authenticated requests, the JWT payload is accessible via `req.user`

```
// ... const token = jsonwebtoken.sign({ userID: id}, jwtSecret, {expiresIn: expireTime});
// ...

// GET /exams
app.get('/api/exams', (req, res) => {
  // Extract userID from JWT payload
  // check if req.user is present, in case the API is used without authentication
  const userID = req.user && req.user.userID;
  dao.listExams(userID)   // e.g., userID is a parameter of the DB query
    .then((exams) => res.json(exams))
    .catch((err) => res.status(503).json(dbErrorObj));
});
```

Logout route

- To logout, simply delete the cookie containing the JWT from the browser
- Need to be done via server SetCookie since it is not directly accessible from the client JS code
  - NB: This does not make the token expire before its deadline, if it is stolen it can still be used until its expiration timestamp

```
app.post('/api/logout', (req, res) => {
  res.clearCookie('token').end();
});
```

CSRF protection for the cookie case

- Express.js does not contain support by default
- Libraries available
- The most frequently adopted one is `csurf`

- `npm install csurf`

CSRF: server side

```
const csrf = require('csurf');
...
const csrfProtection = csrf({
  cookie: true
});
...
app.get('/api/csrf-token', csrfProtection, (req, res) => {
  res.json({ csrfToken: req.csrfToken() });
});
...
// Any non-GET API to be protected with middleware call
app.put('/api/exams/:code', csrfProtection, ...)
```

CSRF: client side

```
// In main App, upon successful authentication:
API.getCSRFToken().then( (response) => this.setState({csrfToken:
response.csrfToken}));


// In API.js
async function getCSRFToken() {
    return new Promise((resolve, reject) => {
        fetch(BASEURL + '/csrf-token').then((response) => {
            if (response.ok) {
                response.json()
                    .then((obj) => { resolve(obj); })

...
```

CSRF: client side: send additional header

```
// In main App, when needed, call API functions
API.updateExam(exam, this.state.csrfToken).then( ...

// In APIs
async function updateExam(exam, csrfToken) {
    return new Promise((resolve, reject) => {
        fetch(BASEURL + '/exams/' + exam.coursecode, {
            method: 'PUT',
            headers: {
                'Content-Type': 'application/json',
                'X-CSRF-Token': csrfToken,
            },
            body: JSON.stringify(exam),
        }).then(
 ...
```

Tip: authentication is a complex problem

- Much more than “simple” JWT + Cookies + CSRF
- Third party authentication (google, facebook, etc.)
  - More complex, but similar principles
  - Oauth2, …
- Many possible attack approaches (XSS, CSRF, …)

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

- **Consult a security expert** before deployment in real world applications!

License

- These slides are distributed under a Creative Commons license "**Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)**"
- **You are free to:**
  - **Share** — copy and redistribute the material in any medium or format
  - **Adapt** — remix, transform, and build upon the material
  - The licensor cannot revoke these freedoms as long as you follow the license terms.
- **Under the following terms:**
  - **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
  - **NonCommercial** — You may not use the material for commercial purposes.
  - **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
  - **No additional restrictions** — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.
- https://creativecommons.org/licenses/by-nc-sa/4.0/