

<WA1/>

<AW1/>

2024

# Authentication

**For some, but not for all**

Fulvio Corno

Luigi De Russis

Enrico Masala



# Outline

- The need for authentication
- HTTP sessions
- Authentication in React and in Express



<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

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 (login with Google, Facebook, ...)
  - ...
- Involve both client and server
  - and requires to understand several new concepts
- Better if you rely upon
  - best practices and “standardized” processes
  - **advice by security experts!**

# Layers of Authorization

Who	What	How	When
User	Login / Logout / Navigate pages		
React App	Is the user logged? Remember user information	State/Context variables	Set at login Destroyed at logout Queried during navigation
Browser	Remembers navigation session	Session Cookie (stores session ID)	Received at login, in HTTP Response Re-sent to server at every HTTP Request
Server	Remember session data	Session storage (creates session ID, remembers associated data: username, group, level, ...)	Created at login Destroyed at logout Retrieved at every HTTP Request
Route (HTTP API)	Check authorization Execute API	Verify session validity	At every (non-public) HTTP Request
Route (Login)	Perform authentication	Check user/pass If ok, create session information	At Login time
Route (Logout)	Forget authentication	Destroy session information	At Logout request
Database (at Login)	Validates user information	Queries & password encryption	At Login time
Database (HTTP API)	Retrieves user information	Queries from session information	At every HTTP Request

Giving memory to HTTP

# COOKIES AND SESSIONS

# Sessions

- **HTTP is stateless**
  - each request is independent and must be self-contained
- A web application may need to keep some information between different interactions
- For example:
  - in an on-line shop, we put a book in a shopping cart
  - we do not want our book to disappear when we go to another page to buy something else!
  - we want our “state” to be remembered while we navigate through the website



# Sessions

- A **session** is temporary and interactive data interchanged between two or more parties (e.g., devices)
- It involves one or more messages in each direction
- Often, one of the parties keeps the state of the application
- It is established at a certain point in time and ended at some later point

# Session ID

- Basic mechanism to maintain session
- Upon authentication, the client receives from the server a session ID
- The session ID allows the server to recognize subsequent HTTP requests *as authenticated*
- Such an information
  - must be stored on the client side
  - must be sent by the client at every request which is part of the session
  - must not be sensitive!
- Typically stored in and sent as **cookies**

# Cookie

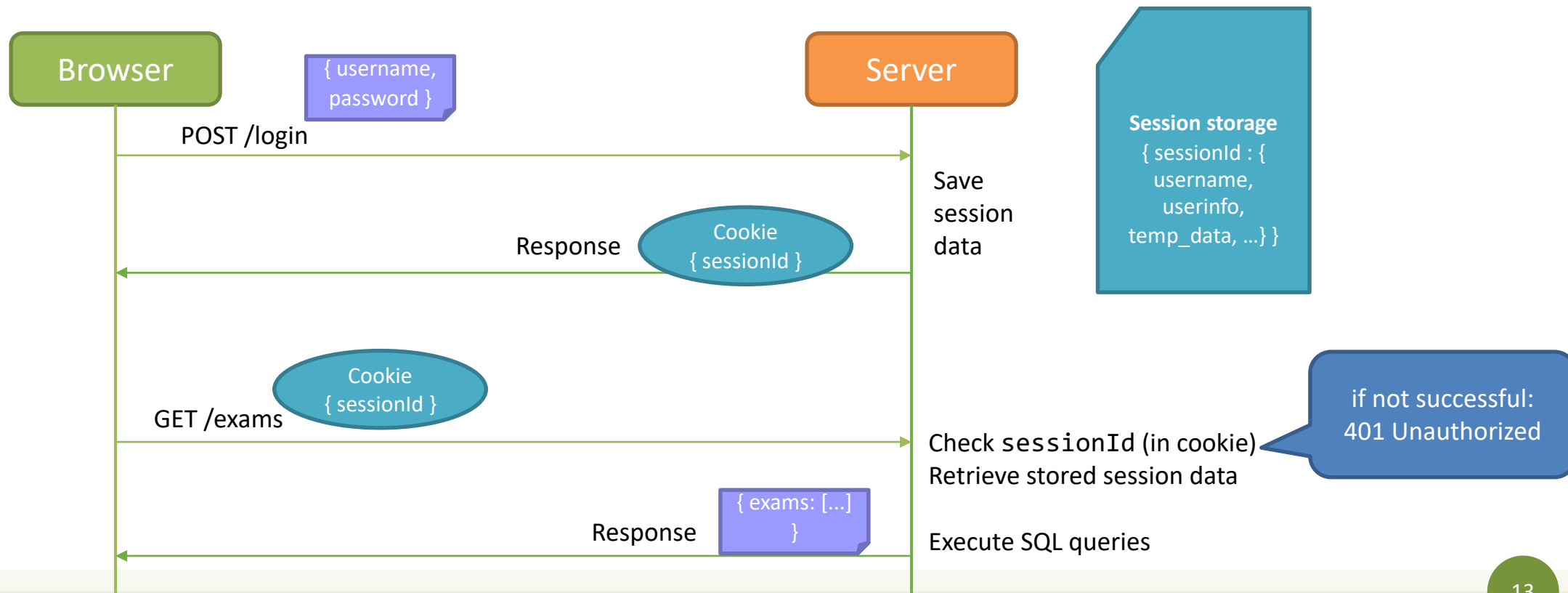
- A small portion of information stored in the browser (in its cookie storage)
- Automatically handled by browsers
- Automatically sent by the browser to servers when performing a request to the same **domain** and **path**
  - options are available to send them in other cases
- Keep in mind that sensitive information should **NEVER** be stored in a cookie!

# Cookie

- Some relevant attributes, typically set by the server:
  - **name**, the name of the cookie [mandatory]
    - Example: `SessionID`
  - **value**, the value contained in the cookie [mandatory]
    - Example: `94$KKDEC3343KCQ1!`
  - **secure**, *if set*, the cookie will be sent to the server over HTTPS, only
  - **httpOnly**, *if set*, the cookie will be **inaccessible to JavaScript** code running in the browser
  - **expiration date**

# Session-based Auth

- The user state is stored on the server
  - in a storage or, for development only, in memory



# A Note About Security...

- **Always** use HTTPS and “secure” cookies (at least in production)
  - use “httpOnly” cookies
- **Never** store sensitive information into cookies
- Rely on **best practices** and avoid to *re-invent the wheel* for auth
- Web applications can be exposed to several “basic” attacks
  - *CSRF* (Cross-Site Request Forgery), a user is tricked by an attacker into submitting a request that they did not intend
  - *XSS* (Cross-Site Scripting), attackers inject malicious JS code into web pages
  - Most of these can be prevented with a proper usage of frameworks, best practices, and dedicated libraries

Authentication and authorization with Passport.js and React

# **AUTH IN PRACTICE**

# Base Login Flow (I)

1. A user fills out a form in the client with a unique user identifier and a password
2. Data is validated and, if ok, is sent to the server, with a POST API
3. The server receives the request and checks whether the user is already registered, and the password matches
  - Password comparison exploits cryptographic hashes
4. If not, it sends back a response to the client
  - “Wrong username and/or password”



# Base Login Flow (II)

5. If username and password are correct, the server generates a session id
6. The server stores the session id (together with some user info retrieved by the database) in its “server session storage”
7. The server replies to the login HTTP request by creating and sending a cookie
  - with name = SessionID, value = the generated session id, httpOnly = true, secure = true (if over HTTPS)
8. The browser receives the response with the cookie
  - the cookie is automatically stored by the browser
  - the response is handled by the web application (e.g., to say "Welcome!")

# Login Form: Use Standard Practice

- Create it as React component with local state

```
<LoginForm userLogin={userLoginCallback}/>
```

```
function LoginForm(props) => {  
  const [username, setUsername] = useState('');  
  const [password, setPassword] = useState('');  
  
  doLogin = (event) => {  
    event.preventDefault();  
    if (... form valid ...) {  
      props.userLoginCallback(username, password); // Make POST request to authentication server  
    } else {  
      // show invalid form fields  
    }  
  }  
}
```

...

# Authentication with Passport



- We are going to use an authentication middleware to authenticate users in Express
  - **Passport**, <http://www.passportjs.org>
  - install with: `npm install passport`
- Passport is flexible and modular
  - supporting 500+ different authentication strategies
  - for instance, username/password, login with Google, login with Facebook, etc.
  - able to adapt to different types of databases (SQL and noSQL)
  - adopting some best practices *under-the-hood*
    - e.g., httpOnly cookies for sessions

# Passport: Configuration

An Express-based server app needs to be configured in three ways before using Passport for authentication:

1. Choose and set up which authentication strategy to adopt
2. Personalize (and install) additional middleware
3. Decide and configure which user info is linked with a specific session

# 1. LocalStrategy

- Strategies define how to authenticate users
- **LocalStrategy** supports authentication with username and password
  - install with: `npm i passport-local`
- `function verify (username, password, callback)`
  - Goal: to find/verify the user that possesses given credentials
- `callback()` supplies Passport with the authenticated user
  - or false and an optional message

```
import passport from 'passport';
import LocalStrategy from 'passport-local';

passport.use(new LocalStrategy( function
verify (username, password, callback) {
    dao.getUser(username,
password).then((user) => {

        if (!user)
            return callback(null, false, {
message: 'Incorrect username and/or
password.' });

        return callback(null, user);
    });
}));
```

# The Verify Function in LocalStrategy

- **username, password**: automatically extracted from `req.body.username` and `req.body.password`
- Must check the validity of the credentials
- **callback()**: communicates the result
  - **callback(null, user)** → valid credentials
  - **callback(null, false)** → invalid credentials, login failed
  - **callback(null, false, { message: 'error' })** → invalid credentials, login failed, with explanation
  - **callback({error: 'err msg'})** → application error (e.g., DB error)
- **user**: *any object* containing information about the currently validated user

```
import passport from 'passport';
import LocalStrategy from 'passport-local';

passport.use(new LocalStrategy( function verify
(username, password, callback) {
    dao.getUser(username, password).then((user) => {

        if (!user)
            return callback(null, false, { message:
'Incorrect username or password.' });

        return callback(null, user);
    });
}));
```

# Storing Passwords in the Server

- **Never** store plain text passwords in the server (e.g., in the database)
- **Always** perform hashing of the password
  - so that nobody can retrieve your password, knowing its hash
  - as hashing is a one-way function
- `scrypt` is a (secure) *password hashing* function that you can use
  - e.g., password ->  
d72c87d0f077c7766f2985dfab30e8955c373a13a1e93d315203939f542ff86e
  - test it at <https://www.browserling.com/tools/scrypt>
- In Node, it is included in the provided `crypto` module

# scrypt

- Two main functions, both async and returning Promises:

1. Hash a password:

```
crypto.scrypt(password, salt, keylen, function(err, hashedPassword))
```

The `salt` should be random and at least 16 bytes long:

```
const salt = crypto.randomBytes(16)
```

`keylen` is the length of the hash to obtain (e.g., 32 or 64).

2. Check if a given password matches with a stored hash:

```
crypto.timingSafeEqual(storedPassword, hashedPassword)
```

The given password must be hashed with the same salt of the stored password



# Password Hash Check (within Passport)

```
export const getUser = (email, password) => {
  return new Promise((resolve, reject) => {
    const sql = 'SELECT * FROM user WHERE email = ?';
    db.get(sql, [email], (err, row) => {
      if (err) { reject(err); }
      else if (row === undefined) { resolve(false); }
      else {
        const user = {id: row.id, username: row.email};

        const salt = row.salt;
        crypto.scrypt(password, salt, 32, (err, hashedPassword) => {
          if (err) reject(err);
          if(!crypto.timingSafeEqual(Buffer.from(row.password, 'hex'), hashedPassword))
            resolve(false);
          else resolve(user);
        });
      }
    });
  });
};
```

## 2. Additional Middleware

- Given Passport modularity, you may want *additional middlewares* for, e.g., enabling sessions
- Sessions are enabled through the `express-session` middleware
  - <https://www.npmjs.com/package/express-session>
  - install with: `npm i express-session`
- By default, `express-session` stores the session in *memory*
  - which is highly inefficient and **NOT** recommended in production
- It also supports different session storages, from files to DB

```
import session from 'express-session';

// enable sessions in Express
app.use(session({
  // set up here express-session
  secret: "a secret phrase of your choice",
  resave: false,
  saveUninitialized: false,
}));

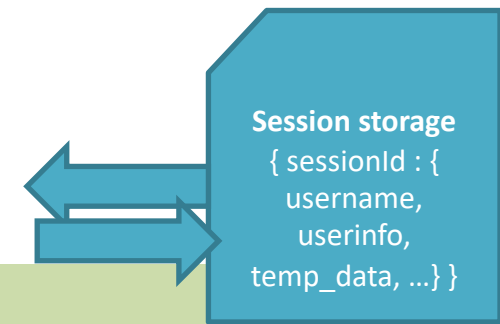
// init Passport to use sessions
app.use(passport.authenticate('session'));
```

## 2. Session Options

- The `express-session` middleware supports various parameters
- The most used ones are:
  - `secret`: used to sign the session ID cookie [**required**]
  - `store`: the session store instance, defaults to `MemoryStore` if not specified
  - `resave`: forces the session to be saved back to the session store, even if the session was never modified during the request. Default (deprecated) value is `true`, typically set to *false*
  - `saveUninitialized`: forces a session that is new but not modified to be saved to the store. Choosing *false* is useful for implementing login sessions, reducing server storage usage, or complying with laws that require permission before setting a cookie. Default (deprecated) value is `true`.

# 3. Session Personalization

- After enabling sessions, you should decide which info to put into them
  - for generating the cookie and for checking the information that arrives within it
- The `serializeUser()` and `deserializeUser()` methods allow you to define callbacks to perform these operations



```
passport.serializeUser((user, cb) => {  
  cb(null, {id: user.id, email:  
    user.username, name: user.name});  
});
```


```
passport.deserializeUser((user, cb) => {  
  return cb(null, user);  
});
```

## 3a. `serializeUser()`

- In the code, we serialize some user info to be stored in the session
  - a subset of the available user info is ok
- Passport takes that user info and stores it internally on `req.session.passport`
  - which is passport's internal mechanism to keep track of things

```
passport.serializeUser((user, cb) => {  
  cb(null, {id: user.id, email:  
user.username, name: user.name});  
});
```

```
passport.deserializeUser((user, cb) => {  
  return cb(null, user);  
});
```



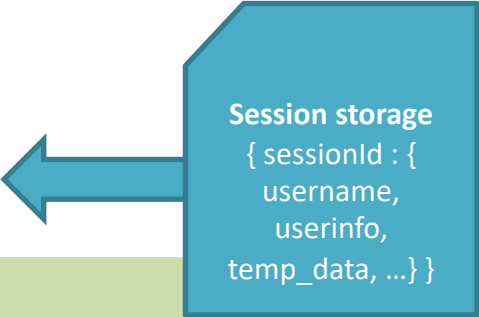
Session storage  
{ sessionId : {  
 username,  
 userinfo,  
 temp\_data, ...} }

## 3b. deserializeUser()

- The same user info that was serialized before will be restored when the session is authenticated by this function
- All the requests to the server will hit this function
- The user object created by `deserializeUser()` will be available in every authenticated request in `req.user`

```
passport.serializeUser((user, cb) => {  
  cb(null, {id: user.id, email:  
user.username, name: user.name});  
});
```

```
passport.deserializeUser((user, cb) => {  
  return cb(null, user);  
});
```



Session storage  
{ sessionId : {  
 username,  
 userinfo,  
 temp\_data, ...} }

# Login with Passport

- After setting everything up, now we can log in a user with Passport
  - adding an Express route able to receive the “login” requests
  - passing the `authenticate(<strategy>)` method as the first additional callback
    - `authenticate('local')` will look for a username and password field in `req.body`

```
app.post('/api/login', passport.authenticate('local'), (req, res) => {  
  
  // This function is called if authentication is successful.  
  // req.user contains the authenticated user.  
  res.json(req.user.username);  
  
});
```

# Storing User Information in React

- With the login response, some user information might be available in the browser
  - e.g., the username
- You might want to store such information, for later usage
- Our suggestion, to keep things simple:
  - store them in a Context (or a State)
  - ask the server for them, when needed (e.g., with `API.getUserInfo()` in a `useEffect`)
- More suggestions:
  - <https://www.robinwieruch.de/react-router-authentication/>



# After the Login...

- Some routes in the server needs to be **protected**
  - i.e., they shall provide a response for authenticated users, *only*
- The workflow shown before (session-based auth) applies
- The browser always sends the HTTP cookie header to any API that requires authentication
  - *beware*: cookie cannot be sent to other domains/ports

# With CORS Enabled

- By default, cookies can be sent to the same origin
  - CORS has mechanisms to overcome this limitation
- In the server, we need to define *both* the credentials and the origin options, when setting up the cors module:

```
const corsOptions = {  
  origin: 'http://localhost:3000',  
  credentials: true,  
};  
app.use(cors(corsOptions));
```

# With CORS Enabled

- In the client, all the fetch requests to protected APIs must include the “*credentials: include*” option:

```
const response = await fetch(SERVER_URL + '/api/exams', {  
  credentials: 'include',  
});
```

- The login request **must** include such an option as well
  - even if it is not to a protected API
  - otherwise the cookie will not be available in subsequent (protected) requests

# Protecting Routes: Basic Way

- Finally, after the session creation, we might want to *protect* some other routes
- To check if a request comes from an authenticated user, we can check Passport's **req.isAuthenticated()** at the beginning of every callback body in each route to protect
  - it returns true if the session id coming with the request is a valid one

# Protecting Routes: Advanced Way

- We can *create* an Express middleware that includes `req.isAuthenticated()`
- and use it either at the application level or at the route level
  - useful, e.g., if we want to handle errors

```
const isLoggedIn = (req, res, next) => {
  if(req.isAuthenticated())
    return next();

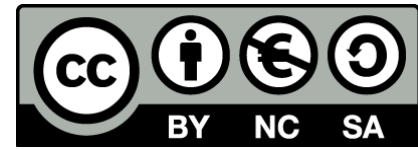
  return res.status(400).json({message : "not authenticated"});
}

app.get('/api/exams', isLoggedIn, (req, res) => {
  ...
});
```

# Logout

- The browser will send a "logout" request to the server
  - e.g., a POST /logout
- The server will clear the session (and delete the stored session id)
  - extremely trivial with Passport!

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



# 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/>

