Metodologie per la Programmazione per il Web - MF0437
Authentication

Docente

Giancarlo **Ruffo**[giancarlo.ruffo@uniupo.it]

Informazioni, materiale e risorse su:

moodle [https://www.dir.uniupo.it/course/view.php?id=16455]

Slide adattate di versioni precedenti a cura dei

Proff. Luigi De Russis ed Alessio Bottrighi

Auth*

Authentication on the Web

- Validating that users are who they claim to be
  - "demonstrate that you are who you say to be"
- Not trivial, error-prone process…
- …but standardized (sort of) and with lot of best practices
- Process at the base of "login"
  - often done with credentials: username + password
  - tons of alternatives exist!

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 specific resource
- granted authorization rights might depend on the identity established during authentication

Both are often used together to protect the access to a system

Authentication and Authorization

- Developing authentication and authorization mechanisms for the Web:
  - is complicated
  - is time-consuming
  - is prone to errors
  - may require interacting with third-party systems
  - …
- Involve both the client and the server
  - and require to understand a few new concepts
- Better if you rely upon:
  - best practices and "standardized" process
  - advice by security experts

Sessions and Cookies

Sessions

- HTTP is **stateless**
  - each request is independent and must be self-contained
- A web application may need to keep some information between pages and between different interactions
- For instance:
  - in an on-line shop, we put bananas in a shopping cart
  - we do not want our bananas to disappear when we go to another page to buy apples!
  - 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 involve one or more messages in each direction
- Often, one of the parties keeps the state of the the application
- It is established at a certain point it time and ended at some later point

Session ID

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

Cookie (rfc 6265)

- A small portion of information stored in the browser
- 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 the other cases
- Keep in mind that sensitive information should **NEVER** be stored in a cookie!

Cookie

- Some interesting attributes, typically set by the server:
  - **name**, the name of the cookie (mandatory)
  - **value**, the value contained in the cookie (mandatory)
  - **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**
- When creating cookies, set those attributes whenever is possible!

source: https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies

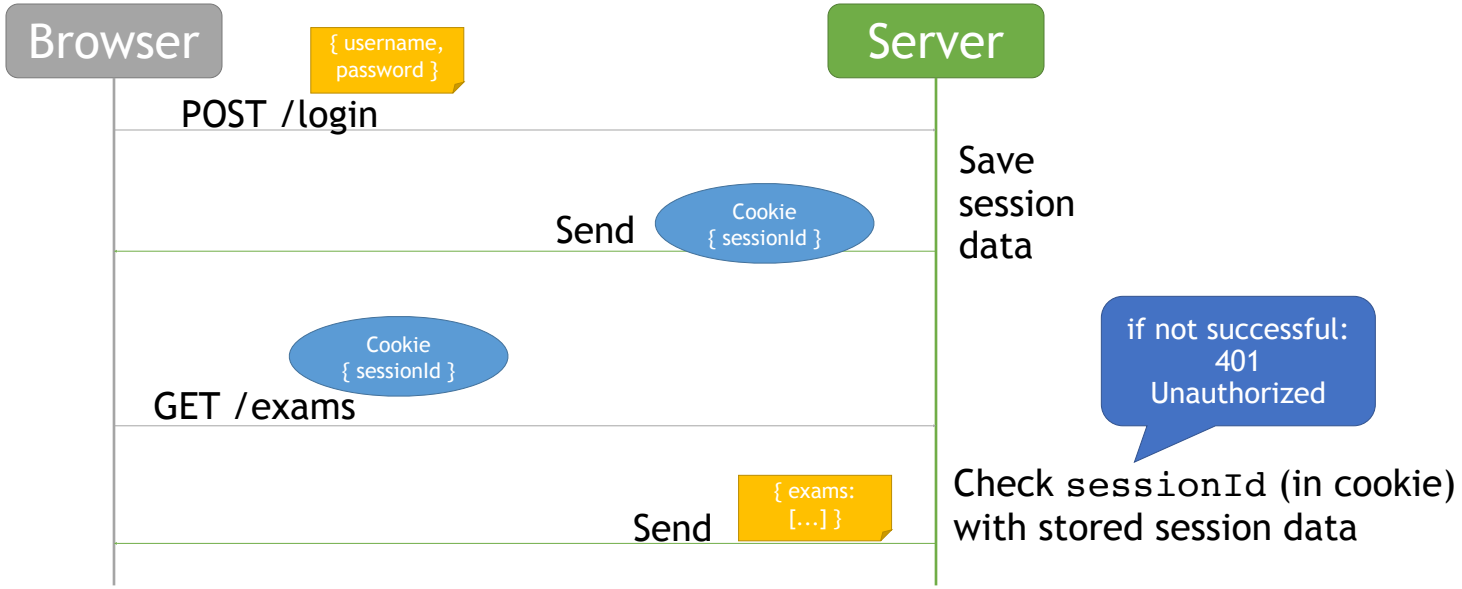
Session-based Auth vs. Token-based Auth

- Session-based Auth: the user state is stored **on the server** (statefull) **- cookie**

- Token-based Auth: the user state is stored **on the client** (stateless) **- token**

Session-based Auth

- The user state is stored **on the server**
  - in some 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
  - even if they are "httpOnly"
- 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 (e.g., https://github.com/expressjs/csurf)

Auth in Practice

Base Registration Flow

1. A user fills out a form with some info
   - including a password and a field unique for the system (e.g., email, username, etc.)
2. The form is sent to the server, with a POST request, upon HTML5 validation
3. The server receives the request and validates the content
4. The server checks whether the user is already registered
5. If yes, it sends back a response ("***A user with the same email is already registered***")
6. If no, insert the information in the database and provide a response

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
- `bcrypt` is a common (and still secure) *password hashing* function that you can use
  - e.g., password -> $2a$12$tk6/gCY.hUDTKIAgNYZgeOen1P0VbQ4mrjqbkIgvDzqktUuTsai7y
  - test it at https://www.browserling.com/tools/bcrypt
- In Express, you can use the `bcrypt` module:
  - https://www.npmjs.com/package/bcrypt

Base Login Flow (I)

1. A user fills out a form with a field unique for the system and a password
2. The form is sent to the server, with a POST request, upon HTML5 validation
3. The server receives the request and checks whether the user is already registered
4. If no, it sends back a response ("*Wrong username/email*")
5. If yes, check if the received password corresponds with the hash memorized in the database for the same user
   - If no, it sends back a response ("*Wrong password*")

Base Login Flow (II)

6. If both the username and the password are correct, the server generates a session id
7. The server stores the session id (together with some user info)
8. The server replies to the login HTTP request by creating and sending a cookie
   - with value = session id, httpOnly = true, secure = true (if over HTTPS)
9. 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!")

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

- What about the ***logout***?
  - The browser will send a "logout" request to the server
  - The server will clear the session (and delete the stored session id)

Login and Sessions 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
- `function(username, password, done)` is the **verify callback**
  - its goal is to find the user that possesses given credentials
- `done()` supply the middleware with the authenticated user (or `false`)

```
const passport = require('passport');
const LocalStrategy = require('passport-
local').Strategy;

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

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

    if (!user.checkPassword(password)) {
      return done(null, false, { message: 'Incorrect
password.' });
    }

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

2. Additional Middleware

- Given Passport modularity, you may want ***additional middleware*** for, e.g., enabling sessions
- Sessions can be enabled through the `express-session` middleware
  - https://www.npmjs.com/package/express-session
- By default, `express-session` stores the session in ***memory***
  - which is highly inefficient and not recommended in production
  - it supports, however, different session storages, from files to DB

```
const session = require('express-session');

app.use(express.static('public'));

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

app.use(passport.initialize());

app.use(passport.session());
```

3. Session Personalization

- After enabling sessions, you should decide which user info to put into the session
- Both for generating the cookie and for checking the information that arrives within the cookie
- The `serializeUser()` and `deserializeUser()` methods allow you to define callbacks to perform these operations
- The `user` object created by `deserializeUser()` will be available in every authenticated request in `req.user`

```
passport.serializeUser(function(user, done) {
  done(null, user.id);
});


passport.deserializeUser(function(id, done) {
  dao.getUserById(id).then((user) => {
    done(null, user);
  })
  .catch((err) => {
    done(err, user);
  });
});
```

Login with Passport

- Logging in a user with Passport:
  - adding an Express route able to receive the "login" requests
  - passing the `authenticate(<strategy>)` method as an additional callback (first)

```
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);

});
```

Protecting Routes

- 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 from the request is a valid one

Protecting Routes: Alternative Way

- Alternatively, we can ***create*** an Express middleware
- and using it either at the application level or at the route level

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

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

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