Web Application Development
(LTAT.05.004)

NODE.JS – III - JWT

MOHAMAD GHARIB
UNIVERSITY OF TARTU
In **MVC** web applications, the user will typically **request a resource** from the **server**, (Controller), which may cause the **controller** to **request** application **data** from the database (**Model**). Then, pass the **data/resource** to the Client (**View**), which will finally format the data for the end user.
The View- Recap

Three views are required:

1. **All posts**: fetches and presents all posts. Pressing on any post, will direct us to the **single post view with the id of the post** as a route parameter.

2. **A single post**: fetches and presents a single post based on the passed id. This view also contain **update** and **delete**, when pressed the post will be updated/deleted and we will be directed to the **All posts view**.

3. **A create post**: allows creating a new post, and contain an **add post button**, when pressed the post will be added and we will be directed to the **All posts view**.

Note: we assume that the server is running at localhost:3000/
Basic system security

Authentication & Authorization
Info. System Security: a Short History

1960s - Physical security: security was limited to the access points, where computers were stored.

1970s - Evolution of personal computer and information flow through telephone lines\(^1\): first hackers appeared, who tries different ways to intercept such information to steal the data.

1980s - Evolution of cyber-crime: hacking and other forms of cyber crimes skyrocketed as hackers find various methods to break into the computer systems benefiting from the absent of strict regulation against hacking/cyber-crimes.

1990s - Hacking becoming a serious concern: As the worldwide web becomes available, people started putting their personal information online; hackers saw this as a potential revenue source. Although firewalls/antivirus programs exists, yet the web was a mostly unsecured.

2000s - Cybercrime becoming a serious issue: hacking was not considered as serious issues but with evolution of hacking methods and the severity of their consequences, governments started taking strong measures against cyber criminals.

2010s - now: Information security nowadays: different protection measures/methods are actively developed but “hackers” are also developing new more sophisticated attack methods to breach such protection measures/methods.

\(^1\) Only large organizations and governments were starting to link computers via telephone lines
What is Security for our applications?
What is Security for our applications?

Who is asking for the recourse/service?

Is it allowed to “use” the recourse/service?
What is Security for our applications?

Who is asking for the recourse/service?

Authentication

Is it allowed to “use” the recourse/service?

Authorization
Authentication

Authentication is the process of verifying that
"you are who you say you are"

Authentication factors:

Knowledge based authentication (something you know): e.g., a password, partial password, personal identification number (PIN), security question, etc.

Ownership/possession-based authentication (something you have): e.g., ID card, security token, implanted device, cell phone with a built-in hardware token, software token, etc.

Inherence-based authentication (something you are): e.g., fingerprint, retinal pattern, signature, face, voice, unique bio-electric signals, other biometric identifiers, etc.
Authentication

Authentication is the process of verifying that

"you are who you say you are"

Authentication can be:

**Single-factor authentication** is the weakest level of authentication, only a single component from one of the three categories of factors is used to authenticate an individual’s identity.

**Multi-factor authentication** involves two or more authentication factors. Clearly, two-factor authentication is a special case of multi-factor authentication.

**Note.** Research suggests that for an effective authentication, at least two, and preferably all three, factors should be verified.
Authorization is the process of verifying that "you are permitted to do what you are trying to do".

Authorization is the function/process of specifying access rights/privileges to resources.

Authorization must always follow authentication: users should first prove that their identities are genuine before the system grant them access to the requested resources.
Basic system security

An example
Basic system security: an example

We consider a simple system that offer all of its services to registered users only, i.e., an authenticated user has full authorities.
A user can **register** by providing a **unique email** and a **password**, which can be used to **authenticate** the user.
Basic system security: an example

A user can **register** by providing a **unique email** and a **password**, which can be used to **authenticate** the user.

Email: Admin@ut.ee
Password: Admin

Email: Admin@ut.ee
Password: Admin
Basic system security: an example

Hashing algorithms allows taking data of any size and apply a mathematical process to it to produce a unique output of string. Regardless of the input length, a hash output is always of the same length.
Basic system security: an example

Once transformed by a hashing algorithm, it’s nearly impossible to revert the data to its original state – So what!

Unsalted passwords are subject to many attacks (e.g., dictionary attack).
Cookies
Cookies

A cookie is a small bit of information that a website stores within your browser.

Cookies themselves are not harmful. However, some cyber attacks can hijack cookies and enable access to your browsing sessions.

The danger lies in their ability to track individuals' browsing histories.
Cookies

A cookie is created when you visit a website.

A **cookie** can be “session” or (non-persistent **cookie**), or persistent **cookie**.
A cookie is created when you visit a website.

A **cookie** can be “session” or (non-persistent **cookie**), or persistent **cookie** .

After its creation, it is sent to be stored in the user browser.
Cookies

A cookie is created when you visit a website.

A **cookie** can be “session” or (non-persistent **cookie**), or persistent **cookie**.

After its creation, it is sent to be stored in the user browser.

When you revisit the website, your browser sends the information stored in the cookie **back** to the site.
JSON Web Token (JWT)
JSON Web Token (JWT), pronounced "jot", is an open industry standard (RFC 7519) for representing claims “securely” between two parties.

JWT defines a compact and self-contained way for “securely” transmitting information between parties as a JSON object.

Note: JWT is only a one way for authenticating users.
JWT is self-contained.

• The auth data is stored in the token not on the server.
• The client can read the data, but cannot change it (httpOnly: true).
• The server "signs" the data by hashing it with a secret.
• The server can verify the token.

JWT is (somehow) secure way to exchange authentication information.

JWT can be subject to some security attacks (e.g., Cross Site Scripting (XSS)).
I need to know who is this user!

In our system, I need to have the user’s information in my system, to authenticate him/her.
How JWT is created, signed and verified

A user can be authenticated by checking the information he/she provide with the information stored in the system. If we have a match (the user is authenticated), the system will create the JWT.
How JWT is created, signed and verified

After creating the JWT, it is sent to be stored in the user’s browser.
After **creating the JWT**, it is sent to be **stored in the user’s browser**.

When the user **revisit** the website, the browser **automatically sends the JWT** (and any cookies) to the **server**.

Result is OK proceed to resource
Result is not OK, redirect to login/signup page
JWT structure
JWT structure

Header  ·  Payload  ·  Signature
JWT structure

- **Header:** consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA.

- **Payload:** contains the claims, which are statements about the user and additional data.

- **Signature:** makes the token secure by taking the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.
JWT structure

Header
- consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA.

Payload
- contains the claims, which are statements about the user and additional data.

Signature
- makes the token secure by taking the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.

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

Example Payload:
```
{ "sub": 1234567890, "name": "John Doe", "admin": true }
```

Example Signature:
```
HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)
```
JWT structure

**Header**
- consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA.

**Payload**
- contains the claims, which are statements about the user and additional data.

**Signature**
- makes the token secure by taking the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.

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

```javascript
{ 
  "sub": 1234567890,
  "name": "John Doe",
  "admin": true 
}
```

encoded by Base64Url

HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)

dre2dS45eeRG9IiwiwaWF0iJoNTE2Mj
JWT structure

The signature is used to verify that the message was not changed, and, in the case of tokens signed with a private key, it can also verify that the sender of the JWT is who it says it is.

Header consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA.

Payload contains the claims, which are statements about the user and additional data.

Signature makes the token secure by taking the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.

```javascript
HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)
```

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```
Building a “secure” App - JWT
Our **App** includes:

A **front-end** *(Vue.js App)* offers:

- A home page *(protected)*, only authenticated users can reach.
- Login and signup pages.
- A logout button

A **backend** *(Node.js App)* allows:

1. Verify authenticated users;
2. Register new users;
3. Login registered new users; and
4. Sign-out logged in users.

A **database** *(Postgres)* contains the user’s data *(id, email, password)*.
Building a “secure” App - JWT

If a user tries to reach the homepage, there is a **navigation guard** that will check whether a user is authenticated.

**How?**
Building a “secure” App - JWT

If a user tries to reach the homepage, there is a “navigation guard” that will check whether a user is authenticated.

How?

Send a GET request to the server for verifying the jwt, if it exist.

The server will respond by sending a JSON object that contains true or false concerning the authentication of the user.

If the user is authenticated (there is a valid jwt), she will reach the homepage. Otherwise, she will be redirected to the login page.
Building a “secure” App - JWT

If the user is already registered, they can use the login page to send her credentials through a POST request to the server.
Building a “secure” App - JWT

If the user is already registered, they can use the login page to send her credentials through a POST request to the server.

If the credentials are correct, she will be logged in and a jwt token will be created and send back to her browser.

This will allow the user to pass the authentication check offered by the “navigation guard”, and reach the protected homepage.
If the user is not already **signed up**, she will not be able to **login in**, but she can navigate to the **signup page** from the **login page**.
Building a “secure” App - JWT

If the user is not already **signed up**, she will not be able to **login in**, but she can navigate to the **signup page** from the **login page**.

The user can **signup** by sending its credentials through the **signup page** as a **POST request** to the server.

This will create a **jwt** token and send back to the user browser, which will allow her to pass the authentication check offered by the **“navigation guard”**, and reach the **protected homepage**.
Building a “secure” App - JWT

A logged in user, can log out by pressing on the logout button.
Building a “secure” App - JWT

A logged in user, can log out by pressing on the logout button.

This will send a GET request to the server, which will remove the jwt token and redirect the user to the login page.
Building a “secure” App - JWT

Client

User registration (signup)

POST
localhost:3000/auth/signup
{email, password}

JWT

Server

{email, password}

jwt.sign({id, email, password})

Database

User does not exist
Building a “secure” App - JWT

User registration (signup)

POST localhost:3000/auth/signup
{email, password}

JWT

Server

User login

POST localhost:3000/auth/login
{email, password}

JWT

Database

{email, password}

User does not exist

{id, email, password}

jwt.sign({id}, secret)

User exists and password is correct

{id, email, password}

jwt.sign({id}, secret)
Building a “secure” App - JWT

User registration (signup)
- POST localhost:3000/auth/signup
  {email, password}

User login
- POST localhost:3000/auth/login
  {email, password}

Authenticate user
- GET localhost:3000/auth/verify
  cookies: jwt
  {"authenticated": authenticated}
Building a “secure” App - JWT

- **User registration (signup)**: POST request to `localhost:3000/auth/signup {email, password}` generates a JWT.
- **User login**: POST request to `localhost:3000/auth/login {email, password}` generates another JWT.
- **Authenticate user**: GET request to `localhost:3000/auth/verify` with cookies containing JWT.
- **User signout**: GET request to `localhost:3000/auth/logout` clears the JWT cookie.

The process involves checking credentials against a database and verifying the JWT with a server-side function `jwt.verify(...)`. If the user exists and their password matches, a new JWT is generated. If the user does not exist, a different response is returned.
Building a “secure” App - Database
```
// database.js

CREATE TABLE IF NOT EXISTS "users" (  
    id uuid PRIMARY KEY DEFAULT gen_random_uuid(),  
    email VARCHAR(200) NOT NULL UNIQUE,  
    password VARCHAR(200) NOT NULL
);

/*
  gen_random_uuid() A system function to generate a random Universally Unique IDentifier (UUID)
  An example of generated uuid: 32165102-4866-4d2d-b90c-7a2fddbb6bc8
*/
```
Building a “secure” App - Backend
server.js – Backend (Node.js)

// server.js
const express = require('express');
const pool = require('./database');
const cors = require('cors');
const bcrypt = require('bcrypt');
const cookieParser = require('cookie-parser');
const jwt = require('jsonwebtoken');

const port = process.env.PORT || 3000;
const app = express();
app.use(cors({ origin: 'http://localhost:8080', credentials: true }));
// We need to include "credentials: true" to allow cookies to be represented
// Also "credentials: 'include'" need to be added in Fetch API in the Vue.js App

app.use(express.json()); // Parses incoming requests with JSON payloads and is based on body-parser.
app.use(cookieParser()); // Parse Cookie header and populate req.cookies with an object keyed by the cookie names.
app.post('/auth/signup', async(req, res) => {
  try {
    const { email, password } = req.body;
    const salt = await bcrypt.genSalt();
    const bcryptPassword = await bcrypt.hash(password, salt)
    const authUser = await pool.query(
      "INSERT INTO users(email, password) values ($1, $2) RETURNING*",
      [email, bcryptPassword]);
    const token = await generateJWT(authUser.rows[0].id);
    res
      .status(201)
      .cookie('jwt', token, { maxAge: 6000000, httpOnly: true })
      .json({ user_id: authUser.rows[0].id })
      .send;
  } catch (err) {res.status(400).send(err.message);}
});
app.post('/auth/signup', async (req, res) => {
  try {
    const { email, password } = req.body;
    const salt = await bcrypt.genSalt();
    const bcryptPassword = await bcrypt.hash(password, salt);
    const authUser = await pool.query(`
      "INSERT INTO users(email, password) values ($1, $2) RETURNING*",
      [email, bcryptPassword]
    `);
    const token = await generateJWT(authUser.rows[0].id);
    res.status(201)
      .cookie('jwt', token, {
        maxAge: 6000000,
        httpOnly: true
      })
      .json({
        user_id: authUser.rows[0].id
      })
      .send;
  } catch (err) {
    res.status(400).send(err.message);
  }
});

//server.js
//generateJWT function

const secret = "gdgdhdbcb770785rgdzqws"; // use a stronger secret
const maxAge = 60 * 60;
// unlike cookies, the expiresIn in jwt token is calculated by seconds not milliseconds
const generateJWT = (id) => {
  return jwt.sign({ id }, secret, { expiresIn: maxAge })
}
app.post('/auth/login', async(req, res) => {
  try {
    const { email, password } = req.body;
    const user = await pool.query("SELECT * FROM users WHERE email = $1", [email]);
    if (user.rows.length === 0)
      return res.status(401).json({ error: "User is not registered" });
    const validPassword = await bcrypt.compare(password, user.rows[0].password);
    if (!validPassword)
      return res.status(401).json({ error: "Incorrect password" });
    const token = await generateJWT(user.rows[0].id);
    res
      .status(201)
      .cookie('jwt', token, {maxAge: 6000000, httpOnly: true })
      .json({ user_id: user.rows[0].id })
      .send;
  } catch (error) {
    res
      .status(401)
      .json({ error: error.message });
  }
});
Authenticate – Backend (Node.js)

```javascript
app.get('/auth/authenticate', async(req, res) => {
    const token = req.cookies.jwt;
    let authenticated = false;
    try {
        if (token) { // checks if the token exists
            // jwt.verify(token, secretOrPublicKey, [options, callback])
            await jwt.verify(token, secret, (err) => {
                if (err) { // not verified, redirect to login page
                    console.log(err.message);
                    res.send({"authenticated": authenticated }); // false
                } else { // token exists and it is verified
                    authenticated = true;
                    res.send({"authenticated": authenticated }); // true
                }
            });
        } else {
            res.send({"authenticated": authenticated }); // false
        }
    } catch (err) {
        res.status(400).send(err.message);
    }
});
```
Logout – Backend (Node.js)

```javascript
app.get('/auth/logout', (req, res) => {
    console.log('delete jwt request arrived');
    res
        .status(202)
        .clearCookie('jwt')
        .send('cookie cleared')
});
```
Building a “secure” App - Frontend
import default {
    user: { authenticated: false },
    authenticated: function() {
        fetch("http://localhost:3000/auth/authenticate", {
            credentials: 'include' // Don't forget to specify this if you need cookies
        }).then((response) => response.json()).then((data) => {
            this.user.authenticated = data.authenticated;
        }).catch((e) => {
            console.log("error logout");
        });
        return this.user.authenticated;
    }
}
import auth from "../auth";

const routes = [{
  path: "/",
  name: "home",
  component: HomeView,
  beforeEnter: async (to, from, next) => {
    let authResult = await auth.authenticated();
    if (!authResult) {
      next("/login")
    } else {
      next();
    }
  }
},

// ../router/index.js
// login.vue <template>

<template>
    <div class="form">
        <h3>LogIn</h3>
        <label for="email">Email</label>
        <input type="email" name="email" required v-model="email">
        <label for="password">Password</label>
        <input type="password" name="password" required v-model="password">
        <div class="container">
            <button @click="LogIn" class="center">LogIn</button>
            <button @click='this.$router.push("/signup")' class="center">Signup</button>
        </div>
    </div>
</template>
// login.vue <script>

LogIn() {
    var data = {email: this.email, password: this.password};
    fetch("http://localhost:3000/auth/login", {
        method: "POST",
        headers: {"Content-Type": "application/json"},
        credentials: 'include', // Don't forget to specify this if you need cookies
        body: JSON.stringify(data))
    .then((response) => response.json())
    .then(() => {
        console.log(data);
        location.assign("/");})
    .catch((e) => {
        console.log("error");});
}
// SignUp.vue <template>
<template>
  <div class="form">
    <h3>SignUp</h3>
    <label for="email">Email</label>
    <input type="email" name="email" required v-model="email">
    <label for="password">Password</label>
    <input type="password" name="password" required v-model="password">
    <button @click="SignUp" class="SignUp">SignUp</button>
  </div>
</template>
// SignUp.vue <script>

SignUp() {
    var data = {email: this.email, password: this.password};
    fetch("http://localhost:3000/auth/signup", {
        method: "POST",
        headers: {"Content-Type": "application/json"},
        credentials: 'include', // Don't forget to specify this if you need cookies
        body: JSON.stringify(data))
    .then((response) => response.json())
    .then(() => {
        location.assign("/");
    })
    .catch((e) => {
        console.log("error");
    });
}
// HomeView.vue <script>
<template>
  ...
  <button v-if = "authResult" @click="Logout" class="center">Logout</button>
  ...
</template>

<script>
  ...
  Logout() {
    fetch("http://localhost:3000/auth/logout", {
      credentials: 'include'}
    .then((response) => response.json())
    .then((data) => {
      console.log('jwt removed');
      location.assign("/"); // why redirect to the home directory?
    })
    .catch((e) => {
      console.log("error logout");})
  }
</script>
Node.Js - III

Try it yourself ...
Where are these tokens/cookies!

In the devTool, navigate to the Application tab, then choose “Cookies”, under which you should find the JWT.

Go to https://courses.cs.ut.ee/2022/WAD/fall, and check how many cookies are put in your browser 😊
Node.Js - III

- Details on how you can set up a Node.js App to deal with cookies/jwt installation are provided on the course Wiki.

- The Frontend App (Vue.js) is available at this Repo: https://github.com/M-Gharib/WAD2022-Lab13-Frontend.git

- The back-end App (Node.js) is available at this Repo: https://github.com/M-Gharib/WAD2022-Lab13-Backend.git
Useful Tool

REST Client allows you to send HTTP request and view the response in Visual Studio Code directly.

REST Client: https://marketplace.visualstudio.com/items?itemName=humao.rest-client
Extra resources

Node.js Tutorial: https://www.w3schools.com/nodejs/

Introduction to Node.js: https://nodejs.dev/en/learn/

Node.js in Action - 1st Edition by Mike Cantelon, Marc Harter, TJ Holowaychuk, Nathan Rajlich
Thank You for your attention

Mohamad Gharib
mohamad.gharib@ut.ee

unitartu
tartuuniversity