SWE 4538: SERVER PROGRAMMING LAB

LAB_03: Session Based Authentication

PREPARED BY :

SHOHEL AHMED || ASSISTANT PROFESSOR
a.shohel@iut-dhaka.edu

FARZANA TABASSUM || LECTURER
farzana@iut-dhaka.edu

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
ISLAMIC UNIVERSITY OF TECHNOLOGY

OCTOBER 8, 2024

Contents

1 Session Based Authentication

In session-based authentication, the server is responsible for creating and maintaining the user's authentication state and providing a way to reference that record in each subsequent request.
The process begins when the user authenticates and provides some credentials. If the credentials are valid, the server creates a persistent record representing that session, containing information such as a random string, user identifier, session start time, session expiration time, and so on. This record is stored in the database and returned to the user's browser as a cookie.
Each subsequent calls sends an HTTP request from the browser containing the session cookie. The server can use this cookie to look up the session record, verify its validity, and make authorization decisions based on the user's identity.

1.1 Passport js

Passport is authentication middleware for Node.js. For further details, visit https://www.passportjs.org/

2 Implementation

At first, we need to install some modules for the authentication.

```
npm install bcrypt bcryptjs cookie-parser cors express-sessio nodemon passport passport-local dotenv
```

2.1 app.js

Import some modules in app.js.

```
const express = require("express");
const app = express();
const cookieParser = require("cookie-parser");
const session = require("express-session");
const passport = require("passport");
const cors = require("cors");
```

Here-

- cookie-parser: Parses cookies attached to client requests.
- express-session: Manages user sessions, which helps to store data between requests.
- passport: Authentication middleware for handling login and sessions.
- cors: Cross-origin resource sharing (CORS) is a browser mechanism which enables controlled access to resources located outside of a given domain.

Session, and Passport Setup CORS Configuration:

```
app.use(
  session({
    secret:"secret", //A string to sign session IDs for security.
    resave: false,  // Avoids resaving a session if nothing has changed.
    saveUninitialized: false,  //Avoid saving uninitialized sessions (sessions without data).
  })
);
app.use(passport.initialize());
app.use(passport.session());
app.use(cookieParser());
```

```
app.use(cors({
  origin: 'http://localhost:3000',
  credentials: true, // Allow cookies to be sent
}));
```

- session(): Initializes sessions, with "secret", "resave", "saveUninitialized". It is used as the session itself needs to be authenticated.
- initialize() and session(): Initializes Passport authentication and integrates it with session handling.
- cors(): Configures CORS to allow cross-origin requests from http://localhost:3000, allowing cookies to be sent between the client and server.

```
app.use(routes);

app.get("/welcome", ensureAuthenticated, (req, res) => {
  res.sendFile(__dirname + "/views/homePage.html");
});
```

ensureAuthenticated: Middleware that protects the /welcome route to ensure only authenticated users can access it.

2.2 routes.js

Add basic routes for authentication

```
const express = require("express");
const router = express.Router();

router.get("/login", getLogin);
router.post("/login", postLogin);
router.get("/register", getRegister);
router.post("/register", postRegister);

module.exports = router;
```

2.3 middleware.js

This file contains middleware that ensures the user is authenticated before accessing certain routes. `req.isAuthenticated()` method checks if the user is already authenticated by the authentication function or not

```
const ensureAuthenticated = (req, res, next) => {
  if (req.isAuthenticated()) {
    next();
  } else {
    res.status(200).json({error: "You do not have access"})
  }
};

module.exports = ensureAuthenticated;
```

2.4 passport.js

This file configures Passport for local authentication.

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

`passport-local`: This is the Passport strategy for authenticating users with a username and password.
`bcrypt`: This library is used to securely hash and compare passwords.

```
function initialize(passport, getUserByEmail, getUserById) {
  const authenticateUser = async (email, password, done) => {
    const user = getUserByEmail(email)
    if (user == null) {
    return done(null, false, { message: 'No user with that email' })
    }

    try {
      if (await bcrypt.compare(password, user.password)) {
        return done(null, user)
      } else {
        console.log("email")
        return done(null, false, { message: 'Password incorrect' })
      }
    } catch (e) {
      return done(e)
    }
  }

  passport.use(new LocalStrategy({ usernameField: 'email' }, authenticateUser))
  passport.serializeUser((user, done) =>{
    done(null, user.id)}) //specify what user data should be stored in the session after a user logs in
  passport.deserializeUser((id, done) => {  // This function retrieves the user data based on the stored user identifier (e.g., user ID).
    return done(null, id)
  })
 }
```

`function initialize(passport, getUserByEmail, getUserById) ...`: initializes the authentication process. It receives three parameters:

- passport: The Passport instance used to configure the strategies.
- getUserByEmail: A function to retrieve a user based on their email (used in authentication).
- getUserById: A function to retrieve a user by their ID.

`authenticateUser` Function (Actual Authentication): It first checks whether a user with the provided email exists or not. If the user exists, then `bcrypt.compare()` compares the password and returns true/false based on that.
`serializeUser((user, done)`: Determines which data of the user object should be stored in the session. The user id (you provide as the second argument of the done function) is saved in the session and is later used to retrieve the whole object via the deserializeUser function.
`deserializeUser((id, done)`: The first argument of deserializeUser corresponds to the key of the user object that was given to the done function. So your whole object is retrieved with help of that key. That key here is the user id.

2.5 controller.js

Import modules/dependencies:

```
const path = require("path");
const bcrypt = require("bcrypt");
const passport = require("passport");
```

```
4
5     const initializePassport = require("../config/passport");
6
7     const users = []; // store the user info here
```

`users`: An array to temporarily store registered users (for simplicity). This array is used to store user objects with id, name, email, and hashed password

```
1     initializePassport(
2       passport,
3       email => users.find(user => user.email === email),
4       id => users.find(user => user.id === id)
5       );
```

This initializes Passport with the local authentication strategy. It passes two functions:
`email => users.find(user => user.email === email)`: This function looks up a user in the `users` array by email.
`id => users.find(user => user.id === id)`: This function finds a user by their ID for session management.
GET requests to view files-

```
1     const getLogin = async (req, res) => {
2       const filePath = path.join(__dirname, "..", "views", "login.html");
3       res.sendFile(filePath);
4     };
5
6     const getRegister = async (req, res) => {
7       const filePath = path.join(__dirname, "..", "views", "register.html
    ");
8       res.sendFile(filePath);
9     };
```

`postRegister` Function:

```
1    const postRegister = async (req, res, next) => {
2   try {
3     const hashedPassword = await bcrypt.hash(req.body.password, 10); //
     req.body.password ==> password should be exact match to register.html
     name=password,  10:how many time you want to generate hash. it's a
     standard default value
4     users.push({
5       id: Date.now().toString(),
6       name: req.body.username,
7       email: req.body.email,
8       password: hashedPassword,
9     });
10
11    res.redirect("/login");
12  } catch {
13    res.redirect("/register");
14  }
15  console.log(users); // show the user list
16 };
```

Password Hashing: When a user registers, their password is hashed using `bcrypt.hash()`. The second argument, 10, is the "salt rounds," which defines how many times the password is hashed. A higher number is more secure but slower.
Adding User: The user's info (ID, username, email, and hashed password) is added to the users array. *Redirect:* After successful registration, the user is redirected to the login page (`/login`). *Error Handling:* If there's an error during registration, the user is redirected back to the registration page (`/register`).
`postLogin` Function:

```
const postLogin = (req, res, next) => {

  passport.authenticate("local", {
    successRedirect: "/welcome",
    failureRedirect: "/login",
    failureFlash: true,
  })(req, res, next);
};
```

This handles the login POST request. It uses `passport.authenticate()` with the **local** strategy.