Server-side
Web Development

Unit 17. Node.js. Security and middelware

Fidel Oltra, Ricardo Sánchez

IES Jaume II El Just
Tavernes de la Valldigna
Departament d'Informàtica
Curs 2023-24

Index

1 Middleware

Middleware can be defined as the software that runs between two others. In the Node.js context, middleware is a function that handles HTTP requests and responses. It can manipulate them and pass them to the next middleware for further processing or terminate the process and independently send a response to the client.

1.1 The "use" method

There are different middlewares available in Node modules, such as the `express.json()` processor we've used in previous examples to handle the body of POST requests and access their content more conveniently. Middleware is applied to an application through its `use` method. So, when we wrote:

```
app.use(express.json());
```

…we were activating the corresponding Express middleware to process request bodies and extract JSON information from them, making it ready to be accessed.

1.2 Defining Our Own Middleware

In addition to this and other examples of third-party middleware that we can incorporate into our projects, we can also define our own middleware functions. We can include it in the request and response processing chain using the application's `use` method. For example, the following middleware logs the IP address of the client sending the request to the console:

```
app.use((req, res, next) => {
    console.log("Request from", req.ip);
    next();
});
```

As we can see, middleware is nothing more than a function that accepts three parameters: the *request*, the *response*, and a reference to the *next* middleware that must be called `next`. Any middleware can end the chain by sending something in the response to the client. If it doesn't, it calls the next middleware.

We can use multiple `use` calls to load different middlewares, and the order in which we make these calls is crucial because it determines the order in which these middlewares will be executed.

```
app.use(express.json());
app.use(function(...) { ... });
app.use(...);
```

There are some useful middlewares, such as the mentioned `express.json()`, or the `router`, which is typically the last element in the chain. It is used to configure different available request routes and their responses.

1.3 Adding Middleware to Specific Routers

When we define independent routers in separate files, we can separately add middleware to each router using the router's own `use` method. For example, we can add middleware in the `routes/contacts.js` file that logs the current date to the console:

```
// routes/contacts.js
let router = express.Router();

router.use((req, res, next) => {
    console.log(new Date().toString());
    next();
});
...
```

2 Security

When we want to secure certain routes or sections of a web application, various mechanisms can be used. In the context of *traditional* web applications (those that serve visible content in a browser, such as HTML content), the traditional authentication mechanism is session-based authentication.

However, when we want to extend the web application beyond the use of a browser and allow other types of clients to connect to the backend (e.g., desktop applications or mobile applications), session-based authentication is not the best option. In such cases, it is necessary to use more universal mechanisms like **token-based authentication**, which is the most widely used authentication type in REST services.

2.1 Fundamentals of Token-Based Authentication

As we have seen on previous units, a token by itself is a meaningless string of text. However, combined with certain keys, it becomes a mechanism to protect our applications from unauthorized access. Token-based authentication is a method by which we ensure that each request to a server is accompanied by a signed token containing the necessary data to verify that the client has been previously validated.

The mechanism used for token-based authentication is as follows:

1. The client sends its authentication data to the server (typically a login and password).
2. The server validates these credentials against some form of storage (usually a database). If they are correct, it generates a token, an encoded string that identifies the user. This token is sent to the client, typically as response data. Internally, it may contain some data that allows identifying the user, such as their login username or email.

> Note: It is not recommended to add confidential information, such as the password, in the token, as it can be easily decoded. This doesn't mean that the token is an insecure authentication mechanism, as the server uses a secret key to encrypt a part of the token, but the rest of the token is more exposed.

3. The client receives the token and stores it locally (using mechanisms like *localStorage* in JavaScript or similar ones in other languages). With each new request, the client resends this token in the request headers, allowing the server to verify that it is an authorized client.

Tokens are usually assigned a lifespan or expiration date (which can be minutes, days, weeks, etc.). With each new connection, the lifespan can be renewed (this is not automatic; we need to do it ourselves). If more than the stipulated time passes without the client attempting to reconnect, they will be asked to authenticate again.

The **JSON Web Token (JWT)** standard is commonly used, defining a compact way to transmit this information between the client and the server using JSON objects.

2.2 Tokens authentication in Node.js

In order to implement authentication in our app we need:

1. A new document (table) in our MongoDB database to store the user's information and credentials.
2. A registration method to register new users and to generate their hashed passwords.
3. An authentication method that receives the user's credentials (typically username/email and password), checks them against the database and generates an authentication token which is resent to the user.

4. An authorization method, based on the user's role. In this step we protect certain routes depending on the user's role.

The next examples will be done in our contacts application example.

2.2.1 The User model

Create a new entity in the models folder in a file named `user.js`:

```
const mongoose = require('mongoose');

let userSchema = new mongoose.Schema({
    email: {
        type: String,
        required: true,
        unique: true,
        trim: true,
        match: /^\w+@[a-zA-Z_]+?\.[a-zA-Z]{2,3}$/i
    },
    password: {
        type: String,
        required: true
    },
    role: {
        type: String,
        default: "user",
        trim: true
    }
});

let User = mongoose.model('users', userSchema);
module.exports = User;
```

Instead of use the match operator to validate the email, we can use the **validator** library. To do that, first, install the packet using npm:

```
npm install validator
```

Then, import the library and replace `match` with `validate: validator.isEmail`:

```
...
const validator = require('validator');
...
email: {
    type: String,
    required: true,
    unique: true,
    trim: true,
    validate: validator.isEmail
},
...
```

As you can see, we will identify the user by its email. Also we assign a basic role with the `'user'` value.

2.2.2 Users registration

To handle user's registration and authentication, we will do a new route controller named `users.js` in the `routes` folder:

```
// routes/users.js

const express = require('express');

let Contact = require(__dirname + '/../models/user.js');
let router = express.Router();

// Here will go the routes

module.exports = router;
```

Remember to add the `users` route in `index.js`:

```
// index.js
...
//Routes
const users = require(__dirname + '/routes/users');
...
// Routers for each group of routes
```

```
...
app.use('/users', users);
...
```

We also need the Bcrypt.js module in order to make the hash of the password. Install it with:

```
npm install bcryptjs
```

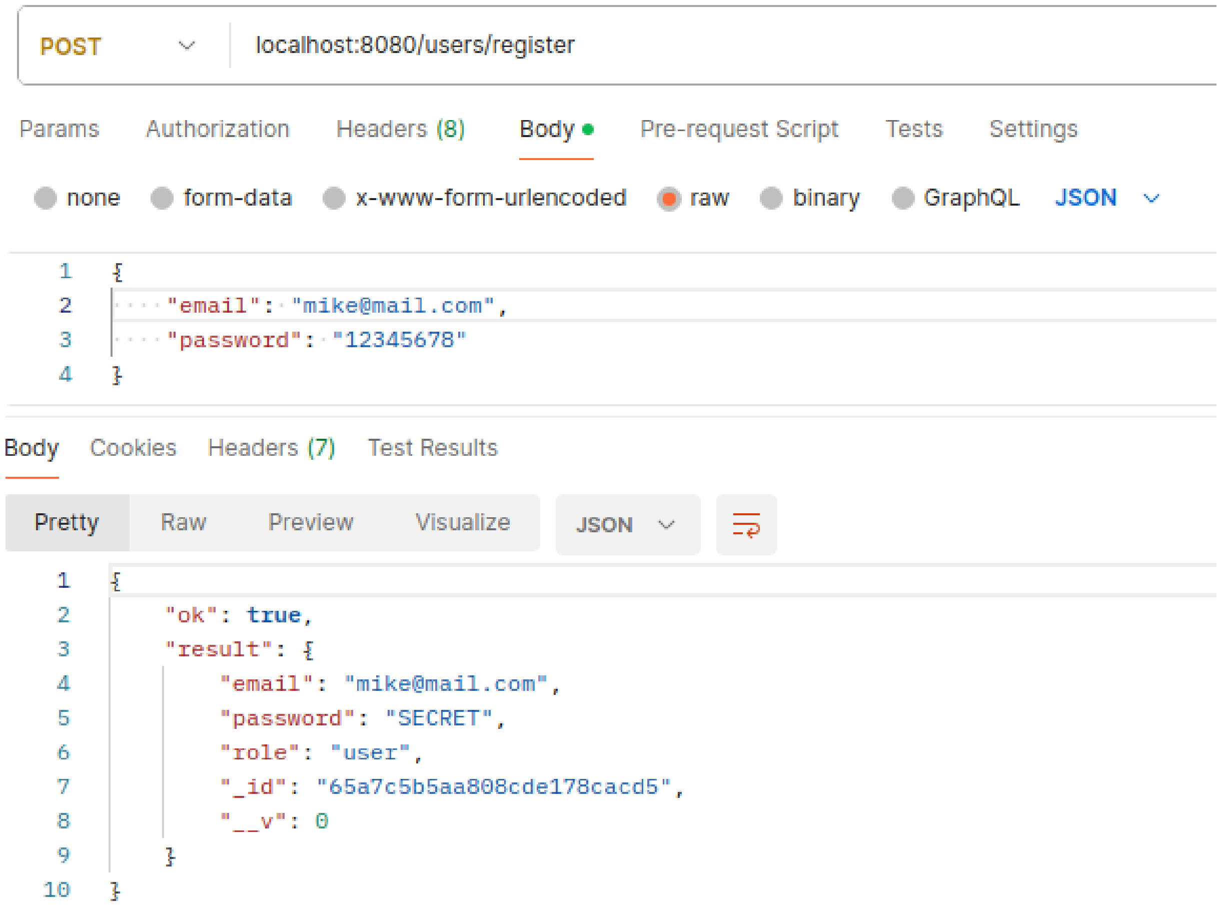
And add the module to the `users.js` file:

```
const express = require('express');
const bcrypt = require('bcryptjs');
...
```

Now, let's create the new route. We will use the POST `/register` route, so the full registration route will be `/users/register`. In the method, we check if the password has a minimum length of 8 characters (if we do that later, when the password is encrypted, the users could write a password with a length less than 8). If the password has the minimum length, we encrypt it with the `bcrypt.hash` method (the second parameter, `10`, is the number of salt rounds used). If successful, create a new user with the hashed password and store it in the database:

```
// routes/users.js

//Register a new user
router.post('/register', (req, res) => {
    //Check the password length before cipher
    if(req.body.password.length < 8)
    {
        res.status(400)
            .send({ok: false,
                error: "The password must have at least 8 characters"});
    } else {
        bcrypt.hash(req.body.password, 10, (err, hash) => {
            let newUser = new User({
                email: req.body.email,
                password: hash
            });
            newUser.save().then(result => {
                //Delete the password after saved
                newUser.password = "SECRET";

                res.status(200)
                    .send({ok: true, result: result});
            }).catch(error => {
                res.status(400)
                    .send({ok: false, error: error});
            });
      });
    }
});
```

If you send a valid email and password, you will get the result, but with the password changed to “SECRET”, for security reasons:



localhost:8080/users/register

```
{
    "email": "mike@mail.com",
    "password": "12345678"
}
```

```
{
    "ok": true,
    "result": {
        "email": "mike@mail.com",
        "password": "SECRET",
        "role": "user",
        "_id": "65a7c5b5aa808cde178cacd5",
        "__v": 0
    }
}
```

Figure 1: Users registration with Postman

You can check that the password has been stored hashed in *mongosh*:

```
contacts> db.users.find({email: "mike@mail.com"})
[
  {
    _id: ObjectId('65a7c5b5aa808cde178cacd5'),
    email: 'mike@mail.com',
    password: '$2a$10$a.0KZ4Zoz34qEpaYY/igje9LK9FmGp5Q4eiwR/Ey5ukYP3kPap4n.',
    role: 'user',
    __v: 0
  }
]
```

2.2.3 Users authentication

To generate JWT token we will need the **jsonwebtoken** library. As the library requires a secret token to sign the token, we also need the **dotenv** library in order to store it in a separate `.env` file (this file should be in your `.gitignore` file so as to maintain it secret). First, we install the modules:

```
npm install dotenv jsonwebtoken
```

And import them on the `users.js` script. We also use the method `dotenv.config` to get the config variables:

```
// routes/users.js
const jwt = require('jsonwebtoken');
const dotenv = require('dotenv');

// Get config vars
dotenv.config();
```

As a simple method to generate a secret token, you can do a separate script with the command:

```
// generateSecret.js
console.log(require('crypto').randomBytes(64).toString('hex'));
```

Execute it writing `node generateSecret.js` on a console. A 64 bit string will be shown on the console. Copy and paste it on the `.env` file, and assign it to a variable named `TOKEN_SECRET`:

```
//.env
TOKEN_SECRET=7db218ad0b4fcfb58f9e4f....
```

We will do a function to generate the tokens in `users.js`:

```
let generateToken = login => {
    return jwt.sign({login: login}, process.env.TOKEN_SECRET, {expiresIn: "1
     ↪  hours"});
};
```

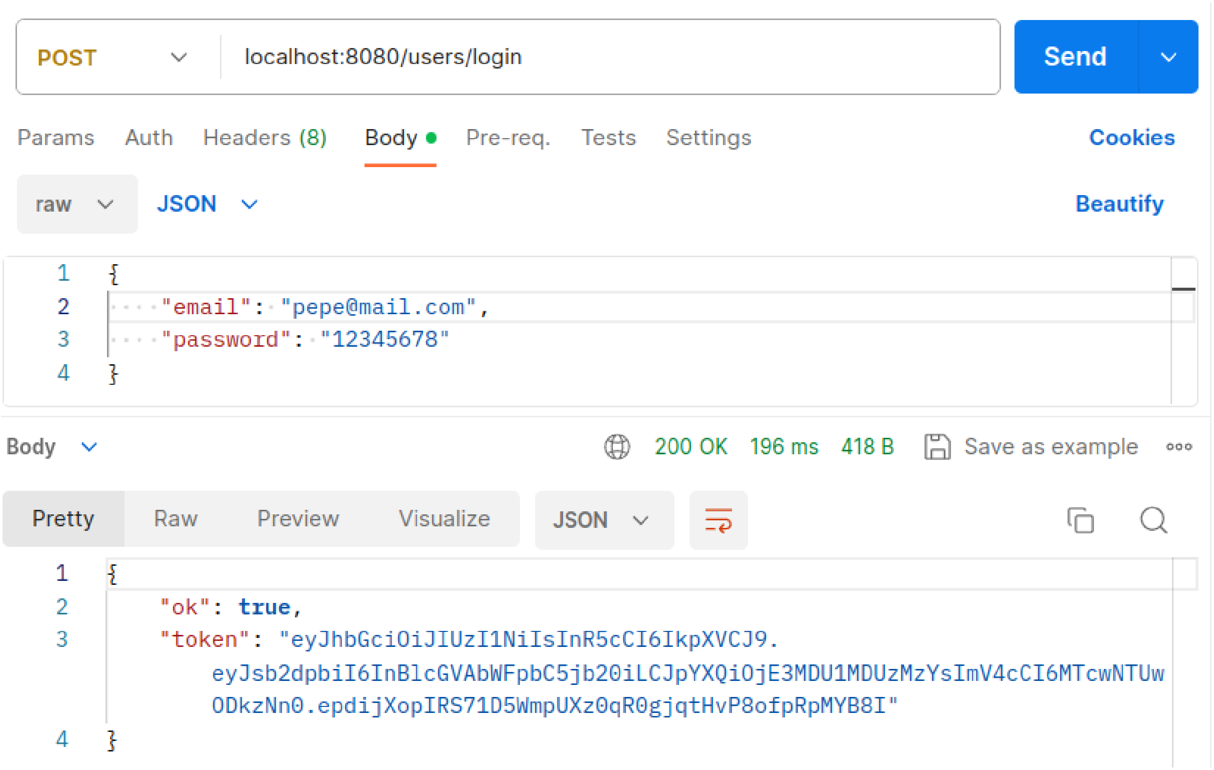
We pass the login parameter (in our case it will be the user's email), the secret token stored in `.env` and a duration of 1 hour.

Time to do the POST `/login` route. First, we get the POST parameters email and password. Then, check if the user identified by its email exists. If so, use the `bcrypt.compareSync` function to check if the password is correct and then, generate a token with the email as login field:

```
// routes/users.js

//Login
router.post('/login', (req, res) => {
    let plainPassword = req.body.password;
    let email = req.body.email;

    User.findOne({email: email})
    .then(result => {
        if(result) {
            if ( bcrypt.compareSync(plainPassword, result.password) ){
                res.status(200)
                    .send({ok: true, token: generateToken(email)});
            } else {
                res.status(400)
                    .send({ok: false, error: "Wrong password"});
            }
        } else {
            res.status(400)
                .send({ok: false, error: "User not found"});
        }

    })
    .catch(error => {
        res.status(400)
            .send({ok: false, error: "Login error: " + error});
    });
});
```

> NOTE: In a real application, it's not recommendable to use different message for "User not found" and "Wrong password", because a hacker can guess if a user is registered. Instead, show something like "User not found or wrong password" for both cases.

Now, if you enter a valid email and password you'll get a token:



POST localhost:8080/users/login

```
{
    "email": "pepe@mail.com",
    "password": "12345678"
}
```

200 OK 196 ms 418 B

```
{
    "ok": true,
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
        eyJsb2dpbiI6InBlcGVAbWFpbC5jb20iLCJpYXQiOjE3MDU1MDUzMzYsImV4cCI6MTcwNTUw
        ODkzNn0.epdijXopIRS71D5WmpUXz0qR0gjqtHvP8ofpRpMYB8I"
}
```

Figure 2: Login with token

2.3 Authorization

Once a user can register and login, it's time to apply authorization based on roles.

In our case, we can do the next authorization levels:

- **Unauthenticated users**. Can access only to the `/users/register` and `/users/login` routes.

- **Authenticated users** with the role `user`. Can access to all the GET routes.
- **Admin users** with the role `admin`. Can access to all the GET, POST, PUT and DELETE routes. It is to say, only admin users can modify the database data.

First of all, we need at least a user with the `admin` role, because the default role `user` is assigned to all the registered users. We can modify directly the database in *mongosh* and assign the role `admin` to the user you want:

```
db.users.updateOne({email: "admin@mail.com"}, {$set: {role: 'admin'} })
```

To do the authentication we are going to do a custom middleware that will take the user's role and check if it's allowed to access certain routes. Create a new script named `authorization.js` where we write a function named `protectRoute`. This function take a role as parameter and check its value:

- If `role` is **""** (empty string) the user has access.
- If `role` is different from "", retrieve the user's email from the token and search for it in the users collection. If the user's rol is `admin`, the user has access. Otherwise, check if the value of the `role` parameters matches the value of the user's stored role.

For instance, if we pass the parameter "user" to the function, it will grant access to the route for the users with the "user" role.

The `authorization.js` script and the function `protectRoute` can be as follow:

```
// authorization.js
const jwt = require('jsonwebtoken');
const dotenv = require('dotenv');

// Get config vars
dotenv.config();

let User = require(__dirname + '/models/user.js');

let protectRoute = role => {
    return (req, res, next) => {
        let token = req.headers['authorization'];
        if (token) {
            //Omit the "Bearer " part
            token = token.substring(7);
            //Verify the token
            let result = jwt.verify(token, process.env.TOKEN_SECRET);
            //Find the token's user
            User.findOne({email: result.login})
            .then(result => {
                if (result && (role === "" || result.role === "admin" ||
                 ↪  role === result.role))
                    next();
                else
                    res.status(401)
                        .send({ok: false, error: "Unauthorized user"});
            }).catch(error => {
                res.status(400)
                    .send({ok: false, error: error});
            });
        } else
            res.status(401)
                .send({ok: false, error: "Unauthorized user"});
    }
};
exports.protectRoute = protectRoute;
```

As you can see, we get the token omitting the first 7 characters of the authorization header and verify it with the `jwt.verify` function. Once the token has been verified, we extract the user and search fot it in the database, allowing access based on the rules previously explained.

Now, we can use the `protectRoute` function in our routes. First, we need to import the script:

```
// routes/contacts.js
const {protectRoute} = require(__dirname + "/../authorization");
```

And then, we can use the middleware as argument in the routes. For instance in `contacts.js`:

```
// routes/contacts.js

router.get('/', protectRoute("user"), (req, res) => {
    ...
router.get('/:id', protectRoute("user"), (req, res) => {
    ...
router.post('/', protectRoute("admin"), (req, res) => {
    ...
router.put('/:id', protectRoute("admin"), (req, res) => {
    ...
router.delete('/:id', protectRoute("admin"), (req, res) => {
    ...
```

In Postman you can check the permissions using tokens of users with different roles. Remember to add the `Authorization` header with "Bearer" and the token value in the Headers section.