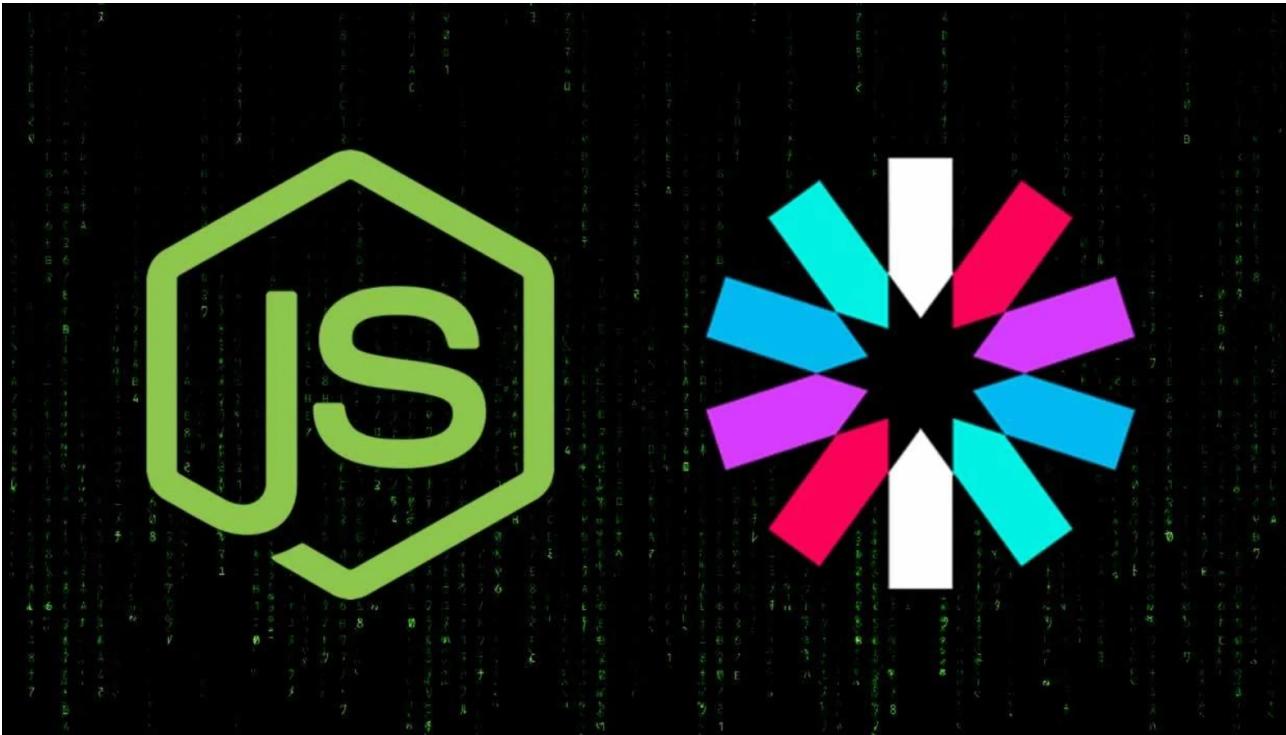


— Node.js

Ch 10: Authorization & Authentication

D. HOSTENS & M. DIMA



— Authentication <-> Authorization

- **Authentication:** Making the identity of a subject known, e.g., by entering a password.
- **Authorization:** : Granting a subject (person or process) rights to access an object (file, system).
- **Summary:**
- **authentication** = logging in, identifying the user.
- **authorization** = accessing resources based on ID.
- Bron: <https://www.kaspersky.nl/blog/identification-authentication-authorization-difference/26124/>

— Objective

- We start from our VivesBib application.
- Link: see Toledo
- Goal: only logged-in users get rights to modify data → C(R)U(D).
- Additional security: only **admin users** can delete data → (CRU)D

	CREATE	READ	UPDATE	DELETE
Visitor	✗	✓	✗	✗
Logged-in	✓	✓	✓	✗
Admin	✓	✓	✓	✓

— Authentication Steps

- Create a user model.
- Create user endpoint/route for registration.
- Create auth endpoint/route for login.
- Hash passwords.
- Return a [JSON Web Token](#).
- Save the [PrivateKey](#) in an environment variable.
- Send the [JWT](#) in headers.

— New endpoints

- Two endpoints:
- Register: POST /api/users
 - { name, email, password }(email must be unique)
- Login: POST /api/auth
 - { email, password }returns a JWT

Register

POST /api/users
{ [name, email, password]
(email must be unique)

Login

POST /api/auth
{ [email, password],
returns a JWT

Create User model models/user.js

— Joi Note

- Syntax changed from version 17 onward:

```
function validateUser(user) {  
    const schema = Joi.object({  
        name: Joi.string().min(5).max(50).required(),  
        email: Joi.string().min(5).max(50).required(),  
        name: Joi.string().min(5).max(255).required().email(),  
        password: Joi.string().min(5).max(255).required()  
    });  
    return schema.validate(user);  
}
```

- Earlier Joi versions (<17):

```
function validateUser(user) {  
    const schema = {  
        name: Joi.string().min(5).max(50).required(),  
        email: Joi.string().min(5).max(50).required(),  
        name: Joi.string().min(5).max(255).required().email(),  
        password: Joi.string().min(5).max(255).required()  
    };  
    return Joi.validate(user, schema);  
}
```

— Create Register user route - routes/users.js

```
const {User, validate} = require('../models/user');
const mongoose = require('mongoose');
const express = require('express');
const router = express.Router();
router.post('/', async (req, res) => {
    const { error } = validate(req.body);
    if (error) return res.status(400).send(error.details[0].message);

    let user = await User.findOne({ email: req.body.email });
    if (user) res.status(400).send('User already registered. ');

    user = new User({
        name: req.body.name,
        email: req.body.email,
        password: req.body.password
    });
    await user.save();
    res.send(user);
});
module.exports = router;
```

— Update index.js

- Import and use the users route and add middleware to index.js

```
const Joi = require('joi');
Joi.objectId = require('joi-objectid')(Joi);
const mongoose = require('mongoose');
const genres = require('./routes/genres');
const customers = require('./routes/customers');
const books = require('./routes/books');
const rentals = require('./routes/rentals');
const users = require('./routes/users');                                ←←
const express = require('express');
const app = express();
mongoose.connect('mongodb://localhost/vivesbib')
  .then(() => console.log('Connected to MongoDB...'))
  .catch(err => console.error('Could not connect to MongoDB...'));
app.use(express.json());
app.use('/api/genres', genres);
app.use('/api/customers', customers);
app.use('/api/books', books);
app.use('/api/rentals', rentals);
app.use('/api/users', users);                                         ←←
const port = process.env.PORT || 3000;
app.listen(port, () => console.log(`Listening on port ${port}...`));
```

— Testing with Postman

- in Postman: we create a new request:
 - Methode: POST
 - Adres: <http://localhost:3000/api/users>
 - Body > raw >JSON

```
{  
  "name": "Vives",  
  "email": "milan@vives.be",  
  "password": "12345"  
}  
• Response  
{  
  "_id": "608ab65c1c705b4242f2ee1d",  
  "name": "Vives",  
  "email": "milan@vives.be",  
  "password": "12345",  
  "__v": 0  
}
```

— Rest Client - api_calls/api_calls_users.http

```
# Users CALLS #
@base_URL=http://localhost:3000/
api/users

post {{base_URL}}
Content-Type: application/json

{
    "name": "Vives",
    "email": "student@vives.be",
    "password": "12345"
}
```

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 103
ETag: W/"67-BHrbB+kjaMpWE26rNTorDl5zwRc"
Date: Mon, 15 May 2023 06:46:37 GMT
Connection: close

{
    "name": "Vives",
    "email": "student@vives.be",
    "password": "12345",
    "_id": "6461d54d46a07a0c717e50d2",
    "__v": 0
}
```

— lodash

- We gaan het wachtwoord niet in *plaintext* terugsturen naar de *client*
- Er zijn twee opties:
- het respons-object aanpassen van `res.send(user);`
- naar:
`res.send({name: user.name, email: user.email});`
- of lodash library gebruiken (underscore ++)
`npm i lodash`
- in `users.js` bovenaan
`const _ = require('lodash');`
- `_` heeft een methode pick om properties te selecteren
`res.send(_.pick(user, ['_id', 'name', 'email']));`

— Extra voorbeeld lodash

- We passen de User-creatie ook aan met lodash

```
user = new User({  
    name: req.body.name,  
    email: req.body.email,  
    password: req.body.password  
})  
await user.save();  
res.send(user);
```

- wordt

```
user = new User(_.pick(req.body,  
['name','email','password']));  
await user.save();  
res.send(_.pick(user, ['_id', 'name', 'email']));
```

- We testen nog eens:

```
{  
    "_id": "608aba586f1781443f2326d4",  
    "name": "Vives",  
    "email": "milan3@vives.be"  
}
```

— Wachtwoorden hashen - bcrypt

- Nu worden alle wachtwoorden als plaintext opgeslagen in de DB
- We installeren het package bcrypt
- npm i bcrypt
- we testen het even uit in een nieuwe file: hash.js
- om herkenbare hashes voor courante wachtwoorden te vermijden hebben we een salt nodig.
- een salt is een random string die aan het wachtwoord toegevoegd wordt



— Salt + Hashing

```
const bcrypt = require('bcrypt');
async function run(){
    const salt = await bcrypt.genSalt(10); // number of rounds
    const hashed = await bcrypt.hash('1234',
salt);
    console.log(salt);
    console.log(hashed);
}
run();
```

- in de terminal

```
milan@les10-starter:~$ node hash.js
$2b$10$CU0KWo9xy/uI6s5xB1QeVu
$2b$10$CU0KWo9xy/uI6s5xB1QeVuuiRk3QfYk6w/
p93pdeGNLIO7H682XDNq
```

— Routehandler aanpassen

- In users.js bovenaan importeren we bcrypt
- ```
const bcrypt = require('bcrypt');
```
- Onder de creatie van de user voegen we de hashmethode toe

```
const salt = await bcrypt.genSalt(10);
user.password = await bcrypt.hash(user.password,
salt);
```

- We testen opnieuw
- In Compass
  - `_id:608abf7bb1def3470cd49ea5  
name:"Vives"  
email:"milan@vives.be"  
password:"$2b$10$JJCvJDlxQ6faQJx6BQ6RB0UHpLPc5SvIBx7rbr6M1.BzadKLCDl.C  
"  
__v:0`

## — Authenticatie

- We maken een nieuwe file in de map routes > auth.js
  - We importeren die in index.js

```
const auth = require('./routes/auth');
...
app.use('/api/auth', auth);
```

- de file auth.js op de volgende slide
- We testen in postman:
- POST request naar <http://localhost:3000/api/auth>
- Body > raw > JSON

```
{
 "email": "milan@vives.be",
 "password": "12345"
}
```

- response  
**true**

## — auth.js

```
const Joi = require('joi');
const bcrypt = require('bcrypt');
const {User} = require('../models/user');
const mongoose = require('mongoose');
const _ = require('lodash');
const express = require('express');
const router = express.Router();
router.post('/', async (req, res) => {
 const { error } = validate(req.body);
 if (error) return res.status(400).send(error.details[0].message);
 let user = await User.findOne({ email: req.body.email });
 if (!user) res.status(400).send('Invalid email or password');
 const validPassword = await bcrypt.compare(req.body.password,
 user.password);
 if (!validPassword) return res.status(400).send('Invalid email or
password');
 res.send(true);
});
function validate(req) {
 const schema = Joi.object({
 email: Joi.string().min(5).max(50).email().required(),
 password: Joi.string().min(5).max(255).required()
 });
 return schema.validate(req);
}
module.exports = router;
```

## — JSON web Tokens

- In de praktijk sturen we geen *true* terug maar een *Token*
- JSON web Token : een lange string die een user identificeert
- Als de user inlogt krijgt hij een token, die token kan hij meesturen met elke request zodanig dat we zekerheid hebben over zijn identiteit
- de JWT wordt bij webapplicaties lokaal opgeslagen
- zie [jwt.io](https://jwt.io)
- een jwt bestaat uit 3 delen:
- header: bevat het gebruikte algoritme
- payload: de data (publieke properties van de user)
- verify signature: een hash van de inhoud + private key (server)
- als de user de payload aanpast klopt de signature niet meer

## — Authenticatie tokens genereren

- We installeren JWT `npm i jsonwebtoken`
- in `auth.js`

```
const jwt = require('jsonwebtoken');
```

- en na het verifiëren van het ww

```
if(!validPassword) return res.status(400).send('Invalid email or password');
 const token = jwt.sign({_id: user._id}, 'jwtPrivateKey');
 res.send(token);
```

- we testen het in Postman:
- response:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJfaWQiOiI2MDhhYmY3YmIxZGVmMzQ3MGNkNDIiYTUiLCJpYXQiOjE2MTk3  
MDkwMTF9.LQUcf6pikgvQzQOpb1P4VHsh3A1msOsDgO\_X1XaKPGw

- We plakken het in het vak encoded van [jwt.io](https://jwt.io) en observeren de 3 velden (header, payload, signature)

# JWT Decoder JWT Encoder

Paste a JWT below that you'd like to decode, validate, and verify.

[Generate example](#)

## ENCODED VALUE

JSON WEB TOKEN (JWT)

COPY CLEAR

Valid JWT  
Signature Verified

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJfaWQi0iI2ODI2MmJmZTZiMWE0OGYwMjNhNTkyNTEiLCJpYXQiOjE3NDczMzIxNDJ9.LNUbWt071yYEp0LVJtV7e1DhHvGpg9RmK0pmcdzUM-0
```

## DECODED HEADER

JSON CLAIMS TABLE

COPY ↗

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

## DECODED PAYLOAD

JSON CLAIMS TABLE

COPY ↗

```
{
 "_id": "68262bfe6b1a48f023a59251",
 "iat": 1747332142
}
```

## JWT SIGNATURE VERIFICATION (OPTIONAL)

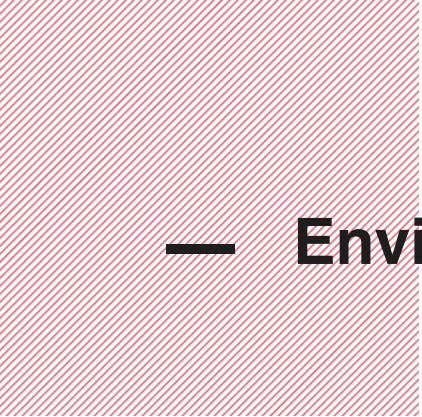
Enter the secret used to sign the JWT below:

SECRET

COPY CLEAR

Valid secret  
milan

Encoding Format



## — Environment variables

- Nu is onze JWT private Key hardcoded. Dit vormt een veiligheidsrisico. Iedereen die toegang heeft tot de broncode kan de PrivateKey kopiëren (bv GitHub)
- Good Practice is de JWT Private Key in een environment variable te stockeren. (lokaal op de computer)
- We gebruiken hiervoor opnieuw Config.
- npm i config
- we maken een nieuwe map aan in ons project met als naam config en met een default.json file

```
{
 "jwtPrivateKey": ""
}
```

## — custom-environment-variables.json

- We maken dan een extra file aan met als naam: `custom-environment-variables.json` (**naam is belangrijk!**)

```
{
 "jwtPrivateKey": "vivesbib_jwtPrivateKey"
}
```

- in auth.js importeren we config en vervangen we de hardgecodeerde 'jwtPrivateKey' door:

```
const config = require('config');
...
 const token = jwt.sign({_id: user._id},
config.get('jwtPrivateKey'));
```

- In index.js willen we de applicatie niet laten opstarten zonder dat de variable ingesteld is. We voegen de check toe onder de declaraties:

```
const config = require('config');
...
if (!config.get('jwtPrivateKey')){
 console.error('FATAL ERROR: jwtPrivateKey not defined');
 process.exit(1);
}
```

## — Environment variabele instellen

- Op Windows: set var=value (cmd), \$Env:Foo = 'value' (powershell)
- Op Mac/linux: export var=value

└ export vivesbib\_jwtPrivateKey=mySecureKey

- We testen onze applicatie opnieuw

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJfaWQiOiI2MDhhYmY3YmIxZGVmMzQ3MGNkNDIiYTUiLCJpYXQiOjE2MTk3MTEzOTJ9.6EffcYi8C4x3fMAAM6cQ8HZleBYDK23SW92KBDyGTq0

- **Opgelet!**
- De *environment variable* wordt lokaal op de computer opgeslagen. Als de applicatie op een andere computer/server gerund wordt dan moet daar de variabele ook aangemaakt worden.
- We prefixen de *environment variable* naam met de naam van de applicatie om te vermijden dat verschillende applicaties dezelfde variabelen overschrijven

## — Response headers instellen

- op dit moment sturen we het token terug naar de client in /routes/auth.js zodat ze niet telkens opnieuw moeten inloggen

```
const token = jwt.sign({_id: user._id},
config.get('jwtPrivateKey'));
res.send(token);
```

- In een real life app zou de user ook nog eerst zijn email adres moeten bevestigen maar daar maken we nu abstractie van. Onze users zijn werknemers van de bib
- We zouden het token in de /routers/users.js bij het aanmaken kunnen terugsturen

```
router.post('/', async (req, res) => {
 ...
 res.send(_.pick(user, ['_id', 'name', 'email'])); ←
```

- maar dit is geen goede aanpak

## — Betere manier: response headers

- Net zoals onze *request* heeft de *response* ook headers
- in /routes/users.js

```
const token = jwt.sign({_id: user._id}, config.get('jwtPrivateKey'));
res.header('x-auth-token', token).send(_.pick(user, ['_id', 'name',
'email']));
```

- we prefixen alle headers met x-...
- we maken gebruik van jwt en config dus we importeren ze ook bovenaan in onze file

```
const config = require('config');
const jwt = require('jsonwebtoken');
```

- we registreren daarna een nieuwe user in postman
- Post > <http://localhost:3000/api/user> > Body > Raw>JSON

```
{ "name": "Vives", "email": "milan10@vives.be", "password": "12345" }
```

Response in headers: x-auth-token: .....

## — Information expert principle

- Dubbele code in auth en users om tokens te genereren
- Onze payload bevat momenteel enkel \_id. Kans is groot dat we later extra velden toevoegen email, role, enz..
- Beter de logica op één plaats houden!
- Welke plaats? Aparte functie in auth.js? NEEN!
- OOP => **Information Expert Principle**
- d.w.z obj wie meest erover weet bevat ook de logica
- bv Koken gebeurt door kok en niet door ober
- Bij ons is het het user object dat verantwoordelijk is
- we maken een methode aan in ons user object
- eerst model aanpassen

## — models/user.js

- We extracten terug ons mongoose schema in een const

```
const User = mongoose.model('User', new mongoose.Schema({
 name: {
 ...
 }
});
```

- wordt

```
const userSchema = new mongoose.Schema({
 name: {
 ...
 }
});
const User = mongoose.model('User', userSchema);
```

- daartussen voegen we een methode toe aan ons schema

```
userSchema.methods.generateAuthToken = function() {
 const token = jwt.sign({_id: this._id},
 config.get('jwtPrivateKey'));
 return token;
}
```

- ook importeren we config en jwt bovenaan

## — routes/auth.js en models/users.js

- We vervangen de token generatie door de methode

```
const token = user.generateAuthToken();
```

- we registreren een nieuwe user in postman
- Post > <http://localhost:3000/api/user> > Body >  
Raw>JSON

```
{ "name": "Vives", "email": "milan12@vives.be", "password": "12345" }
```

- Response in headers: x-auth-token:  
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJfaWQiOiI2MDk0Mjk3MzF  
hMzc4MDMwODRIZjBhZGYiLCJpYXQiOjE2MjAzMjI2NzV9.xrA7Nw  
UK34UJAgBEY8MGq9AR8whDCpHN0J40JrsBOck

- onze nieuwe methode werkt

## — Authorization middleware

- We willen dat bepaalde endpoints enkel beschikbaar zijn voor *authenticated users*
- bv de post route in /routes/genres
- we kunnen hiervoor logica inbouwen in de route bv:

```
router.post('/', async (req, res) => {
 const token = req.header('x-auth-token');
 // bij geen token res.send(401)...
```

- als we dit doen dan moeten we dezelfde logica in alle routes implementeren. => **extracten in auth middleware**
- We hebben eerder middleware functies gezien
- we maken een nieuwe map aan middleware
- daarin een file auth.js (van autorisatie !!)

## — middleware/auth.js

```
const jwt = require ('jsonwebtoken');
const config = require ('config');
module.exports = function (req, res, next) {
 const token = req.header('x-auth-token');
 if (!token) return res.status(401).send('Access
denied. No token provided');
 try{
 const decoded = jwt.verify(token,
config.get('jwtPrivateKey'));
 req.user = decoded;
 next();
 }
 catch (ex){
 res.status(400).send('Invalid token');
 }
}
```

## — Routes afschermen

- de auth middleware toevoegen in index.js?
- Nee! Niet alle routes zijn beschermd
- we passen het specifiek toe.
- onze routes hebben 3 argumenten, de eerste is de route, tweede is optioneel de middleware en dan de route handler
- elke route die de auth middleware implementeert zal door de autorisatie middlewarefunctie gestuurd worden
- vb van implementatie:  
`router.post('/', auth, async (req, res) => {  
...  
})`

## — routes/genres.js

- We beschermen een aantal routes in genres.js
- bovenaan importeren we onze auth middleware

```
const auth = require('../middleware/auth');
```

- de post route

```
router.post('/', auth, async (req, res) => {
 const { error } = validate(req.body);
 if (error) return res.status(400).send(error.details[0].message);
 let genre = new Genre({ name: req.body.name });
 genre = await genre.save();
 res.send(genre);
});
```

- we testen het in postman > POST request naar <http://localhost:3000/api/genres/> > Send (401)
- header tab > key = x-auth-token > send (400)
- met correcte token > 200 OK (met genre JSON obj in body)



## — Testing with Rest Client

```
#POST Request met JWT in Headers

post {{base_URL}}
Content-Type: application/json
x-auth-token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJfaWQiOiI2Mjc3ZTZhOWQwZGQ2YTdmMGQ4ZDY5ZmYiLCJpYXQiOjE2NTIwMjUwMDF9.
ZxcMpnIvR7D_7870HGjGT16loGpFS-eE-TGoKIPn1XU

{
 "name": "genre2"
}
```

## — de ingelogde user ophalen

- We importeren de auth middleware

```
const auth = require('../middleware/auth');
```

- we maken een nieuwe route in /routes/users.js

```
router.get('/:id') is niet veilig!! Je kan info
van andere users ophalen
```

- onze auth mw heeft een req.body object waar de payload in zit. Momenteel enkel de user \_id
- met die user \_id kunnen we de user ophalen uit de db
- we sturen het password niet mee met de andere props

```
router.get('/me', auth, async (req, res) => {
 const user = await
 User.findById(req.user._id).select('-password');
 res.send(user);
});
```

## — Testen in Postman

- We maken een nieuwe GET request aan naar: <http://localhost:3000/api/users/me>

- Zonder token: Access denied. No token provided

- Met foutief token. In headers key toevoegen **x-auth-token** met value **1234** (400) Invalid token

- Met correct token: 200 OK

```
{
 "_id": "609429731a37803084ef0adf",
 "name": "Vives",
 "email": "milan12@vives.be",
 "__v": 0
}
```

# — Rest Client

```

Send Request
get {{base_URL}}/me
```

```
HTTP/1.1 401 Unauthorized
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 32
ETag: W/"20-xDhqsLgNrlUTL0jrgGPTBu
Date: Thu, 15 May 2025 19:25:35 GMT
Connection: close
```

Access denied. No token provided



```

Send Request
get {{base_URL}}/me
x-auth-token: wrongToken
```

```
HTTP/1.1 400 Bad Request
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 13
ETag: W/"d-esFQYRnWYNusohXXSwdo4jjdx
Date: Thu, 15 May 2025 19:56:18 GMT
Connection: close
```

Invalid token



```

Send Request
get {{base_URL}}/me
x-auth-token: eyJhbGciOiJIUzI1NiIsInR5cCIE

1 HTTP/1.1 200 OK
2 X-Powered-By: Express
3 Content-Type: application/json
4 Content-Length: 85
5 ETag: W/"55-k7Qp5nY3Kwq0YaTCt
6 Date: Thu, 15 May 2025 19:55:18 GMT
7 Connection: close
9 {
10 "_id": "68262bfe6b1a48f023a
11 "name": "Vives1",
12 "email": "docent1@vives.be"
13 "__v": 0
14 }
```



## — User uitloggen

- Het token wordt nu niet opgeslagen op de server
- Het is bad practice om tokens op te slaan in de DB
- Met een token kan je ook zonder wachtwoord bepaalde resources bereiken
- Als je toch tokens in de DB moet opslaan dan eerst encrypteren!
- Dus aangezien het token zich enkel bij de client bevindt kunnen we gewoon client-side het verwijderen om uit te loggen.
- Bijkomende security measure: Tokens enkel via HTTPS sturen (encrypted)

## — Role-based authorization

- Wat als data verwijderen enkel door admins mag?
- In ons user model hebben we nu 3 properties: name, email, password. We voegen een derde toe:  
**isAdmin: Boolean**
- We voegen de property toe handmatig aan een user in MongoDB compass: potloodje > klikken op + links van password > Add field after password: isAdmin value true, type van string naar Boolean veranderen en UPDATE
- we nemen deze property mee in onze jwt token.  
**...jwt.sign({\_id: this.\_id})...**
- wordt  
**...jwt.sign({\_id: this.\_id, isAdmin: this.isAdmin})...**

## — middleware/admin.js

- we voegen een nieuwe middleware functie toe om te controleren of de ingelogde user admin is of niet
- we maken hiervoor een nieuwe mw file aan admin.js
- onze auth.js mw maakt req.user aan dus als we admin.js daarna oproepen kunnen we dit object gebruiken

```
// 400 Bad Request //401 = unauthorised // 403
Forbidden
module.exports = function (req, res, next) {
 if (!req.user.isAdmin) return
 res.status(403).send('Acces Denied');
 next();
}
```

## — admin middleware toepassen

- in /routes/genres.js in de DELETE method

```
router.delete('/:id', [auth, admin], async (req, res) => {
 const genre = await
Genre.findByIdAndRemove(req.params.id);
 if (!genre) return res.status(404).send('The
genre with the given ID was not found.');
 res.send(genre);
});
```

- we importeren de admin mw bovenaan

```
const admin = require('../middleware/admin');
```

- we testen in Postman

## — Testen: verwijder één genre

- We zoeken een geldig genre id in compass:  
6094366aab3fa733183d8f38
- In Postman maken we een nieuwe Request aan
- DELETE, [http://localhost:3000/api/genres/  
6094366aab3fa733183d8f38](http://localhost:3000/api/genres/6094366aab3fa733183d8f38)
- Zonder token: Access denied. No token provided
- Foutief Token: (x-auth-token in headers) Invalid token
- Niet Admin Token: Acces Denied
- Admin Token: 200 OK

```
{
 "_id": "6094366aab3fa733183d8f38",
 "name": "sci-fi2",
 "__v": 0
}
```

# — Rest Client

Invalid Token

```
Preference
@objId4=68264a8c1043204358662931
Send Request
delete {{base_URL}}/{{objId4}}
Content-Type: application/json
x-auth-token: faultyToken..
```

HTTP/1.1 400 Bad Request

X-Powered-By: Express  
Content-Type: text/html; charset=UTF-8  
Content-Length: 13  
ETag: W/"d-esHnWYnXXSwdo4"  
Date: Thu, 15 May 2025 20:10:02  
Connection: close

Invalid token

Not Admin Token

```
@objId4=68264a8c1043204358662931
Send Request
delete {{base_URL}}/{{objId4}}
Content-Type: application/json
x-auth-token: eyJhbGciOiJIUzI1NiIsInR5cCI6
```

HTTP/1.1 403 Forbidden

X-Powered-By: Express  
Content-Type: text/html; charset=UTF-8  
Content-Length: 12  
ETag: W/"c-Kug4wTzb/IvSNq0pB50z...c1Pk"  
Date: Thu, 15 May 2025 20:13:48  
Connection: close

Access Denied

Admin user  
Valid Token  
Valid Object ID Genre

```
Preference
@objId4=68264a8c1043204358662931
Send Request
delete {{base_URL}}/{{objId4}}
x-auth-token: eyJhbGciOiJIUzI1NiIsInR5cCI6
```

HTTP/1.1 200 OK

X-Powered-By: Express  
Content-Type: application/json; charset=utf-8  
Content-Length: 58  
ETag: W/"3a-lrXDNUFQ1zr1mNVZCogc5NwADJA"  
Date: Thu, 15 May 2025 20:16:52 GMT  
Connection: close

```
{
 "_id": "68264a8c1043204358662931",
 "name": "genre2",
 "__v": 0}
```

