

Pile complète

Partie 4

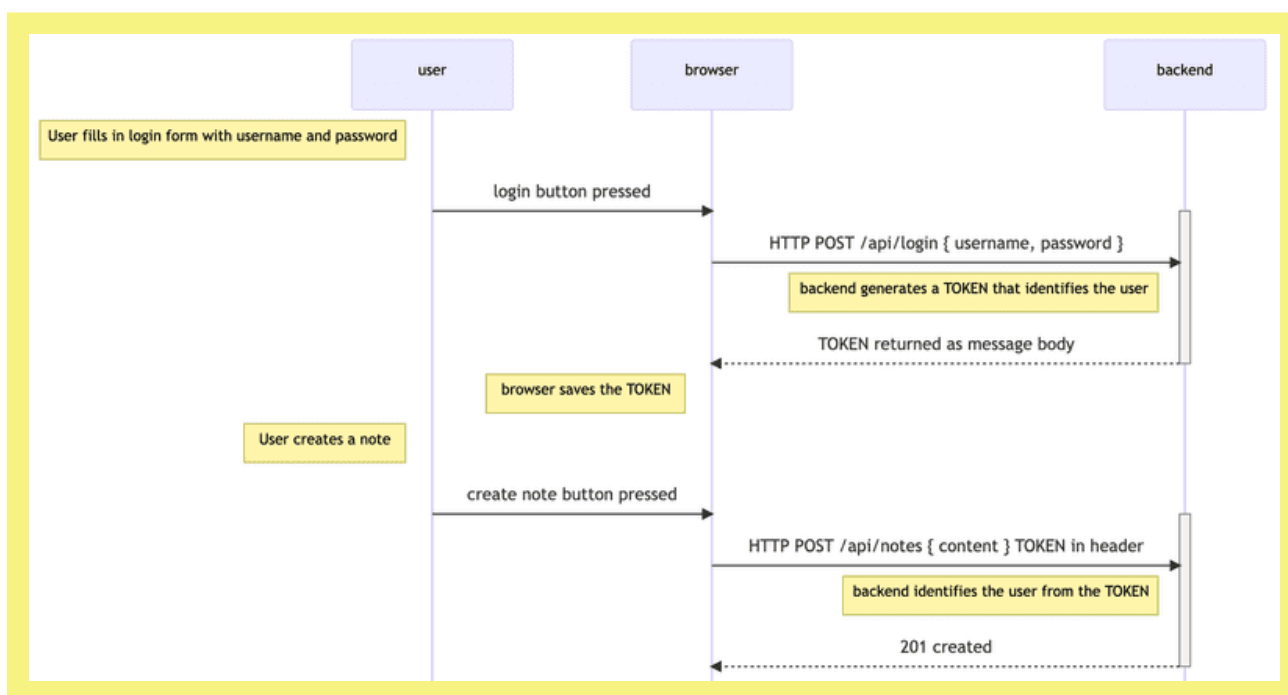
Authentification par jeton

d Authentification par jeton

Les utilisateurs doivent pouvoir se connecter à notre application, et lorsqu'un utilisateur est connecté, ses informations utilisateur doivent être automatiquement jointes à toutes les nouvelles notes qu'il crée.

Nous allons maintenant implémenter la prise en charge de l'authentification basée sur des jetons sur le backend.

Les principes de l'authentification basée sur des jetons sont représentés dans le diagramme de séquence suivant :



- L'utilisateur commence par se connecter à l'aide d'un formulaire de connexion implémenté avec React

- Nous ajouterons le formulaire de connexion au frontend dans la partie 5
- Cela amène le code React à envoyer le nom d'utilisateur et le mot de passe à l'adresse du serveur `/api/login` sous la forme d'une requête HTTP POST.
- Si le nom d'utilisateur et le mot de passe sont corrects, le serveur génère un *jeton* qui identifie d'une manière ou d'une autre l'utilisateur connecté.
 - Le jeton est signé numériquement, ce qui le rend impossible à falsifier (par des moyens cryptographiques)
- Le backend répond avec un code d'état indiquant que l'opération a réussi et renvoie le jeton avec la réponse.
- Le navigateur enregistre le jeton, par exemple dans l'état d'une application React.
- Lorsque l'utilisateur crée une nouvelle note (ou effectue une autre opération nécessitant une identification), le code React envoie le jeton au serveur avec la requête.
- Le serveur utilise le jeton pour identifier l'utilisateur

Commençons d'abord par implémenter la fonctionnalité de connexion. Installez la bibliothèque jsonwebtoken, qui nous permet de générer des jetons Web JSON.

```
npm install jsonwebtoken
```

[copie](#)

Le code pour la fonctionnalité de connexion va dans le fichier `controllers/login.js`.

```
const jwt = require('jsonwebtoken')
const bcrypt = require('bcrypt')
const loginRouter = require('express').Router()
const User = require('../models/user')

loginRouter.post('/', async (request, response) => {
  const { username, password } = request.body

  const user = await User.findOne({ username })
  const passwordCorrect = user === null
    ? false
    : await bcrypt.compare(password, user.passwordHash)

  if (!(user && passwordCorrect)) {
    return response.status(401).json({
      error: 'invalid username or password'
    })
  }

  const userForToken = {
    username: user.username,
    id: user._id,
  }

  const token = jwt.sign(userForToken, process.env.SECRET)
```

[copie](#)

```
response
  .status(200)
  .send({ token, username: user.username, name: user.name })
})

module.exports = loginRouter
```

Le code commence par rechercher l'utilisateur dans la base de données par le *nom d'utilisateur* attaché à la demande.

```
const user = await User.findOne({ username })
```

[copie](#)

Ensuite, il vérifie le *mot de passe*, également joint à la demande.

```
const passwordCorrect = user === null
  ? false
  : await bcrypt.compare(password, user.passwordHash)
```

[copie](#)

Étant donné que les mots de passe eux-mêmes ne sont pas enregistrés dans la base de données, mais *que les hachages* sont calculés à partir des mots de passe, la méthode `bcrypt.compare` est utilisée pour vérifier si le mot de passe est correct :

```
await bcrypt.compare(password, user.passwordHash)
```

[copie](#)

Si l'utilisateur n'est pas trouvé ou si le mot de passe est incorrect, la requête reçoit le code d'état 401 non autorisé. La raison de l'échec est expliquée dans le corps de la réponse.

```
if (!(user && passwordCorrect)) {
  return response.status(401).json({
    error: 'invalid username or password'
  })
}
```

[copie](#)

Si le mot de passe est correct, un jeton est créé avec la méthode `jwt.sign`. Le jeton contient le nom d'utilisateur et l'identifiant de l'utilisateur sous une forme signée numériquement.

```
const userForToken = {
  username: user.username,
  id: user._id,
}

const token = jwt.sign(userForToken, process.env.SECRET)
```

[copie](#)

Le jeton a été signé numériquement à l'aide d'une chaîne de la variable d'environnement *SECRET* comme *secret*. La signature numérique garantit que seules les parties qui connaissent le secret peuvent générer un jeton valide. La valeur de la variable d'environnement doit être définie dans le fichier *.env*.

Une demande réussie reçoit une réponse avec le code d'état *200 OK*. Le jeton généré et le nom d'utilisateur de l'utilisateur sont renvoyés dans le corps de la réponse.

```
response
  .status(200)
  .send({ token, username: user.username, name: user.name })
```

[copie](#)

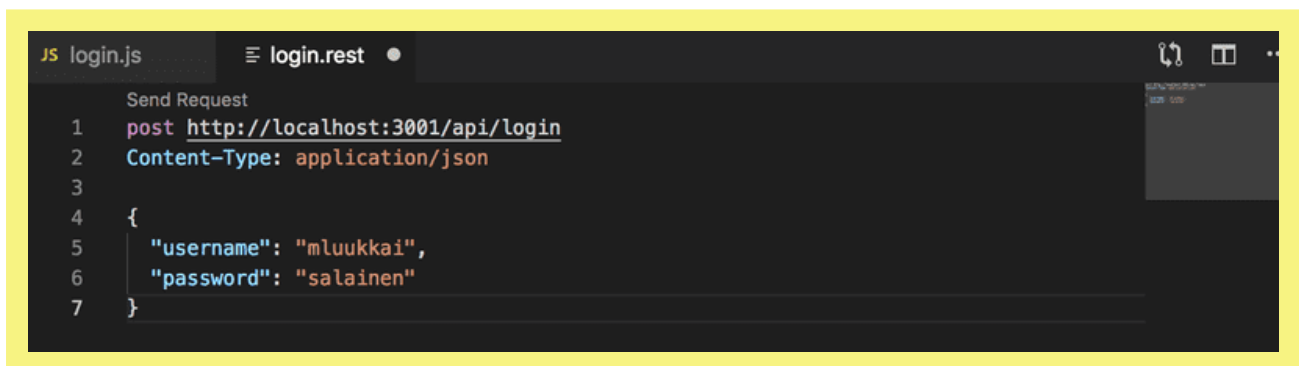
Il ne reste plus qu'à ajouter le code de connexion à l'application en ajoutant le nouveau routeur à *app.js*.

```
const loginRouter = require('./controllers/login')
fs
...

app.use('/api/login', loginRouter)
```

[copie](#)

Essayons de nous connecter à l'aide du client REST de VS Code :



Cela ne fonctionne pas. Le message suivant s'affiche sur la console :

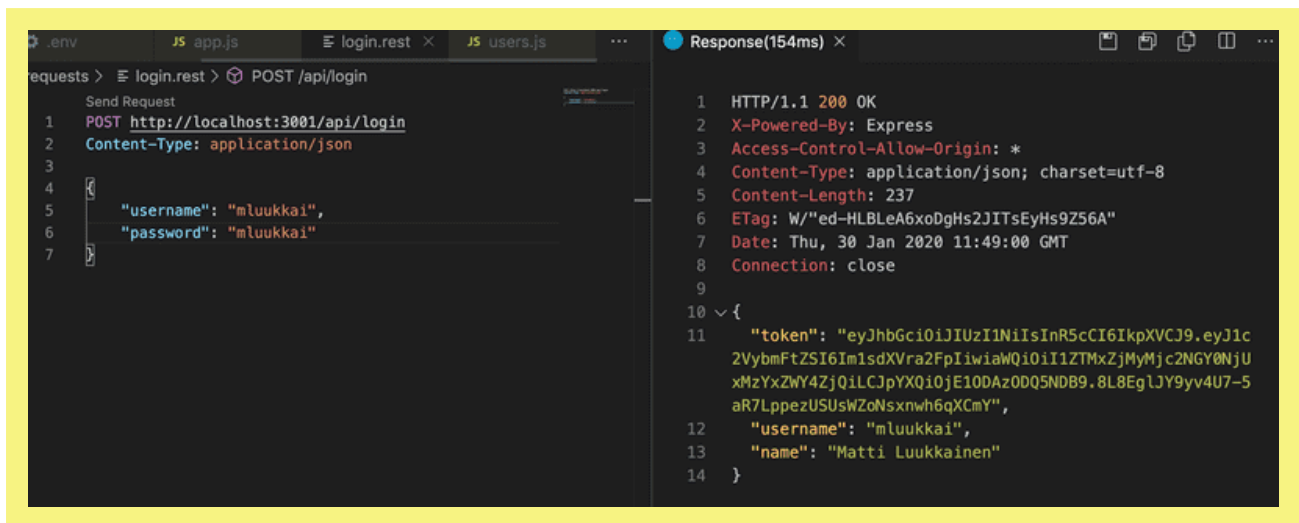
```
(node:32911) UnhandledPromiseRejectionWarning: Error: secretOrPrivateKey must have a value
    at Object.module.exports [as sign] (/Users/mluukkai/opetus/_2019fullstack-koodit/osa3/notes-backend/node_modules/jsonwebtoken/sign.js:101:20)
    at loginRouter.post (/Users/mluukkai/opetus/_2019fullstack-koodit/osa3/notes-backend/controllers/login.js:26:21)
(node:32911) UnhandledPromiseRejectionWarning: Unhandled promise rejection. This error originated either by throwing inside of an async function without a catch block, or by rejecting a promise which was not handled with .catch(). (rejection id: 2)
```

[copie](#)

La commande `jwt.sign(userForToken, process.env.SECRET)` échoue. Nous avons oublié de définir une valeur pour la variable d'environnement *SECRET*. Il peut s'agir de n'importe quelle

chaîne. Lorsque nous définissons la valeur dans le fichier `.env` (et redémarrons le serveur), la connexion fonctionne.


Une connexion réussie renvoie les détails de l'utilisateur et le jeton :



```
requests > login.rest > POST /api/login
Send Request
1 POST http://localhost:3001/api/login
2 Content-Type: application/json
3
4 {
5   "username": "mluukkai",
6   "password": "mluukkai"
7 }

Response(154ms) X
1 HTTP/1.1 200 OK
2 X-Powered-By: Express
3 Access-Control-Allow-Origin: *
4 Content-Type: application/json; charset=utf-8
5 Content-Length: 237
6 ETag: W/"ed-HLBLEA6xoDgHs2JITsEyHs9Z56A"
7 Date: Thu, 30 Jan 2020 11:49:00 GMT
8 Connection: close
9
10 {
11   "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6Im1sdXVra2FpIiwiaWQiOiIiIiwiaWF0IjE1ZTMxZjMyMjc2NGY0NjUxMzYxZWY4ZjQlLCJpYXQiOiJlE1ODAzODQ5NDh9.8L8EglJY9yv4U7-5aR7LppeZUSUsWZoNsxnwh6qXCmY",
12   "username": "mluukkai",
13   "name": "Matti Luukkainen"
14 }
```

Un nom d'utilisateur ou un mot de passe incorrect renvoie un message d'erreur et le code d'état approprié :



```
requests > login.rest > POST /api/login
Send Request
1 POST http://localhost:3001/api/login
2 Content-Type: application/json
3
4 {
5   "username": "mluukkai",
6   "password": "wrong"
7 }

1 HTTP/1.1 401 Unauthorized
2 X-Powered-By: Express
3 Access-Control-Allow-Origin: *
4 Content-Type: application/json; charset=utf-8
5 Content-Length: 40
6 ETag: W/"28-o0F+kGSS37KN6k7gEZFvLtWpuSE"
7 Date: Thu, 30 Jan 2020 12:50:52 GMT
8 Connection: close
9
10 {
11   "error": "invalid username or password"
12 }
```

Limiter la création de nouvelles notes aux utilisateurs connectés

Modifions la création de nouvelles notes afin qu'elle ne soit possible que si la demande de publication est associée à un jeton valide. La note est ensuite enregistrée dans la liste de notes de l'utilisateur identifié par le jeton.

Il existe plusieurs façons d'envoyer le jeton du navigateur au serveur. Nous utiliserons l'en-tête `Authorization`. L'en-tête indique également quel schéma d'authentification est utilisé. Cela peut être nécessaire si le serveur propose plusieurs façons de s'authentifier. L'identification du schéma indique au serveur comment les informations d'identification jointes doivent être interprétées.

Le schéma *Bearer* est adapté à nos besoins.

En pratique, cela signifie que si le jeton est, par exemple, la chaîne `eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6Im1sdXVra2FpIiwiaWQiOiIiIiwiaWF0IjE1ZTMxZjMyMjc2NGY0NjUxMzYxZWY4ZjQlLCJpYXQiOiJlE1ODAzODQ5NDh9.8L8EglJY9yv4U7-5aR7LppeZUSUsWZoNsxnwh6qXCmY`, l'en-tête `Authorization` aura la valeur :

Porteur eyJhbGciOiJIUzI1NiIsInR5c2VybmFtZSI6Im1sdXVra2FpIiwiaW

copie

La création de nouvelles notes changera comme suit (*controllers/notes.js*) :

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

// ...
const getTokenFrom = request => {
  const authorization = request.get('authorization')
  if (authorization && authorization.startsWith('Bearer ')) {
    return authorization.replace('Bearer ', '')
  }
  return null
}

notesRouter.post('/', async (request, response) => {
  const body = request.body
  const decodedToken = jwt.verify(getTokenFrom(request), process.env.SECRET)
  if (!decodedToken.id) {
    return response.status(401).json({ error: 'token invalid' })
  }
  const user = await User.findById(decodedToken.id)

  const note = new Note({
    content: body.content,
    important: body.important === undefined ? false : body.important,
    user: user._id
  })

  const savedNote = await note.save()
  user.notes = user.notes.concat(savedNote._id)
  await user.save()

  response.json(savedNote)
})
```

copie

La fonction d'assistance `getTokenFrom` isole le jeton de l' en-tête *d'autorisation* . La validité du jeton est vérifiée avec `jwt.verify` . La méthode décode également le jeton ou renvoie l'objet sur lequel le jeton est basé.

```
const decodedToken = jwt.verify(token, process.env.SECRET)
```

copie

Si le jeton est manquant ou non valide, l'exception *JsonWebTokenError* est générée. Nous devons étendre le middleware de gestion des erreurs pour prendre en charge ce cas particulier :

```
const errorHandler = (error, request, response, next) => {
  if (error.name === 'CastError') {
    return response.status(400).send({ error: 'malformatted id' })
  }
}
```

copie

```

    } else if (error.name === 'ValidationError') {
      return response.status(400).json({ error: error.message })
    } else if (error.name === 'MongoServerError' && error.message.includes('E11000 duplicate key
error')) {
      return response.status(400).json({ error: 'expected `username` to be unique' })
    } else if (error.name === 'JsonWebTokenError') {
      return response.status(401).json({ error: 'token invalid' })
    }

    next(error)
  }
}

```

L'objet décodé à partir du jeton contient les champs *nom d'utilisateur* et *identifiant*, qui indiquent au serveur qui a effectué la demande.

If the object decoded from the token does not contain the user's identity (`decodedToken.id` is undefined), error status code 401 unauthorized is returned and the reason for the failure is explained in the response body.

```

if (!decodedToken.id) {
  return response.status(401).json({
    error: 'token invalid'
  })
}

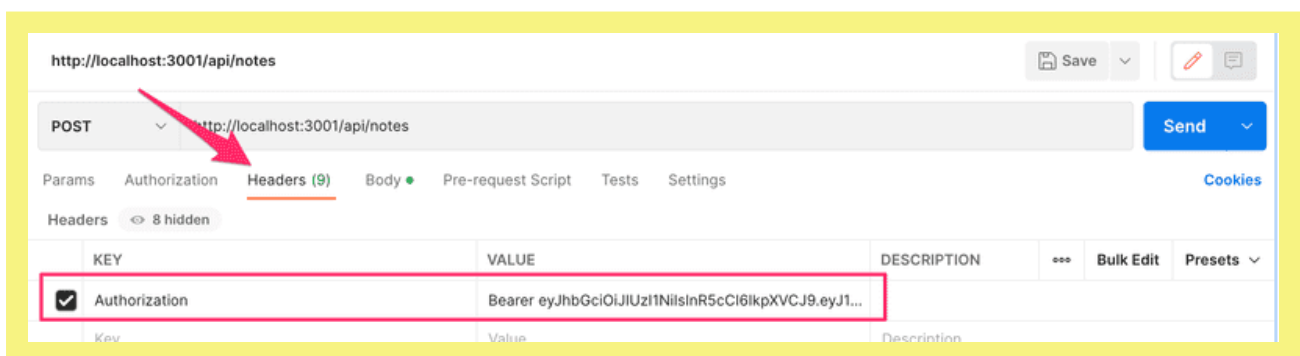
```

[copy](#)

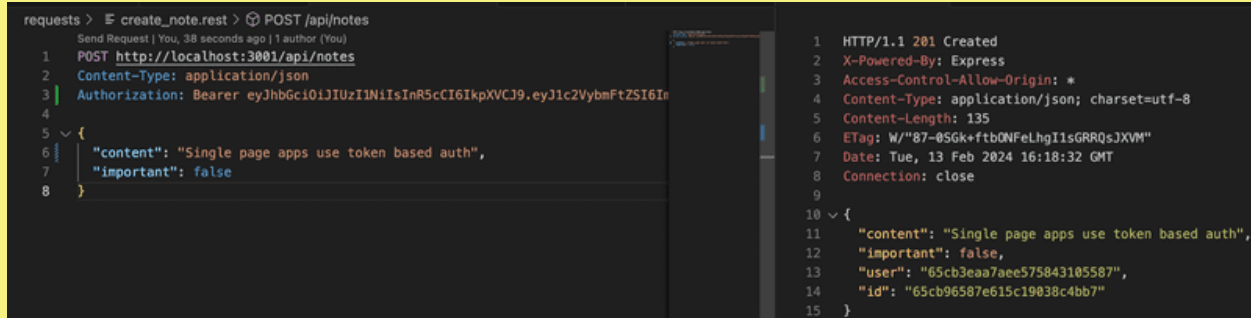
When the identity of the maker of the request is resolved, the execution continues as before.

A new note can now be created using Postman if the *authorization* header is given the correct value, the string *Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1...*, where the second value is the token returned by the *login* operation.

Using Postman this looks as follows:



and with Visual Studio Code REST client



Current application code can be found on [GitHub](#), branch *part4-9*.

If the application has multiple interfaces requiring identification, JWT's validation should be separated into its own middleware. An existing library like `express-jwt` could also be used.

Problems of Token-based authentication

Token authentication is pretty easy to implement, but it contains one problem. Once the API user, eg. a React app gets a token, the API has a blind trust to the token holder. What if the access rights of the token holder should be revoked?

There are two solutions to the problem. The easier one is to limit the validity period of a token:

```
loginRouter.post('/', async (request, response) => {
  const { username, password } = request.body

  const user = await User.findOne({ username })
  const passwordCorrect = user === null
    ? false
    : await bcrypt.compare(password, user.passwordHash)

  if (!(user && passwordCorrect)) {
    return response.status(401).json({
      error: 'invalid username or password'
    })
  }

  const userForToken = {
    username: user.username,
    id: user._id,
  }

  // token expires in 60*60 seconds, that is, in one hour
  const token = jwt.sign(
    userForToken,
    process.env.SECRET,
    { expiresIn: 60*60 }
  )

  response
    .status(200)
    .send({ token, username: user.username, name: user.name })
})
```


Once the token expires, the client app needs to get a new token. Usually, this happens by forcing the user to re-login to the app.

The error handling middleware should be extended to give a proper error in the case of an expired token:

```
const errorHandler = (error, request, response, next) => {
  logger.error(error.message)

  if (error.name === 'CastError') {
    return response.status(400).send({ error: 'malformatted id' })
  } else if (error.name === 'ValidationError') {
    return response.status(400).json({ error: error.message })
  } else if (error.name === 'MongoServerError' && error.message.includes('E11000 duplicate key error')) {
    return response.status(400).json({
      error: 'expected `username` to be unique'
    })
  } else if (error.name === 'JsonWebTokenError') {
    return response.status(401).json({
      error: 'invalid token'
    })
  } else if (error.name === 'TokenExpiredError') {
    return response.status(401).json({
      error: 'token expired'
    })
  }

  next(error)
}
```

[copy](#)

The shorter the expiration time, the more safe the solution is. So if the token gets into the wrong hands or user access to the system needs to be revoked, the token is only usable for a limited amount of time. On the other hand, a short expiration time forces a potential pain to a user, one must login to the system more frequently.

The other solution is to save info about each token to the backend database and to check for each API request if the access rights corresponding to the tokens are still valid. With this scheme, access rights can be revoked at any time. This kind of solution is often called a *server-side session*.

The negative aspect of server-side sessions is the increased complexity in the backend and also the effect on performance since the token validity needs to be checked for each API request to the database. Database access is considerably slower compared to checking the validity of the token itself. That is why it is quite common to save the session corresponding to a token to a *key-value database* such as Redis, that is limited in functionality compared to eg. MongoDB or a relational database, but extremely fast in some usage scenarios.

When server-side sessions are used, the token is quite often just a random string, that does not include any information about the user as it is quite often the case when jwt-tokens are used. For each API request, the server fetches the relevant information about the identity of the user from the database. It is also quite usual that instead of using Authorization-header, *cookies* are used as the mechanism for transferring the token between the client and the server.

End notes

There have been many changes to the code which have caused a typical problem for a fast-paced software project: most of the tests have broken. Because this part of the course is already jammed with new information, we will leave fixing the tests to a non-compulsory exercise.

Username, password and applications using token authentication must always be used over HTTPS. We could use a Node HTTPS server in our application instead of the HTTP server (it requires more configuration). On the other hand, the production version of our application is in Fly.io, so our application stays secure: Fly.io routes all traffic between a browser and the Fly.io server over HTTPS.

We will implement login to the frontend in the next part.

NOTE: At this stage, in the deployed notes app, it is expected that the creating a note feature will stop working as the backend login feature is not yet linked to the frontend.

Exercises 4.15.-4.23.

In the next exercises, the basics of user management will be implemented for the Bloglist application. The safest way is to follow the course material from part 4 chapter User administration to the chapter Token authentication. You can of course also use your creativity.

One more warning: If you notice you are mixing `async/await` and `then` calls, it is 99% certain you are doing something wrong. Use either or, never both.

4.15: Blog List Expansion, step 3

Implement a way to create new users by doing an HTTP POST request to address `api/users`. Users have a `username`, `password` and `name`.

Do not save passwords to the database as clear text, but use the `bcrypt` library like we did in part 4 chapter Creating users.

NB Some Windows users have had problems with `bcrypt`. If you run into problems, remove the library with command

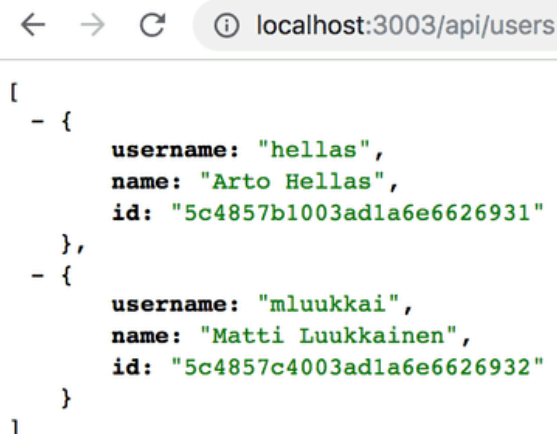
```
npm uninstall bcrypt
```

A small, light gray button with the word "copy" in a sans-serif font.

and install `bcryptjs` instead.

Implement a way to see the details of all users by doing a suitable HTTP request.

The list of users can, for example, look as follows:



```
[
  - {
    username: "hellas",
    name: "Arto Hellas",
    id: "5c4857b1003ad1a6e6626931"
  },
  - {
    username: "mluukkai",
    name: "Matti Luukkainen",
    id: "5c4857c4003ad1a6e6626932"
  }
]
```

4.16*: Blog List Expansion, step 4

Add a feature which adds the following restrictions to creating new users: Both username and password must be given and both must be at least 3 characters long. The username must be unique.

The operation must respond with a suitable status code and some kind of an error message if an invalid user is created.

NB Do not test password restrictions with Mongoose validations. It is not a good idea because the password received by the backend and the password hash saved to the database are not the same thing. The password length should be validated in the controller as we did in part 3 before using Mongoose validation.

Also, **implement tests** that ensure invalid users are not created and that an invalid add user operation returns a suitable status code and error message.

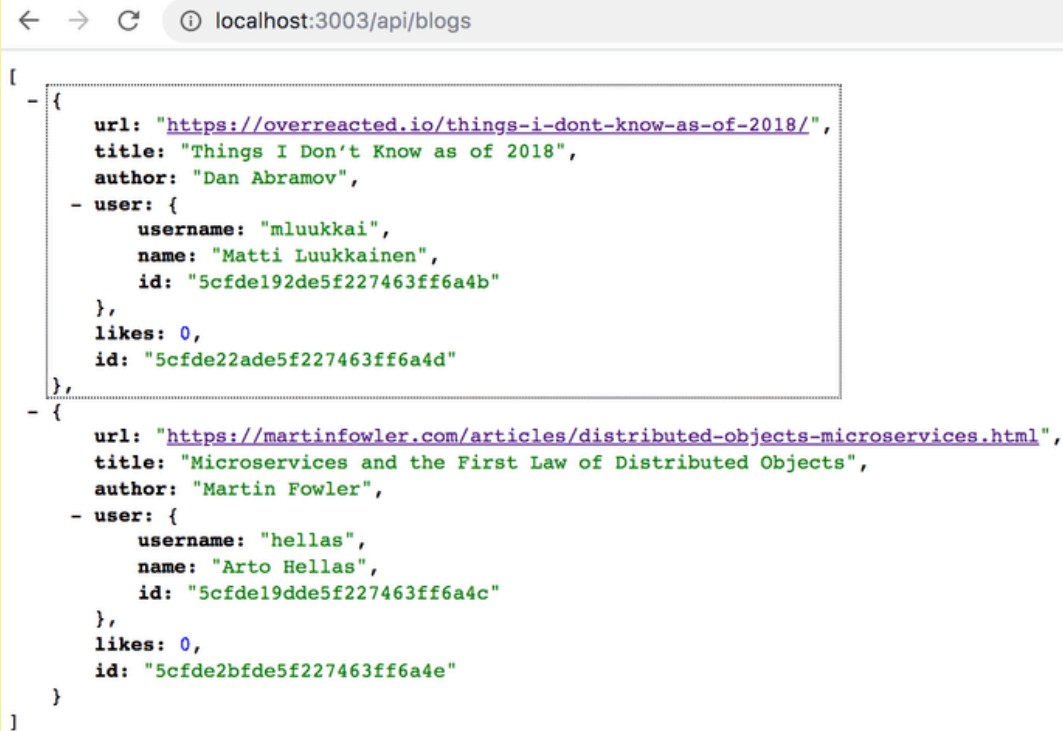
NB if you decide to define tests on multiple files, you should note that by default each test file is executed in its own process (see `Test execution model` in the documentation). The consequence of this is that different test files are executed at the same time. Since the tests share the same database, simultaneous execution may cause problems, which can be avoided by executing the tests with the option `--test-concurrency=1`, i.e. defining them to be executed sequentially.

4.17: Blog List Expansion, step 5

Expand blogs so that each blog contains information on the creator of the blog.

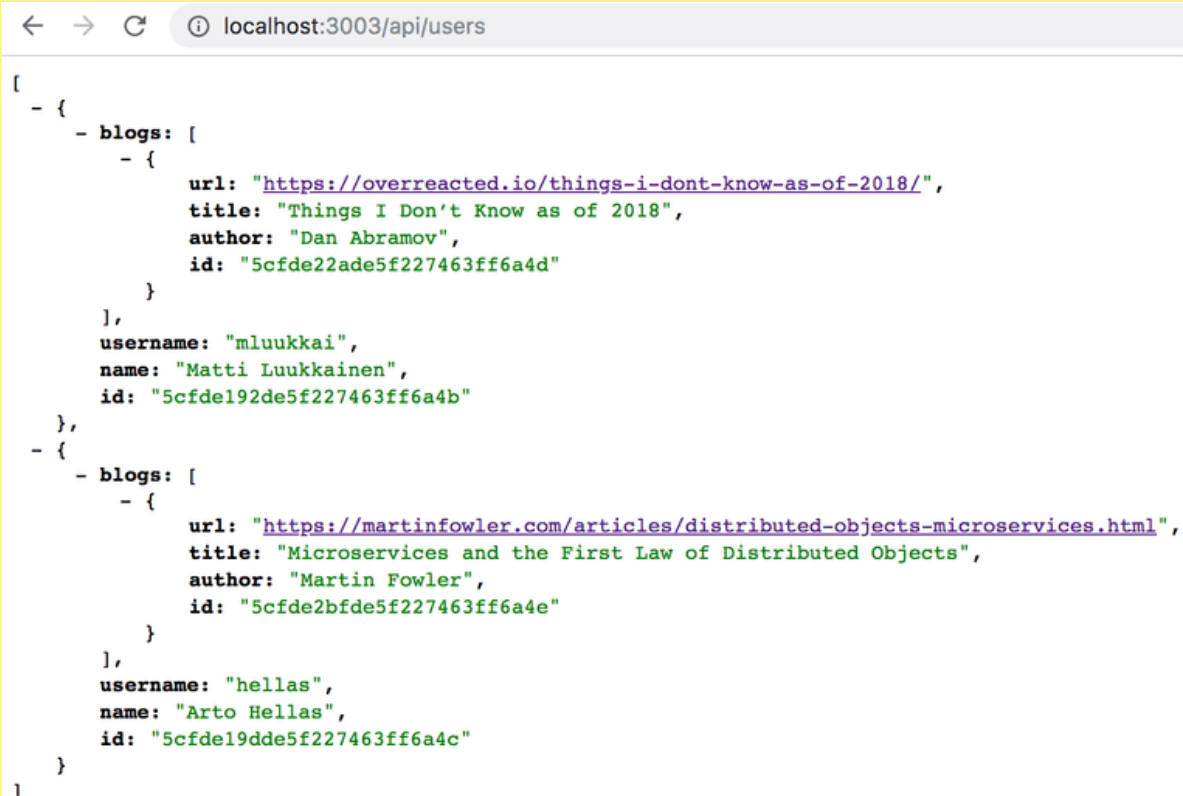
Modify adding new blogs so that when a new blog is created, *any* user from the database is designated as its creator (for example the one found first). Implement this according to part 4 chapter `populate`. Which user is designated as the creator does not matter just yet. The functionality is finished in exercise 4.19.

Modify listing all blogs so that the creator's user information is displayed with the blog:



```
[
  - {
    url: "https://overreacted.io/things-i-dont-know-as-of-2018/",
    title: "Things I Don't Know as of 2018",
    author: "Dan Abramov",
    - user: {
      username: "mluukkai",
      name: "Matti Luukkainen",
      id: "5cfde192de5f227463ff6a4b"
    },
    likes: 0,
    id: "5cfde22ade5f227463ff6a4d"
  },
  - {
    url: "https://martinfowler.com/articles/distributed-objects-microservices.html",
    title: "Microservices and the First Law of Distributed Objects",
    author: "Martin Fowler",
    - user: {
      username: "hellas",
      name: "Arto Hellas",
      id: "5cfde19dde5f227463ff6a4c"
    },
    likes: 0,
    id: "5cfde2bfde5f227463ff6a4e"
  }
]
```

and listing all users also displays the blogs created by each user:



```
[
  - {
    - blogs: [
      - {
        url: "https://overreacted.io/things-i-dont-know-as-of-2018/",
        title: "Things I Don't Know as of 2018",
        author: "Dan Abramov",
        id: "5cfde22ade5f227463ff6a4d"
      }
    ],
    username: "mluukkai",
    name: "Matti Luukkainen",
    id: "5cfde192de5f227463ff6a4b"
  },
  - {
    - blogs: [
      - {
        url: "https://martinfowler.com/articles/distributed-objects-microservices.html",
        title: "Microservices and the First Law of Distributed Objects",
        author: "Martin Fowler",
        id: "5cfde2bfde5f227463ff6a4e"
      }
    ],
    username: "hellas",
    name: "Arto Hellas",
    id: "5cfde19dde5f227463ff6a4c"
  }
]
```

4.18: Blog List Expansion, step 6

Implement token-based authentication according to part 4 chapter Token authentication.

4.19: Blog List Expansion, step 7

Modify adding new blogs so that it is only possible if a valid token is sent with the HTTP POST request. The user identified by the token is designated as the creator of the blog.

4.20*: Blog List Expansion, step 8

This example from part 4 shows taking the token from the header with the `getTokenFrom` helper function in `controllers/blogs.js`.

If you used the same solution, refactor taking the token to a middleware. The middleware should take the token from the `Authorization` header and assign it to the `token` field of the `request` object.

In other words, if you register this middleware in the `app.js` file before all routes

```
app.use(middleware.tokenExtractor)
```

[copy](#)

Routes can access the token with `request.token` :

```
blogsRouter.post('/', async (request, response) => {  
  // ..  
  const decodedToken = jwt.verify(request.token, process.env.SECRET)  
  // ..  
})
```

[copy](#)

Remember that a normal middleware function is a function with three parameters, that at the end calls the last parameter `next` to move the control to the next middleware:

```
const tokenExtractor = (request, response, next) => {  
  // code that extracts the token  
  
  next()  
}
```

[copy](#)

4.21*: Blog List Expansion, step 9

Change the delete blog operation so that a blog can be deleted only by the user who added it. Therefore, deleting a blog is possible only if the token sent with the request is the same as that of the blog's creator.

If deleting a blog is attempted without a token or by an invalid user, the operation should return a suitable status code.

Note that if you fetch a blog from the database,

```
const blog = await Blog.findById(...)
```

[copy](#)

the field `blog.user` does not contain a string, but an object. So if you want to compare the ID of the object fetched from the database and a string ID, a normal comparison operation does not work. The ID fetched from the database must be parsed into a string first.

```
if ( blog.user.toString() === userid.toString() ) ...
```

[copy](#)

4.22*: Blog List Expansion, step 10

Both the new blog creation and blog deletion need to find out the identity of the user who is doing the operation. The middleware `tokenExtractor` that we did in exercise 4.20 helps but still both the handlers of `post` and `delete` operations need to find out who the user holding a specific token is.

Now create a new middleware `userExtractor`, that finds out the user and sets it to the request object. When you register the middleware in `app.js`

```
app.use(middleware.userExtractor)
```

[copy](#)

the user will be set in the field `request.user` :

```
blogsRouter.post('/', async (request, response) => {
  // get user from request object
  const user = request.user
  // ..
})

blogsRouter.delete('/:id', async (request, response) => {
  // get user from request object
  const user = request.user
  // ..
})
```

[copy](#)

Note that it is possible to register a middleware only for a specific set of routes. So instead of using `userExtractor` with all the routes,

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

// use the middleware in all routes
app.use(middleware.userExtractor)

app.use('/api/blogs', blogsRouter)
app.use('/api/users', usersRouter)
app.use('/api/login', loginRouter)
```

[copy](#)

we could register it to be only executed with path `/api/blogs` routes:

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

// use the middleware only in /api/blogs routes
app.use('/api/blogs', middleware.userExtractor, blogsRouter)
app.use('/api/users', usersRouter)
app.use('/api/login', loginRouter)
```

copy

As can be seen, this happens by chaining multiple middlewares as the arguments of the function *use*. It would also be possible to register a middleware only for a specific operation:

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

router.post('/', middleware.userExtractor, async (request, response) => {
  // ...
})
```

copy

4.23*: Blog List Expansion, step 11

After adding token-based authentication the tests for adding a new blog broke down. Fix them. Also, write a new test to ensure adding a blog fails with the proper status code *401 Unauthorized* if a token is not provided.

This is most likely useful when doing the fix.

This is the last exercise for this part of the course and it's time to push your code to GitHub and mark all of your finished exercises to the exercise submission system.

[Propose changes to material](#)

Part 4c
Previous part

Part 4e
Next part

À propos du cours

Contenu du cours

FAQ

Partenaires

Défi



UNIVERSITY OF HELSINKI

