



## c User administration

We want to add user authentication and authorization to our application. Users should be stored in the database and every note should be linked to the user who created it. Deleting and editing a note should only be allowed for the user who created it.

Let's start by adding information about users to the database. There is a one-to-many relationship between the user (*User*) and notes (*Note*):



If we were working with a relational database the implementation would be straightforward. Both resources would have their separate database tables, and the id of the user who created a note would be stored in the notes table as a foreign key.

When working with document databases the situation is a bit different, as there are many different ways of modeling the situation.

The existing solution saves every note in the *notes collection* in the database. If we do not want to change this existing collection, then the natural choice is to save users in their own collection, *users* for example.

Like with all document databases, we can use object id's in Mongo to reference documents in other collections. This is similar to using foreign keys in relational databases.

Traditionally document databases like Mongo do not support *join queries* that are available in relational databases, used for aggregating data from multiple tables. However starting from version 3.2. Mongo has supported lookup aggregation queries. We will not be taking a look at this functionality in this course.

If we need a functionality similar to join queries, we will implement it in our application code by making multiple queries. In certain situations Mongoose can take care of joining and aggregating data, which gives the appearance of a join query. However, even in these situations Mongoose

makes multiple queries to the database in the background.

## References across collections

If we were using a relational database the note would contain a *reference key* to the user who created it. In document databases we can do the same thing.

Let's assume that the *users* collection contains two users:

```
[
  {
    username: 'mluukkai',
    _id: 123456,
  },
  {
    username: 'hellas',
    _id: 141414,
  },
];
```

The *notes* collection contains three notes that all have a *user* field that references a user in the *users* collection:

```
[
  {
    content: 'HTML is easy',
    important: false,
    _id: 221212,
    user: 123456,
  },
  {
    content: 'The most important operations of HTTP protocol are GET and POST',
    important: true,
    _id: 221255,
    user: 123456,
  },
  {
    content: 'A proper dinosaur codes with Java',
    important: false,
    _id: 221244,
    user: 141414,
  },
];
```

Document databases do not demand the foreign key to be stored in the note resources, it could *also* be stored in the users collection, or even both:

```
[
  {
    username: 'mluukkai',
    _id: 123456,
    notes: [221212, 221255],
  },
  {
    username: 'hellas',
    _id: 141414,
    notes: [221244],
  },
]
```

Since users can have many notes, the related ids are stored in an array in the *notes* field.

Document databases also offer a radically different way of organizing the data: In some situations it might be beneficial to nest the entire notes array as a part of the documents in the users collection:

```
[
  {
    username: 'mluukkai',
    _id: 123456,
    notes: [
      {
        content: 'HTML is easy',
        important: false,
      },
      {
        content: 'The most important operations of HTTP protocol are GET and POST',
        important: true,
      },
    ],
  },
  {
    username: 'hellas',
    _id: 141414,
    notes: [
      {
        content: 'A proper dinosaur codes with Java',
        important: false,
      },
    ],
  },
]
```

In this schema notes would be tightly nested under users and the database would not generate

ids for them.

The structure and schema of the database is not as self-evident as it was with relational databases. The chosen schema must be one which supports the use cases of the application the best. This is not a simple design decision to make, as all use cases of the applications are not known when the design decision is made.

Paradoxically, schema-less databases like Mongo require developers to make far more radical design decisions about data organization at the beginning of the project than relational databases with schemas. On average, relational databases offer a more-or-less suitable way of organizing data for many applications.

## Mongoose schema for users

In this case, we make the decision to store the ids of the notes created by the user in the user document. Let's define the model for representing a user in the *models/user.js* file:

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

const userSchema = new mongoose.Schema({
  username: String,
  name: String,
  passwordHash: String,
  notes: [
    {
      type: mongoose.Schema.Types.ObjectId,
      ref: 'Note'
    }
  ],
})

userSchema.set('toJSON', {
  transform: (document, returnedObject) => {
    returnedObject.id = returnedObject._id.toString()
    delete returnedObject._id
    delete returnedObject.__v
    // the passwordHash should not be revealed
    delete returnedObject.passwordHash
  }
})

const User = mongoose.model('User', userSchema)

module.exports = User
```

The ids of the notes are stored within the user document as an array of Mongo ids. The definition is as follows:

```
{
  type: mongoose.Schema.Types.ObjectId,
  ref: 'Note'
}
```

The type of the field is *ObjectId* that references *note*-style documents. Mongo does not inherently know that this is a field that references notes, the syntax is purely related to and defined by Mongoose.

Let's expand the schema of the note defined in the *model/note.js* file so that the note contains information about the user who created it:

```
const noteSchema = new mongoose.Schema({
  content: {
    type: String,
    required: true,
    minlength: 5
  },
  date: Date,
  important: Boolean,
  user: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'User'
  }
})
```

In stark contrast to the conventions of relational databases, *references are now stored in both documents*: the note references the user who created it, and the user has an array of references to all of the notes created by them.

## Creating users

Let's implement a route for creating new users. Users have a unique *username*, a *name* and something called a *passwordHash*. The password hash is the output of a one-way hash function applied to the user's password. It is never wise to store unencrypted plain text passwords in the database!

Let's install the bcrypt package for generating the password hashes:

```
npm install bcrypt
```

Creating new users happens in compliance with the RESTful conventions discussed in part 3, by making an HTTP POST request to the *users* path.

Let's define a separate *router* for dealing with users in a new *controllers/users.js* file. Let's take the router into use in our application in the *app.js* file, so that it handles requests made to the */api/users* url:

```
const usersRouter = require('./controllers/users')

// ...

app.use('/api/users', usersRouter)
```

The contents of the file that defines the router are as follows:

```
const bcrypt = require('bcrypt')
const usersRouter = require('express').Router()
const User = require('../models/user')

usersRouter.post('/', async (request, response) => {
  const body = request.body

  const saltRounds = 10
  const passwordHash = await bcrypt.hash(body.password, saltRounds)

  const user = new User({
    username: body.username,
    name: body.name,
    passwordHash,
  })

  const savedUser = await user.save()

  response.json(savedUser)
})

module.exports = usersRouter
```

The password sent in the request is *not* stored in the database. We store the *hash* of the password that is generated with the `bcrypt.hash` function.

The fundamentals of storing passwords are outside the scope of this course material. We will not discuss what the magic number 10 assigned to the saltRounds variable means, but you can read more about it in the linked material.

Our current code does not contain any error handling or input validation for verifying that the username and password are in the desired format.

The new feature can and should initially be tested manually with a tool like Postman. However testing things manually will quickly become too cumbersome, especially once we implement

functionality that enforces usernames to be unique.

It takes much less effort to write automated tests, and it will make the development of our application much easier.

Our initial tests could look like this:

```
const bcrypt = require('bcrypt')
const User = require('../models/user')

//...

describe('when there is initially one user in db', () => {
  beforeEach(async () => {
    await User.deleteMany({})

    const passwordHash = await bcrypt.hash('sekret', 10)
    const user = new User({ username: 'root', passwordHash })

    await user.save()
  })

  test('creation succeeds with a fresh username', async () => {
    const usersAtStart = await helper.usersInDb()

    const newUser = {
      username: 'mluukkai',
      name: 'Matti Luukkainen',
      password: 'salainen',
    }

    await api
      .post('/api/users')
      .send(newUser)
      .expect(200)
      .expect('Content-Type', /application\/json/)

    const usersAtEnd = await helper.usersInDb()
    expect(usersAtEnd).toHaveLength(usersAtStart.length + 1)

    const usernames = usersAtEnd.map(u => u.username)
    expect(usernames).toContain(newUser.username)
  })
})
```

The tests use the `usersInDb()` helper function that we implemented in the `tests/test_helper.js` file. The function is used to help us verify the state of the database after a user is created:

```
const User = require('../models/user')

// ...

const usersInDb = async () => {
  const users = await User.find({})
  return users.map(u => u.toJSON())
}

module.exports = {
  initialNotes,
  nonExistingId,
  notesInDb,
  usersInDb,
}
```

The *beforeEach* block adds a user with the username *root* to the database. We can write a new test that verifies that a new user with the same username can not be created:

```
describe('when there is initially one user in db', () => {
  // ...

  test('creation fails with proper statuscode and message if username already taken', async () => {
    const usersAtStart = await helper.usersInDb()

    const newUser = {
      username: 'root',
      name: 'Superuser',
      password: 'salainen',
    }

    const result = await api
      .post('/api/users')
      .send(newUser)
      .expect(400)
      .expect('Content-Type', /application\/json/)

    expect(result.body.error).toContain(`username` to be unique`)

    const usersAtEnd = await helper.usersInDb()
    expect(usersAtEnd).toHaveLength(usersAtStart.length)
  })
})
```

The test case obviously will not pass at this point. We are essentially practicing test-driven development (TDD), where tests for new functionality are written before the functionality is implemented.



Let's validate the uniqueness of the username with the help of Mongoose validators. As we mentioned in exercise [3.19](#), Mongoose does not have a built-in validator for checking the uniqueness of a field. We can find a ready-made solution for this from the [mongoose-unique-validator](#) npm package. Let's install it:

```
npm install mongoose-unique-validator
```

We must make the following changes to the schema defined in the *models/user.js* file:

```
const mongoose = require('mongoose')
const uniqueValidator = require('mongoose-unique-validator')

const userSchema = new mongoose.Schema({
  username: {
    type: String,
    unique: true
  },
  name: String,
  passwordHash: String,
  notes: [
    {
      type: mongoose.Schema.Types.ObjectId,
      ref: 'Note'
    }
  ],
})

userSchema.plugin(uniqueValidator)

// ...
```

We could also implement other validations into the user creation. We could check that the username is long enough, that the username only consists of permitted characters, or that the password is strong enough. Implementing these functionalities is left as an optional exercise.

Before we move onward, let's add an initial implementation of a route handler that returns all of the users in the database:

```
usersRouter.get('/', async (request, response) => {
  const users = await User.find({})
  response.json(users)
})
```

For making new user in production or development environemnt, you may send POST request to

`/api/users/` via Postman or REST Client in following format:

```
{
  "notes": [],
  "username": "root",
  "name": "Superuser",
  "password": "salainen"
}
```

The list looks like this:



You can find the code for our current application in its entirety in the *part4-7* branch of [this github repository](#).

## Creating a new note

The code for creating a new note has to be updated so that the note is assigned to the user who created it.

Let's expand our current implementation so, that the information about the user who created a note is sent in the *userId* field of the request body:

```
const User = require('../models/user')

//...

notesRouter.post('/', async (request, response, next) => {
  const body = request.body

  const user = await User.findById(body.userId)
```

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

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

response.json(savedNote)
})
```

It's worth noting that the *user* object also changes. The *id* of the note is stored in the *notes* field:

```
const user = await User.findById(body.userId)

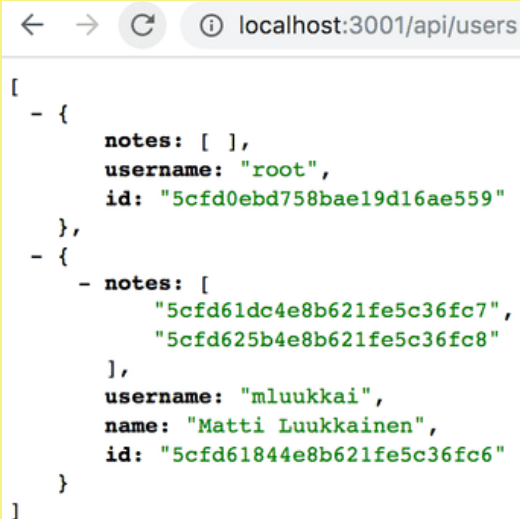
// ...

user.notes = user.notes.concat(savedNote._id)
await user.save()
```

Let's try to create a new note



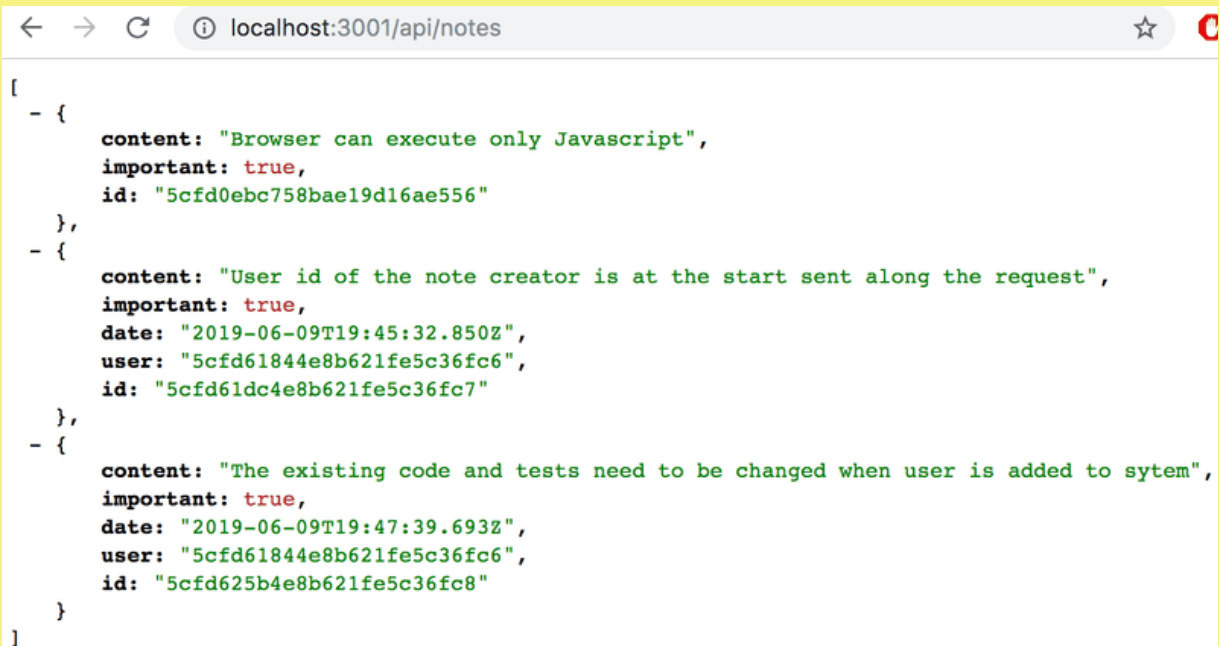
The operation appears to work. Let's add one more note and then visit the route for fetching all users:



```
[
  - {
    notes: [ ],
    username: "root",
    id: "5cfd0ebd758bae19d16ae559"
  },
  - {
    - notes: [
      "5cfd61dc4e8b621fe5c36fc7",
      "5cfd625b4e8b621fe5c36fc8"
    ],
    username: "mluukkai",
    name: "Matti Luukkainen",
    id: "5cfd61844e8b621fe5c36fc6"
  }
]
```

We can see that the user has two notes.

Likewise, the ids of the users who created the notes can be seen when we visit the route for fetching all notes:



```
[
  - {
    content: "Browser can execute only Javascript",
    important: true,
    id: "5cfd0ebc758bae19d16ae556"
  },
  - {
    content: "User id of the note creator is at the start sent along the request",
    important: true,
    date: "2019-06-09T19:45:32.850Z",
    user: "5cfd61844e8b621fe5c36fc6",
    id: "5cfd61dc4e8b621fe5c36fc7"
  },
  - {
    content: "The existing code and tests need to be changed when user is added to sytem",
    important: true,
    date: "2019-06-09T19:47:39.693Z",
    user: "5cfd61844e8b621fe5c36fc6",
    id: "5cfd625b4e8b621fe5c36fc8"
  }
]
```

## Populate

We would like our API to work in such a way, that when an HTTP GET request is made to the `/api/users` route, the user objects would also contain the contents of the user's notes, and not just their id. In a relational database, this functionality would be implemented with a *join query*.

As previously mentioned, document databases do not properly support join queries between collections, but the Mongoose library can do some of these joins for us. Mongoose accomplishes the join by doing multiple queries, which is different from join queries in relational databases which are *transactional*, meaning that the state of the database does not change during the time

that the query is made. With join queries in Mongoose, nothing can guarantee that the state between the collections being joined is consistent, meaning that if we make a query that joins the user and notes collections, the state of the collections may change during the query.

The Mongoose join is done with the populate method. Let's update the route that returns all users first:

```
usersRouter.get('/', async (request, response) => {
  const users = await User
    .find({}).populate('notes')

  response.json(users)
})
```

The populate method is chained after the *find* method making the initial query. The parameter given to the populate method defines that the *ids* referencing *note* objects in the *notes* field of the *user* document will be replaced by the referenced *note* documents.

The result is almost exactly what we wanted:



```
[
  - {
    - notes: [
      - {
        content: "user id of the note creator is at the start sent along the request",
        important: true,
        date: "2020-01-29T21:12:50.634Z",
        user: "5e31f322764f4651361ef8f4",
        id: "5e31f55286d91753a61271ec"
      },
      - {
        content: "The existing code and tests need to be changed when users are added to the system",
        important: true,
        date: "2020-01-29T21:14:33.282Z",
        user: "5e31f322764f4651361ef8f4",
        id: "5e31f5b986d91753a61271ed"
      }
    ],
    username: "mmluukkai",
    name: "Matti Luukkainen",
    id: "5e31f322764f4651361ef8f4"
  },
  - {
    notes: [ ],
    username: "root",
    name: "superuser",
    id: "5e31f339764f4651361ef8f5"
  }
]
```

We can use the populate parameter for choosing the fields we want to include from the documents. The selection of fields is done with the Mongo syntax:

```
usersRouter.get('/', async (request, response) => {
  const users = await User
    .find({}).populate('notes', { content: 1, date: 1 })
})
```

```
    response.json(users)
  });
```

The result is now exactly like we want it to be:



Let's also add a suitable population of user information to notes:

```
notesRouter.get('/', async (request, response) => {
  const notes = await Note
    .find({}).populate('user', { username: 1, name: 1 })

  response.json(notes)
});
```

Now the user's information is added to the *user* field of note objects.

```

- {
  content: "user id of the note creator is at the start sent along the request",
  important: true,
  date: "2020-01-29T21:12:50.634Z",
  user: {
    username: "mluukkai",
    name: "Matti Luukkainen",
    id: "5e31f322764f4651361ef8f4"
  },
  id: "5e31f55286d91753a61271ec"
},
- {
  content: "The existing code and tests need to be changed when users are added to the system",
  important: true,
  date: "2020-01-29T21:14:33.282Z",
  user: {
    username: "mluukkai",
    name: "Matti Luukkainen",
    id: "5e31f322764f4651361ef8f4"
  },
  id: "5e31f5b986d91753a61271ed"
}
]

```

It's important to understand that the database does not actually know that the ids stored in the *user* field of notes reference documents in the user collection.

The functionality of the *populate* method of Mongoose is based on the fact that we have defined "types" to the references in the Mongoose schema with the *ref* option:

About course

```

const noteSchema = new mongoose.Schema({
  content: {
    type: String,
    required: true,
    minlength: 5
  },
  date: Date,
  important: Boolean,
  user: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'User'
  }
})

```

You can find the code for our current application in its entirety in the *part4-8* branch of [this github repository](#).

[Propose changes to material](#)

Part 4b  
Previous part



Part 4d  
Next part

UNIVERSITY OF HELSINKI

**HOUSTON**