{() => fs}

Fullstack > Part 13 > Using relational databases with Sequelize

a Using relational databases with Sequelize

In this section we will explore node applications that use relational databases. During the section we will build a Node-backend using a relational database for a familiar note application from sections 3-5. To complete this part, you will need a reasonable knowledge of relational databases and SQL. There are many online courses on SQL databases, eg. SQLbolt and Intro to SQL by Khan Academy.

There are 24 exercises in this part, and you need to complete each exercise for completing the course. Exercises are submitted via the submissions system just like in the previous parts, but unlike parts 0 to 7, the submission goes to a different "course instance".

Advantages and disadvantages of document databases

We have used the MongoDB database in all the previous sections of the course. Mongo is a document database and one of its most characteristic features is that it is *schemaless*, i.e. the database has only a very limited awareness of what kind of data is stored in its collections. The schema of the database exists only in the program code, which interprets the data in a specific way, e.g. by identifying that some of the fields are references to objects in another collection.

In the example application of parts 3 and 4, the database stores notes and users.

A collection of *notes* that stores notes looks like the following:

```
[
  {
    "_id": "600c0e410d10256466898a6c",
    "content": "HTML is easy"
```

copy

```
    "date": 2021-01-23T11:53:37.292+00:00,
    "important": false
    "__v": 0
  },
  {
    "_id": "600c0edde86c7264ace9bb78",
    "content": "CSS is hard"
    "date": 2021-01-23T11:56:13.912+00:00,
    "important": true
    "__v": 0
  },
]
```

Users saved in the *users* collection looks like the following:

```
[
  {
    "_id": "600c0e410d10256466883a6a",
    "username": "mluukkai",
    "name": "Matti Luukkainen",
    "passwordHash" : "$2b$10$Df1yYJRiQuu3Sr4tUrk.SerVz1JKtBHlBOARfY0PBn/Uo7qr8Ocou",
    "__v": 9,
    notes: [
      "600c0edde86c7264ace9bb78",
      "600c0e410d10256466898a6c"
    ]
  },
]
```

MongoDB does know the types of the fields of the stored entities, but it has no information about which collection of entities the user record ids are referring to. MongoDB also does not care what fields the entities stored in the collections have. Therefore MongoDB leaves it entirely up to the programmer to ensure that the correct information is being stored in the database.

There are both advantages and disadvantages to not having a schema. One of the advantages is the flexibility that schema agnosticism brings: since the schema does not need to be defined at the database level, application development may be faster in certain cases, and easier, with less effort needed in defining and modifying the schema in any case. Problems with not having a schema are related to error-proneness: everything is left up to the programmer. The database itself has no way of checking whether the data in it is *honest*, i.e. whether all mandatory fields have values, whether the reference type fields refer to existing entities of the right type in general, etc.

The relational databases that are the focus of this section, on the other hand, lean heavily on the existence of a schema, and the advantages and disadvantages of schema databases are almost the opposite compared of the non-schema databases.

The reason why the the previous sections of the course used MongoDB is precisely because of its schema-less nature, which has made it easier to use the database for someone with little knowledge of

relational databases. For most of the use cases of this course, I personally would have chosen to use a relational database.

Application database

For our application we need a relational database. There are many options, but we will be using the currently most popular Open Source solution PostgreSQL. You can install Postgres (as the database is often called) on your machine, if you wish to do so. An easier option would be using Postgres as a cloud service, e.g. ElephantSQL.

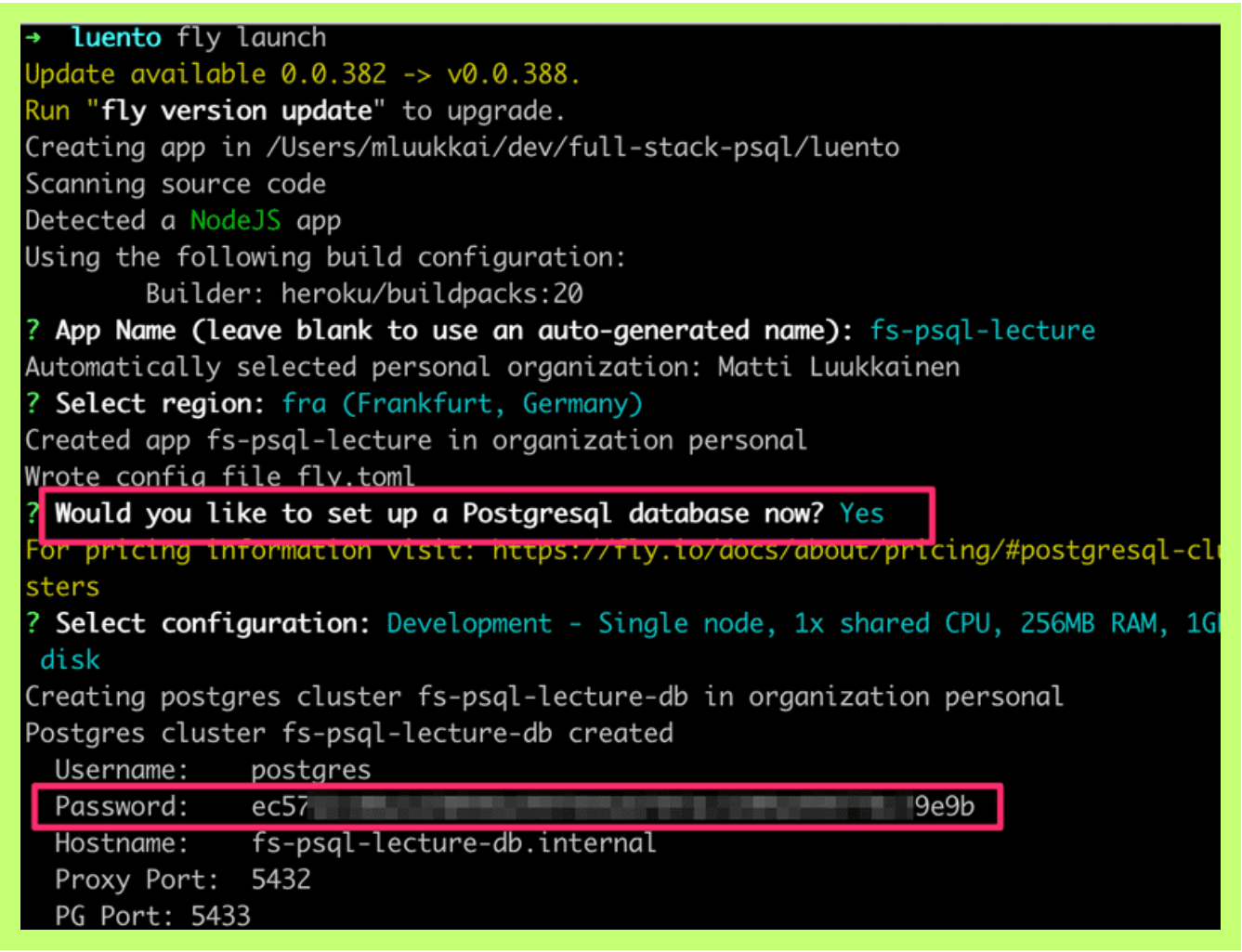
However, we will be taking advantage of the fact that it is possible to create a Postgres database for the application on the Fly.io and Heroku cloud service platforms, which are familiar from the parts 3 and 4.

In the theory material of this section, we will be building a Postgres-enabled version from the backend of the notes-storage application, which was built in sections 3 and 4.

Since we don't need any database in the cloud in this part (we only use the application locally), there is a possibility to use the lessons of the course part 12 and use Postgres locally with Docker. After the Postgres instructions for cloud services, we also give a short instruction on how to easily get Postgres up and running with Docker.

Fly.io

Let us create a new Fly.io-app by running the command `fly launch` in a directory where we shall add the code of the app. Let us also create the Postgres database for the app:



When creating the app, Fly.io reveals the password of the database that will be needed when connecting the app to the database. *This is the only time it is shown in plain text so it is essential to save it somewhere* (but not in any public place such as GitHub).

Note that if you only need the database, and are not planning to deploy the app to Fly.io, it is also possible to just create the database to Fly.io.

A psql console connection to the database can be opened as follows

```
flyctl postgres connect -a <app_name-db>
```

copy

in my case the app name is *fs-psql-lecture* so the command is the following:

```
flyctl postgres connect -a fs-psql-lecture-db
```

copy

Heroku

If Heroku is used, a new Heroku application is created when inside a suitable directory. After that a database is added to to the app:

```
heroku create
# Returns an app-name for the app you just created in heroku.

heroku addons:create heroku-postgresql:hobby-dev -a <app-name>
```

We can use the `heroku config` command to get the *connect string*, which is required to connect to the database:

```
heroku config -a <app-name>
=== cryptic-everglades-76708 Config Vars
DATABASE_URL: postgres://<username>:thepasswordishere@<host-of-postgres-addon>:5432/<db-name>
```

The database can be accessed by running `psql` command on the Heroku server as follows (note that the command parameters depend on the connection url of the Heroku database):

```
heroku run psql -h <host-of-postgres-addon> -p 5432 -U <username> <dbname> -a <app-name>
```

The commands asks the password and opens the psql console:

```
Password for user <username>:
psql (13.4 (Ubuntu 13.4-1.pgdg20.04+1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression: off)
Type "help" for help.

postgres=#
```

Docker

This instruction assumes that you master the basic use of Docker to the extent taught by e.g. part 12.

Start Postgres Docker image with the command

```
docker run -e POSTGRES_PASSWORD=mysecretpassword -p 5432:5432 postgres
```

A psql console connection to the database can be opened using the `docker exec` command. First you need to find out the id of the container:

```
$ docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED          STATUS          PORTS
NAMES
ff3f49eadf27   postgres   "docker-entrypoint.s…"   31 minutes ago   Up 31 minutes
0.0.0.0:5432->5432/tcp   great_raman
docker exec -it ff3f49eadf27 psql -U postgres postgres
psql (15.2 (Debian 15.2-1.pgdg110+1))
Type "help" for help.

postgres=#
```

Defined in this way, the data stored in the database is persisted only as long as the container exists. The data can be preserved by defining a volume for the data, see more here.

Using the psql console

Particularly when using a relational database, it is essential to access the database directly as well. There are many ways to do this, there are several different graphical user interfaces, such as pgAdmin. However, we will be using Postgres psql command-line tool.

When the console is opened, let's try the main psql command `\d`, which tells you the contents of the database:

```
Password for user <username>:
psql (13.4 (Ubuntu 13.4-1.pgdg20.04+1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256,
compression: off)
Type "help" for help.

postgres=# \d
Did not find any relations.
```

As you might guess, there is currently nothing in the database.

Let's create a table for notes:

```
CREATE TABLE notes (
    id SERIAL PRIMARY KEY,
    content text NOT NULL,
    important boolean,
    date time
);
```

A few points: column *id* is defined as a *primary key*, which means that the value in the column id must be unique for each row in the table and the value must not be empty. The type for this column is defined as `SERIAL`, which is not the actual type but an abbreviation for an integer column to which Postgres automatically assigns a unique, increasing value when creating rows. The column named *content* with type text is defined in such a way that it must be assigned a value.

Let's look at the situation from the console. First, the `\d` command, which tells us what tables are in the database:

```
postgres=# \d
            List of relations
 Schema |     Name     |   Type   |  Owner
--------+--------------+----------+----------
 public | notes        | table    | username
 public | notes_id_seq | sequence | username
(2 rows)
```

In addition to the *notes* table, Postgres created a subtable called *notes_id_seq*, which keeps track of what value is assigned to the *id* column when creating the next note.

With the command `\d notes`, we can see how the *notes* table is defined:

```
postgres=# \d notes;
                                  Table "public.notes"
 Column    |          Type          | Collation | Nullable |             Default
-----------+------------------------+-----------+----------+-----------------------------
------
 id        | integer                |           | not null |
nextval('notes_id_seq'::regclass)
 content   | text                   |           | not null |
 important | boolean                |           |          |
 date      | time without time zone |           |          |
Indexes:
    "notes_pkey" PRIMARY KEY, btree (id)
```

Therefore the column *id* has a default value, which is obtained by calling the internal function of Postgres *nextval*.

Let's add some content to the table:

```
insert into notes (content, important) values ('Relational databases rule the world',
true);
insert into notes (content, important) values ('MongoDB is webscale', false);
```

And let's see what the created content looks like:

```
postgres=# select * from notes;
 id |               content               | important | date
----+-------------------------------------+-----------+------
  1 | relational databases rule the world | t         |
  2 | MongoDB is webscale                 | f         |
(2 rows)
```

If we try to store data in the database that is not according to the schema, it will not succeed. The value of a mandatory column cannot be missing:

```
postgres=# insert into notes (important) values (true);
ERROR: null value in column "content" of relation "notes" violates not-null constraint
DETAIL: Failing row contains (9, null, t, null).
```

The column value cannot be of the wrong type:

```
postgres=# insert into notes (content, important) values ('only valid data can be saved',
1);
ERROR: column "important" is of type boolean but expression is of type integer
LINE 1: ...tent, important) values ('only valid data can be saved', 1); ^
```

Columns that don't exist in the schema are not accepted either:

```
postgres=# insert into notes (content, important, value) values ('only valid data can be
saved', true, 10);
ERROR: column "value" of relation "notes" does not exist
LINE 1: insert into notes (content, important, value) values ('only ...
```

Next it's time to move on to accessing the database from the application.

Node application using a relational database

Let's start the application as usual with the *npm init* and install *nodemon* as a development dependency and also the following runtime dependencies:

```
npm install express dotenv pg sequelize
```

Of these, the latter sequelize is the library through which we use Postgres. Sequelize is a so-called Object relational mapping (ORM) library that allows you to store JavaScript objects in a relational

database without using the SQL language itself, similar to Mongoose that we used with MongoDB.

Let's test that we can connect successfully. Create the file *index.js* and add the following content:

```
require('dotenv').config()
const { Sequelize } = require('sequelize')

const sequelize = new Sequelize(process.env.DATABASE_URL)

const main = async () => {
  try {
    await sequelize.authenticate()
    console.log('Connection has been established successfully.')
    sequelize.close()
  } catch (error) {
    console.error('Unable to connect to the database:', error)
  }
}

main()
```

Note: if you use Heroku, you might need an extra option in connecting the database

```
const sequelize = new Sequelize(process.env.DATABASE_URL, {
  dialectOptions: {
    ssl: {
      require: true,
      rejectUnauthorized: false
    }
  },
})
```

The database *connect string*, that contains the database address and the credentials must be defined in the file *.env*

If Heroku is used, the connect string can be seen by using the command `heroku config`. The contents of the file *.env* should be something like the following:

```
$ cat .env
DATABASE_URL=postgres://<username>:thepasswordishere@ec2-54-83-137-206.compute-1.amazonaws.com:5432/<databasename>
```

When using Fly.io, the local connection to the database should first be enabled by tunneling the localhost port 5432 to the Fly.io database port using the following command

```
flyctl proxy 5432 -a <app-name>-db
```

in my case the command is

```
flyctl proxy 5432 -a fs-psql-lecture-db
```

The command must be left running while the database is used. So do not close the console!

The Fly.io connect-string is of the form

```
$ cat .env
DATABASE_URL=postgres://postgres:thepasswordishere@127.0.0.1:5432/postgres
```

Password was shown when the database was created, so hopefully you have not lost it!

The last part of the connect string, *postgres* refers to the database name. The name could be any string but we use here *postgres* since it is the default database that is automatically created within a Postgres database. If needed, new databases can be created with the command CREATE DATABASE.

If you use Docker, the connect string is:

```
DATABASE_URL=postgres://postgres:mysecretpassword@localhost:5432/postgres
```

Once the connect string has been set up in the file *.env* we can test for a connection:

```
$ node index.js
Executing (default): SELECT 1+1 AS result
Connection has been established successfully.
```

If and when the connection works, we can then run the first query. Let's modify the program as follows:

```
require('dotenv').config()
const { Sequelize, QueryTypes } = require('sequelize')

const sequelize = new Sequelize(process.env.DATABASE_URL, {
  dialectOptions: {
    ssl: {
      require: true,
      rejectUnauthorized: false
```

```
    }
  },
});

const main = async () => {
  try {
    await sequelize.authenticate()
    const notes = await sequelize.query("SELECT * FROM notes", { type: QueryTypes.SELECT
})
    console.log(notes)
    sequelize.close()
  } catch (error) {
    console.error('Unable to connect to the database:', error)
  }
}

main()
```

Executing the application should print as follows:

```
Executing (default): SELECT * FROM notes
[
  {
    id: 1,
    content: 'Relational databases rule the world',
    important: true,
    date: null
  },
  {
    id: 2,
    content: 'MongoDB is webscale',
    important: false,
    date: null
  }
]
```

Even though Sequelize is an ORM library, which means there is little need to write SQL yourself when using it, we just used direct SQL with the sequelize method query.

Since everything seems to be working, let's change the application into a web application.

```
require('dotenv').config()
const { Sequelize, QueryTypes } = require('sequelize')
const express = require('express')
const app = express()

const sequelize = new Sequelize(process.env.DATABASE_URL, {
  dialectOptions: {
    ssl: {
      require: true,
```

```
      rejectUnauthorized: false
    }
  },
});

app.get('/api/notes', async (req, res) => {
  const notes = await sequelize.query("SELECT * FROM notes", { type: QueryTypes.SELECT })
  res.json(notes)
})
const PORT = process.env.PORT || 3001
app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`)
})
```

The application seems to be working. However, let's now switch to using Sequelize instead of SQL as it is intended to be used.

Model

When using Sequelize, each table in the database is represented by a model, which is effectively it's own JavaScript class. Let's now define the model *Note* corresponding to the table *notes* for the application by changing the code to the following format:

```
require('dotenv').config()
const { Sequelize, Model, DataTypes } = require('sequelize')
const express = require('express')
const app = express()

const sequelize = new Sequelize(process.env.DATABASE_URL, {
  dialectOptions: {
    ssl: {
      require: true,
      rejectUnauthorized: false
    }
  },
});

class Note extends Model {}
Note.init({
  id: {
    type: DataTypes.INTEGER,
    primaryKey: true,
    autoIncrement: true
  },
  content: {
    type: DataTypes.TEXT,
    allowNull: false
  },
  important: {
    type: DataTypes.BOOLEAN
  },
```

```
    date: {
      type: DataTypes.DATE
    }
  }, {
    sequelize,
    underscored: true,
    timestamps: false,
    modelName: 'note'
  })

  app.get('/api/notes', async (req, res) => {
    const notes = await Note.findAll()
    res.json(notes)
  })

  const PORT = process.env.PORT || 3001
  app.listen(PORT, () => {
    console.log(`Server running on port ${PORT}`)
  })
```

A few comments on the code: There is nothing very surprising about the *Note* definition of the model, each column has a type defined, as well as other properties if necessary, such as whether it is the main key of the table. The second parameter in the model definition contains the *sequelize* attribute as well as other configuration information. We also defined that the table does not have to use the timestamps columns (created_at and updated_at).

We also defined *underscored: true*, which means that table names are derived from model names as plural snake case versions. Practically this means that, if the name of the model, as in our case is "Note", then the name of the corresponding table is its plural version written with a lower case initial letter, i.e. *notes*. If, on the other hand, the name of the model would be "two-part", e.g. *StudyGroup*, then the name of the table would be *study_groups*. Sequelize automatically infers table names, but also allows explicitly defining them.

The same naming policy applies to columns as well. If we had defined that a note is associated with *creationYear*, i.e. information about the year it was created, we would define it in the model as follows:

```
Note.init({
  // ...
  creationYear: {
    type: DataTypes.INTEGER,
  },
})
```

The name of the corresponding column in the database would be *creation_year*. In code, reference to the column is always in the same format as in the model, i.e. in "camel case" format.

We have also defined *modelName: 'note'*, the default "model name" would be capitalized *Note*. However we want to have a lowercase initial, it will make a few things a bit more convenient going forward.

The database operation is easy to do using the query interface provided by models, the method findAll works exactly as it is assumed by it's name to work:

```
app.get('/api/notes', async (req, res) => {
  const notes = await Note.findAll()
  res.json(notes)
})
```

The console tells you that the method call *Note.findAll()* causes the following query:

```
Executing (default): SELECT "id", "content", "important", "date" FROM "notes" AS "note";
```

Next, let's implement an endpoint for creating new notes:

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

// ...

app.post('/api/notes', async (req, res) => {
  console.log(req.body)
  const note = await Note.create(req.body)
  res.json(note)
})
```

Creating a new note is done by calling the model's *Note* method create and passing as a parameter an object that defines the values of the columns.

Instead of the *create* method, it is also possible to save to a database using the build method first to create a Model-object from the desired data, and then calling the save method on it:

```
const note = Note.build(req.body)
await note.save()
```

Calling the *build* method does not save the object in the database yet, so it is still possible to edit the object before the actual save event:

```
const note = Note.build(req.body)
note.important = true
await note.save()
```

For the use case of the example code, the `create` method is better suited, so let's stick to that.

If the object being created is not valid, there is an error message as a result. For example, when trying to create a note without content, the operation fails, and the console reveals the reason to be *SequelizeValidationError: notNull Violation Note.content cannot be null*:

```
(node:39109) UnhandledPromiseRejectionWarning: SequelizeValidationError: notNull
Violation: Note.content cannot be null
    at InstanceValidator._validate (/Users/mluukkai/opetus/fs-
psql/node_modules/sequelize/lib/instance-validator.js:78:13)
    at processTicksAndRejections (internal/process/task_queues.js:93:5)
```

Let's add some simple error handling when adding a new note:

```
app.post('/api/notes', async (req, res) => {
  try {
    const note = await Note.create(req.body)
    return res.json(note)
  } catch(error) {
    return res.status(400).json({ error })
  }
})
```

Exercises 13.1.-13.3.

In the tasks of this section, we will build a blog application backend similar to the tasks in section 4, which should be compatible with the frontend in section 5 except for error handling. We will also add various features to the backend that the frontend in section 5 will not know how to use.

Exercise 13.1.

Create a GitHub repository for the application and create a new Fly.io or Heroku application for it, as well as a Postgres database. As mentioned here you might set up your database also somewhere else, and in that case the Fly.io of Heroku app is not needed.

Make sure you are able to establish a connection to the application database.

Exercise 13.2.

On the command-line, create a *blogs* table for the application with the following columns:

- id (unique, incrementing id)

- author (string)
- url (string that cannot be empty)
- title (string that cannot be empty)
- likes (integer with default value zero)

Add at least two blogs to the database.

Save the SQL-commands you used at the root of the application repository in a file called *commands.sql*

Exercise 13.3.

Create a functionality in your application which prints the blogs in the database on the command-line, e.g. as follows:

```
$ node cli.js
Executing (default): SELECT * FROM blogs
Dan Abramov: 'On let vs const', 0 likes
Laurenz Albe: 'Gaps in sequences in PostgreSQL', 0 likes
```

Creating database tables automatically

Our application now has one unpleasant side, it assumes that a database with exactly the right schema exists, i.e. that the table *notes* has been created with the appropriate *create table* command.

Since the program code is being stored on GitHub, it would make sense to also store the commands that create the database in the context of the program code, so that the database schema is definitely the same as what the program code is expecting. Sequelize is actually able to generate a schema automatically from the model definition by using the models method `sync`.

Let's now destroy the database from the console by entering the following command:

```
drop table notes;
```

The `\d` command reveals that the table has been lost from the database:

```
postgres=# \d
Did not find any relations.
```

The application no longer works.

Let's add the following command to the application immediately after the model *Note* is defined:

```
Note.sync()
```

When the application starts, the following is printed on the console:

```
Executing (default): CREATE TABLE IF NOT EXISTS "notes" ("id" SERIAL , "content" TEXT NOT
NULL, "important" BOOLEAN, "date" TIMESTAMP WITH TIME ZONE, PRIMARY KEY ("id"));
```

That is, when the application starts, the command *CREATE TABLE IF NOT EXISTS "notes"...* is executed which creates the table *notes* if it does not already exist.

Other operations

Let's complete the application with a few more operations.

Searching for a single note is possible with the method `findByPk`, because it is retrieved based on the id of the primary key:

```
app.get('/api/notes/:id', async (req, res) => {
  const note = await Note.findByPk(req.params.id)
  if (note) {
    res.json(note)
  } else {
    res.status(404).end()
  }
})
```

Retrieving a single note causes the following SQL command:

```
Executing (default): SELECT "id", "content", "important", "date" FROM "notes" AS "note"
WHERE "note". "id" = '1';
```

If no note is found, the operation returns *null*, and in this case the relevant status code is given.

Modifying the note is done as follows. Only the modification of the *important* field is supported, since the application's frontend does not need anything else:

```
app.put('/api/notes/:id', async (req, res) => {
  const note = await Note.findByPk(req.params.id)
```

```
  if (note) {
    note.important = req.body.important
    await note.save()
    res.json(note)
  } else {
    res.status(404).end()
  }
})
```

The object corresponding to the database row is retrieved from the database using the *findByPk* method, the object is modified and the result is saved by calling the *save* method of the object corresponding to the database row.

The current code for the application is in its entirety on GitHub, branch *part13-1*.

Printing the objects returned by Sequelize to the console

The JavaScript programmer's most important tool is *console.log*, whose aggressive use gets even the worst bugs under control. Let's add console printing to the single note path:

```
app.get('/api/notes/:id', async (req, res) => {
  const note = await Note.findByPk(req.params.id)
  if (note) {
    console.log(note)
    res.json(note)
  } else {
    res.status(404).end()
  }
})
```

We can see that the end result is not exactly what we expected:

```
note {
  dataValues: {
    id: 1,
    content: 'Notes are attached to a user',
    important: true,
    date: 2021-10-03T15:00:24.582Z,
  },
  _previousDataValues: {
    id: 1,
    content: 'Notes are attached to a user',
    important: true,
    date: 2021-10-03T15:00:24.582Z,
  },
  _changed: Set(0) {},
  _options: {
```

```
    isNewRecord: false,
    _schema: null,
    _schemaDelimiter: '',
    raw: true,
    attributes: [ 'id', 'content', 'important', 'date' ]
  },
  isNewRecord: false
}
```

In addition to the note information, all sorts of other things are printed on the console. We can reach the desired result by calling the model-object method `toJSON`:

```
app.get('/api/notes/:id', async (req, res) => {
  const note = await Note.findByPk(req.params.id)
  if (note) {
    console.log(note.toJSON())
    res.json(note)
  } else {
    res.status(404).end()
  }
})
```

Now the result is exactly what we want:

```
{ id: 1,
  content: 'MongoDB is webscale',
  important: false,
  date: 2021-10-09T13:52:58.693Z }
```

In the case of a collection of objects, the method toJSON does not work directly, the method must be called separately for each object in the collection:

```
app.get('/api/notes', async (req, res) => {
  const notes = await Note.findAll()

  console.log(notes.map(n=>n.toJSON()))

  res.json(notes)
})
```

The print looks like the following:

```
[ { id: 1,
    content: 'MongoDB is webscale',
```

```
    important: false,
    date: 2021-10-09T13:52:58.693Z },
  { id: 2,
    content: 'Relational databases rule the world',
    important: true,
    date: 2021-10-09T13:53:10.710Z } ]
```

However, perhaps a better solution is to turn the collection into JSON for printing by using the method JSON.stringify:

```
app.get('/api/notes', async (req, res) => {
  const notes = await Note.findAll()

  console.log(JSON.stringify(notes))

  res.json(notes)
})
```

This way is better especially if the objects in the collection contain other objects. It is also often useful to format the objects on the screen in a slightly more reader-friendly format. This can be done with the following command:

```
console.log(JSON.stringify(notes, null, 2))
```

The print looks like the following:

```
[
  {
    "id": 1,
    "content": "MongoDB is webscale",
    "important": false,
    "date": "2021-10-09T13:52:58.693Z"
  },
  {
    "id": 2,
    "content": "Relational databases rule the world",
    "important": true,
    "date": "2021-10-09T13:53:10.710Z"
  }
]
```

Exercise 13.4.

Exercise 13.4.

Transform your application into a web application that supports the following operations

- GET api/blogs (list all blogs)
- POST api/blogs (add a new blog)
- DELETE api/blogs/:id (delete a blog)

Propose changes to material

Part 12
Previous part

Part 13b
Next part

About course

Course contents

FAQ

Partners

Challenge


UNIVERSITY OF HELSINKI


HOUSTON

{() => fs}

Fullstack > Part 13 > Join tables and queries

b Join tables and queries

Application structuring

So far, we have written all the code in the same file. Now let's structure the application a little better. Let's create the following directory structure and files:

```
index.js
util
  config.js
  db.js
models
  index.js
  note.js
controllers
  notes.js
```

The contents of the files are as follows. The file *util/config.js* takes care of handling the environment variables:

```
require('dotenv').config()

module.exports = {
  DATABASE_URL: process.env.DATABASE_URL,
  PORT: process.env.PORT || 3001,
}
```

The role of the file *index.js* is to configure and launch the application:

```
const express = require('express')
const app = express()

const { PORT } = require('./util/config')
const { connectToDatabase } = require('./util/db')

const notesRouter = require('./controllers/notes')

app.use(express.json())

app.use('/api/notes', notesRouter)

const start = async () => {
  await connectToDatabase()
  app.listen(PORT, () => {
    console.log(`Server running on port ${PORT}`)
  })
}

start()
```

Starting the application is slightly different from what we have seen before, because we want to make sure that the database connection is established successfully before the actual startup.

The file *util/db.js* contains the code to initialize the database:

```
const Sequelize = require('sequelize')
const { DATABASE_URL } = require('./config')

const sequelize = new Sequelize(DATABASE_URL)

const connectToDatabase = async () => {
  try {
    await sequelize.authenticate()
    console.log('connected to the database')
  } catch (err) {
    console.log('failed to connect to the database')
    return process.exit(1)
  }

  return null
}

module.exports = { connectToDatabase, sequelize }
```

The notes in the model corresponding to the table to be stored are saved in the file *models/note.js*

```
const { Model, DataTypes } = require('sequelize')

const { sequelize } = require('../util/db')

class Note extends Model {}

Note.init({
  id: {
    type: DataTypes.INTEGER,
    primaryKey: true,
    autoIncrement: true
  },
  content: {
    type: DataTypes.TEXT,
    allowNull: false
  },
  important: {
    type: DataTypes.BOOLEAN
  },
  date: {
    type: DataTypes.DATE
  }
}, {
  sequelize,
  underscored: true,
  timestamps: false,
  modelName: 'note'
})

module.exports = Note
```

The file *models/index.js* is almost useless at this point, as there is only one model in the application. When we start adding other models to the application, the file will become more useful because it will eliminate the need to import files defining individual models in the rest of the application.

```
const Note = require('./note')

Note.sync()

module.exports = {
  Note
}
```

The route handling associated with notes can be found in the file *controllers/notes.js*:

```
const router = require('express').Router()
```

```
const { Note } = require('../models')

router.get('/', async (req, res) => {
  const notes = await Note.findAll()
  res.json(notes)
})

router.post('/', async (req, res) => {
  try {
    const note = await Note.create(req.body)
    res.json(note)
  } catch(error) {
    return res.status(400).json({ error })
  }
})

router.get('/:id', async (req, res) => {
  const note = await Note.findByPk(req.params.id)
  if (note) {
    res.json(note)
  } else {
    res.status(404).end()
  }
})

router.delete('/:id', async (req, res) => {
  const note = await Note.findByPk(req.params.id)
  if (note) {
    await note.destroy()
  }
  res.status(204).end()
})

router.put('/:id', async (req, res) => {
  const note = await Note.findByPk(req.params.id)
  if (note) {
    note.important = req.body.important
    await note.save()
    res.json(note)
  } else {
    res.status(404).end()
  }
})

module.exports = router
```

The structure of the application is good now. However, we note that the route handlers that handle a single note contain a bit of repetitive code, as all of them begin with the line that searches for the note to be handled:

```
const note = await Note.findByPk(req.params.id)
```

copy

Let's refactor this into our own *middleware* and implement it in the route handlers:

```
const noteFinder = async (req, res, next) => {
  req.note = await Note.findByPk(req.params.id)
  next()
}

router.get('/:id', noteFinder, async (req, res) => {
  if (req.note) {
    res.json(req.note)
  } else {
    res.status(404).end()
  }
})

router.delete('/:id', noteFinder, async (req, res) => {
  if (req.note) {
    await req.note.destroy()
  }
  res.status(204).end()
})

router.put('/:id', noteFinder, async (req, res) => {
  if (req.note) {
    req.note.important = req.body.important
    await req.note.save()
    res.json(req.note)
  } else {
    res.status(404).end()
  }
})
```

The route handlers now receive *three* parameters, the first being a string defining the route and second being the middleware *noteFinder* we defined earlier, which retrieves the note from the database and places it in the *note* property of the *req* object. A small amount of copypaste is eliminated and we are satisfied!

The current code for the application is in its entirety on GitHub, branch *part13-2*.

Exercises 13.5.-13.7.

Exercise 13.5.

Change the structure of your application to match the example above, or to follow some other similar clear convention.

Exercise 13.6.

Also, implement support for changing the number of a blog's likes in the application, i.e. the operation

`PUT /api/blogs/:id` (modifying the like count of a blog)

The updated number of likes will be relayed with the request:

```
{
  likes: 3
}
```

Exercise 13.7.

Centralize the application error handling in middleware as in part 3. You can also enable middleware express-async-errors as we did in part 4.

The data returned in the context of an error message is not very important.

At this point, the situations that require error handling by the application are creating a new blog and changing the number of likes on a blog. Make sure the error handler handles both of these appropriately.

User management

Next, let's add a database table *users* to the application, where the users of the application will be stored. In addition, we will add the ability to create users and token-based login as we implemented in part 4. For simplicity, we will adjust the implementation so that all users will have the same password *secret*.

The model defining users in the file *models/user.js* is straightforward

```
const { Model, DataTypes } = require('sequelize')

const { sequelize } = require('../util/db')

class User extends Model {}

User.init({
  id: {
    type: DataTypes.INTEGER,
    primaryKey: true,
    autoIncrement: true
  },
  username: {
    type: DataTypes.STRING,
    unique: true,
    allowNull: false
```

```
    },
    name: {
      type: DataTypes.STRING,
      allowNull: false
    },
  }, {
    sequelize,
    underscored: true,
    timestamps: false,
    modelName: 'user'
  })

  module.exports = User
```

The username field is set to unique. The username could have basically been used as the primary key of the table. However, we decided to create the primary key as a separate field with integer value *id*.

The file *models/index.js* expands slightly:

```
const Note = require('./note')
const User = require('./user')

Note.sync()
User.sync()

module.exports = {
  Note, User
}
```

The route handlers that take care of creating a new user in the *controllers/users.js* file and displaying all users do not contain anything dramatic

```
const router = require('express').Router()

const { User } = require('../models')

router.get('/', async (req, res) => {
  const users = await User.findAll()
  res.json(users)
})

router.post('/', async (req, res) => {
  try {
    const user = await User.create(req.body)
    res.json(user)
  } catch(error) {
    return res.status(400).json({ error })
  }
})
```

```
router.get('/:id', async (req, res) => {
  const user = await User.findByPk(req.params.id)
  if (user) {
    res.json(user)
  } else {
    res.status(404).end()
  }
})

module.exports = router
```

The router handler that handles the login (file *controllers/login.js*) is as follows:

copy

```
const jwt = require('jsonwebtoken')
const router = require('express').Router()

const { SECRET } = require('../util/config')
const User = require('../models/user')

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

  const user = await User.findOne({
    where: {
      username: body.username
    }
  })

  const passwordCorrect = body.password === 'secret'

  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, SECRET)

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

module.exports = router
```

The POST request will be accompanied by a username and a password. First, the object corresponding to the username is retrieved from the database using the *User* model with the findOne method:

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

From the console, we can see that the SQL statement corresponds to the method call

```
SELECT "id", "username", "name"
FROM "users" AS "User"
WHERE "User". "username" = 'mluukkai';
```

If the user is found and the password is correct (i.e. `secret` for all the users), A *jsonwebtoken* containing the user's information is returned in the response. To do this, we install the dependency

```
npm install jsonwebtoken
```

The file *index.js* expands slightly

```
const notesRouter = require('./controllers/notes')
const usersRouter = require('./controllers/users')
const loginRouter = require('./controllers/login')

app.use(express.json())

app.use('/api/notes', notesRouter)
app.use('/api/users', usersRouter)
app.use('/api/login', loginRouter)
```

The current code for the application is in its entirety on GitHub, branch *part13-3*.

Connection between the tables

Users can now be added to the application and users can log in, but this in itself is not a very useful feature yet. We would like to add the features that only a logged-in user can add notes, and that each note is associated with the user who created it. To do this, we need to add a *foreign key* to the *notes* table.

When using Sequelize, a foreign key can be defined by modifying the *models/index.js* file as follows

```
const Note = require('./note')
const User = require('./user')

User.hasMany(Note)
Note.belongsTo(User)
Note.sync({ alter: true })
User.sync({ alter: true })

module.exports = {
  Note, User
}
```

So this is how we define that there is a `one-to-many` relationship connection between the *users* and *notes* entries. We also changed the options of the *sync* calls so that the tables in the database match changes made to the model definitions. The database schema looks like the following from the console:

```
postgres=# \d users
                                    Table "public.users"
  Column  | Type | Collation | Nullable | Default
----------+------------------------+-----------+----------+-----------------------------
-----
 id | integer | not null | nextval('users_id_seq'::regclass)
 username | character varying(255) | | not null |
 name | character varying(255) | | not null |
Indexes:
    "users_pkey" PRIMARY KEY, btree (id)
Referenced by:
    TABLE "notes" CONSTRAINT "notes_user_id_fkey" FOREIGN KEY (user_id) REFERENCES
users(id) ON UPDATE CASCADE ON DELETE SET NULL

postgres=# \d notes
                                     Table "public.notes"
  Column  | Type | Collation | Nullable | Default
-----------+--------------------------+-----------+----------+--------------------------
--------
 id | integer | not null | nextval('notes_id_seq'::regclass)
 content | text | | not null |
 important | boolean | | | |
 date | timestamp with time zone | | | |
 user_id | integer | | | |
Indexes:
    "notes_pkey" PRIMARY KEY, btree (id)
Foreign-key constraints:
    "notes_user_id_fkey" FOREIGN KEY (user_id) REFERENCES users(id) ON UPDATE CASCADE ON
DELETE SET NULL
```

The foreign key *user_id* has been created in the *notes* table, which refers to rows of the *users* table.

Now let's make every insertion of a new note be associated to a user. Before we do the proper implementation (where we associate the note with the logged-in user's token), let's hard code the note to be attached to the first user found in the database:

```
router.post('/', async (req, res) => {
  try {
    const user = await User.findOne()
    const note = await Note.create({...req.body, userId: user.id})
    res.json(note)
  } catch(error) {
    return res.status(400).json({ error })
  }
})
```

Pay attention to how there is now a *user_id* column in the notes at the database level. The corresponding object in each database row is referred to by Sequelize's naming convention as opposed to camel case (*userId*) as typed in the source code.

Making a join query is very easy. Let's change the route that returns all users so that each user's notes are also shown:

```
router.get('/', async (req, res) => {
  const users = await User.findAll({
    include: {
      model: Note
    }
  })
  res.json(users)
})
```

So the join query is done using the `include` option as a query parameter.

The SQL statement generated from the query is seen on the console:

```
SELECT "User". "id", "User". "username", "User". "name", "Notes". "id" AS "Notes.id",
"Notes". "content" AS "Notes.content", "Notes". "important" AS "Notes.important",
"Notes". "date" AS "Notes.date", "Notes". "user_id" AS "Notes.UserId"
FROM "users" AS "User" LEFT OUTER JOIN "notes" AS "Notes" ON "User". "id" = "Notes".
"user_id";
```

The end result is also as you might expect

Proper insertion of notes

Let's change the note insertion by making it work the same as in part 4, i.e. the creation of a note can only be successful if the request corresponding to the creation is accompanied by a valid token from login. The note is then stored in the list of notes created by the user identified by the token:

```
const tokenExtractor = (req, res, next) => {
  const authorization = req.get('authorization')
  if (authorization && authorization.toLowerCase().startsWith('bearer ')) {
    try {
      req.decodedToken = jwt.verify(authorization.substring(7), SECRET)
    } catch{
      return res.status(401).json({ error: 'token invalid' })
    }
  }  else {
    return res.status(401).json({ error: 'token missing' })
  }
  next()
}

router.post('/', tokenExtractor, async (req, res) => {
  try {
    const user = await User.findByPk(req.decodedToken.id)
    const note = await Note.create({...req.body, userId: user.id, date: new Date()})
    res.json(note)
  } catch(error) {
    return res.status(400).json({ error })
```

```
  }
})
```

The token is retrieved from the request headers, decoded and placed in the *req* object by the *tokenExtractor* middleware. When creating a note, a *date* field is also given indicating the time it was created.

Fine-tuning

Our backend currently works almost the same way as the Part 4 version of the same application, except for error handling. Before we make a few extensions to backend, let's change the routes for retrieving all notes and all users slightly.

We will add to each note information about the user who added it:

```
router.get('/', async (req, res) => {
  const notes = await Note.findAll({
    attributes: { exclude: ['userId'] },
    include: {
      model: User,
      attributes: ['name']
    }
  })
  res.json(notes)
})
```

We have also restricted the values of which fields we want. For each note, we return all fields including the *name* of the user associated with the note but excluding the *userId*.

Let's make a similar change to the route that retrieves all users, removing the unnecessary field *userId* from the notes associated with the user:

```
router.get('/', async (req, res) => {
  const users = await User.findAll({
    include: {
      model: Note,
      attributes: { exclude: ['userId'] }
    }
  })
  res.json(users)
})
```

The current code for the application is in its entirety on GitHub, branch *part13-4*.

Attention to the definition of the models

The most perceptive will have noticed that despite the added column *user_id*, we did not make a change to the model that defines notes, but we can still add a user to note objects:

```
const user = await User.findByPk(req.decodedToken.id)
const note = await Note.create({ ...req.body, userId: user.id, date: new Date() })
```

copy

The reason for this is that we specified in the file *models/index.js* that there is a one-to-many connection between users and notes:

```
const Note = require('./note')
const User = require('./user')

User.hasMany(Note)
Note.belongsTo(User)

// ...
```

copy

Sequelize will automatically create an attribute called *userId* on the *Note* model to which, when referenced gives access to the database column *user_id*.

Keep in mind, that we could also create a note as follows using the `build` method:

```
const user = await User.findByPk(req.decodedToken.id)

// create a note without saving it yet
const note = Note.build({ ...req.body, date: new Date() })
 // put the user id in the userId property of the created note
note.userId = user.id
// store the note object in the database
await note.save()
```

copy

This is how we explicitly see that *userId* is an attribute of the notes object.

We could define the model as follows to get the same result:

```
Note.init({
  id: {
    type: DataTypes.INTEGER,
    primaryKey: true,
    autoIncrement: true
  },
```

copy

```
    content: {
      type: DataTypes.TEXT,
      allowNull: false
    },
    important: {
      type: DataTypes.BOOLEAN
    },
    date: {
      type: DataTypes.DATE
    },
    userId: {
      type: DataTypes.INTEGER,
      allowNull: false,
      references: { model: 'users', key: 'id' },
    }
  }, {
    sequelize,
    underscored: true,
    timestamps: false,
    modelName: 'note'
  })

  module.exports = Note
```

Defining at the class level of the model as above is usually unnecessary

```
User.hasMany(Note)
Note.belongsTo(User)
```

copy

Instead we can achieve the same with this. Using one of the two methods is necessary otherwise Sequelize does not know how at the code level to connect the tables to each other.

Exercises 13.8.-13.12.

Exercise 13.8.

Add support for users to the application. In addition to ID, users have the following fields:

- name (string, must not be empty)
- username (string, must not be empty)

Unlike in the material, do not prevent Sequelize from creating timestamps *created_at* and *updated_at* for users

All users can have the same password as the material. You can also choose to properly implement passwords as in part 4.

Implement the following routes

- `POST api/users` (adding a new user)
- `GET api/users` (listing all users)
- `PUT api/users/:username` (changing a username, keep in mind that the parameter is not id but username)

Make sure that the timestamps *created_at* and *updated_at* automatically set by Sequelize work correctly when creating a new user and changing a username.

Exercise 13.9.

Sequelize provides a set of pre-defined validations for the model fields, which it performs before storing the objects in the database.

It's decided to change the user creation policy so that only a valid email address is valid as a username. Implement validation that verifies this issue during the creation of a user.

Modify the error handling middleware to provide a more descriptive error message of the situation (for example, using the Sequelize error message), e.g.

```
{
    "error": [
        "Validation isEmail on username failed"
    ]
}
```

Exercise 13.10.

Expand the application so that the current logged-in user identified by a token is linked to each blog added. To do this you will also need to implement a login endpoint `POST /api/login`, which returns the token.

Exercise 13.11.

Make deletion of a blog only possible for the user who added the blog.

Exercise 13.12.

Modify the routes for retrieving all blogs and all users so that each blog shows the user who added it and each user shows the blogs they have added.

More queries

So far our application has been very simple in terms of queries, queries have searched for either a single row based on the primary key using the method `findByPk` or they have searched for all rows in the table using the method `findAll`. These are sufficient for the frontend of the application made in Section 5, but let's expand the backend so that we can also practice making slightly more complex queries.

Let's first implement the possibility to retrieve only important or non-important notes. Let's implement this using the `query-parameter` `important`:

```
router.get('/', async (req, res) => {
  const notes = await Note.findAll({
    attributes: { exclude: ['userId'] },
    include: {
      model: User,
      attributes: ['name']
    },
    where: {
      important: req.query.important === "true"
    }
  })
  res.json(notes)
})
```

Now the backend can retrieve important notes with a request to `http://localhost:3001/api/notes?important=true` and non-important notes with a request to `http://localhost:3001/api/notes?important=false`

The SQL query generated by Sequelize contains a WHERE clause that filters rows that would normally be returned:

```
SELECT "note". "id", "note". "content", "note". "important", "note". "date", "user". "id"
AS "user.id", "user". "name" AS "user.name"
FROM "notes" AS "note" LEFT OUTER JOIN "users" AS "user" ON "note". "user_id" = "user".
"id"
WHERE "note". "important" = true;
```

Unfortunately, this implementation will not work if the request is not interested in whether the note is important or not, i.e. if the request is made to `http://localhost:3001/api/notes`. The correction can be done in several ways. One, but perhaps not the best way to do the correction would be as follows:

```
const { Op } = require('sequelize')

router.get('/', async (req, res) => {
  let important = {
```

```
    [Op.in]: [true, false]
  }
  if ( req.query.important ) {
    important = req.query.important === "true"
  }

  const notes = await Note.findAll({
    attributes: { exclude: ['userId'] },
    include: {
      model: User,
      attributes: ['name']
    },
    where: {
      important
    }
  })
  res.json(notes)
})
```

The *important* object now stores the query condition. The default query is

```
where: {
  important: {
    [Op.in]: [true, false]
  }
}
```

i.e. the *important* column can be *true* or *false*, using one of the many Sequelize operators `Op.in`. If the query parameter *req.query.important* is specified, the query changes to one of the two forms

```
where: {
  important: true
}
```

or

```
where: {
  important: false
}
```

depending on the value of the query parameter.

The database might now contain some note rows that do not have the value for the column *important* set. After the above changes, these notes can not be found with the queries. Let us set the missing values in the psql console and change the schema so that the column does not allow a null value:

```
Note.init(
  {
    id: {
      type: DataTypes.INTEGER,
      primaryKey: true,
      autoIncrement: true,
    },
    content: {
      type: DataTypes.TEXT,
      allowNull: false,
    },
    important: {
      type: DataTypes.BOOLEAN,
      allowNull: false,
    },
    date: {
      type: DataTypes.DATE,
    },
  },
  // ...
)
```

The functionality can be further expanded by allowing the user to specify a required keyword when retrieving notes, e.g. a request to `http://localhost:3001/api/notes?search=database` will return all notes mentioning *database* or a request to `http://localhost:3001/api/notes?search=javascript&important=true` will return all notes marked as important and mentioning *javascript*. The implementation is as follows

```
router.get('/', async (req, res) => {
  let important = {
    [Op.in]: [true, false]
  }

  if ( req.query.important ) {
    important = req.query.important === "true"
  }

  const notes = await Note.findAll({
    attributes: { exclude: ['userId'] },
    include: {
      model: User,
      attributes: ['name']
    },
    where: {
      important,
      content: {
        [Op.substring]: req.query.search ? req.query.search : ''
      }
    }
  })
```

```
  res.json(notes)
})
```

Sequelize's `Op.substring` generates the query we want using the LIKE keyword in SQL. For example, if we make a query to `http://localhost:3001/api/notes?search=database&important=true` we will see that the SQL query it generates is exactly as we expect.

```
SELECT "note". "id", "note". "content", "note". "important", "note". "date", "user". "id"
AS "user.id", "user". "name" AS "user.name"
FROM "notes" AS "note" LEFT OUTER JOIN "users" AS "user" ON "note". "user_id" = "user".
"id"
WHERE "note". "important" = true AND "note". "content" LIKE '%database%';
```

There is still a beautiful flaw in our application that we see if we make a request to `http://localhost:3001/api/notes`, i.e. we want all the notes, our implementation will cause an unnecessary WHERE in the query, which may (depending on the implementation of the database engine) unnecessarily affect the query efficiency:

```
SELECT "note". "id", "note". "content", "note". "important", "note". "date", "user". "id"
AS "user.id", "user". "name" AS "user.name"
FROM "notes" AS "note" LEFT OUTER JOIN "users" AS "user" ON "note". "user_id" = "user".
"id"
WHERE "note". "important" IN (true, false) AND "note". "content" LIKE '%%';
```

Let's optimize the code so that the WHERE conditions are used only if necessary:

```
router.get('/', async (req, res) => {
  const where = {}

  if (req.query.important) {
    where.important = req.query.important === "true"
  }

  if (req.query.search) {
    where.content = {
      [Op.substring]: req.query.search
    }
  }

  const notes = await Note.findAll({
    attributes: { exclude: ['userId'] },
    include: {
      model: User,
      attributes: ['name']
    },
```

```
    where
  })

  res.json(notes)
})
```

If the request has search conditions e.g. http://localhost:3001/api/notes?search=database&important=true, a query containing WHERE is formed

```
SELECT "note". "id", "note". "content", "note". "important", "note". "date", "user". "id"
AS "user.id", "user". "name" AS "user.name"
FROM "notes" AS "note" LEFT OUTER JOIN "users" AS "user" ON "note". "user_id" = "user".
"id"
WHERE "note". "important" = true AND "note". "content" LIKE '%database%';
```

If the request has no search conditions http://localhost:3001/api/notes, then the query does not have an unnecessary WHERE

```
SELECT "note". "id", "note". "content", "note". "important", "note". "date", "user". "id"
AS "user.id", "user". "name" AS "user.name"
FROM "notes" AS "note" LEFT OUTER JOIN "users" AS "user" ON "note". "user_id" = "user".
"id";
```

The current code for the application is in its entirety on GitHub, branch *part13-5*.

Exercises 13.13.-13.16

Exercise 13.13.

Implement filtering by keyword in the application for the route returning all blogs. The filtering should work as follows

- `GET /api/blogs?search=react` returns all blogs with the search word *react* in the *title* field, the search word is case-insensitive
- `GET /api/blogs` returns all blogs

This should be useful for this task and the next one.

Exercise 13.14.

Expand the filter to search for a keyword in either the *title* or *author* fields, i.e.

`GET /api/blogs?search=jami` returns blogs with the search word *jami* in the *title* field or in the *author* field

Exercise 13.15.

Modify the blogs route so that it returns blogs based on likes in descending order. Search the documentation for instructions on ordering,

Exercise 13.16.

Make a route for the application `/api/authors` that returns the number of blogs for each author and the total number of likes. Implement the operation directly at the database level. You will most likely need the group by functionality, and the sequelize.fn aggregator function.

The JSON returned by the route might look like the following, for example:

```
[
  {
    author: "Jami Kousa",
    articles: "3",
    likes: "10"
  },
  {
    author: "Kalle Ilves",
    articles: "1",
    likes: "2"
  },
  {
    author: "Dan Abramov",
    articles: "1",
    likes: "4"
  }
]
```

Bonus task: order the data returned based on the number of likes, do the ordering in the database query.

Propose changes to material

Part 13a
Previous part

Part 13c
Next part

About course

Course contents

FAQ

Partners

Challenge


HOUSTON

{() => fs}

Fullstack > Part 13 > Migrations, many-to-many relationships

c Migrations, many-to-many relationships

Migrations

Let's keep expanding the backend. We want to implement support for allowing users with *admin status* to put users of their choice in disabled mode, preventing them from logging in and creating new notes. In order to implement this, we need to add boolean fields to the users' database table indicating whether the user is an admin and whether the user is disabled.

We could proceed as before, i.e. change the model that defines the table and rely on Sequelize to synchronize the changes to the database. This is specified by these lines in the file *models/index.js*

```
const Note = require('./note')
const User = require('./user')

Note.belongsTo(User)
User.hasMany(Note)

Note.sync({ alter: true })
User.sync({ alter: true })

module.exports = {
  Note, User
}
```

copy

However, this approach does not make sense in the long run. Let's remove the lines that do the synchronization and move to using a much more robust way, migrations provided by Sequelize (and

many other libraries).

In practice, a migration is a single JavaScript file that describes some modification to a database. A separate migration file is created for each single or multiple changes at once. Sequelize keeps a record of which migrations have been performed, i.e. which changes caused by the migrations are synchronized to the database schema. When creating new migrations, Sequelize keeps up to date on which changes to the database schema are yet to be made. In this way, changes are made in a controlled manner, with the program code stored in version control.

First, let's create a migration that initializes the database. The code for the migration is as follows

```
const { DataTypes } = require('sequelize')

module.exports = {
  up: async ({ context: queryInterface }) => {
    await queryInterface.createTable('notes', {
      id: {
        type: DataTypes.INTEGER,
        primaryKey: true,
        autoIncrement: true
      },
      content: {
        type: DataTypes.TEXT,
        allowNull: false
      },
      important: {
        type: DataTypes.BOOLEAN,
        allowNull: false
      },
      date: {
        type: DataTypes.DATE
      },
    })
    await queryInterface.createTable('users', {
      id: {
        type: DataTypes.INTEGER,
        primaryKey: true,
        autoIncrement: true
      },
      username: {
        type: DataTypes.STRING,
        unique: true,
        allowNull: false
      },
      name: {
        type: DataTypes.STRING,
        allowNull: false
      },
    })
    await queryInterface.addColumn('notes', 'user_id', {
      type: DataTypes.INTEGER,
      allowNull: false,
```

```
      references: { model: 'users', key: 'id' },
    })
  },
  down: async ({ context: queryInterface }) => {
    await queryInterface.dropTable('notes')
    await queryInterface.dropTable('users')
  },
}
```

The migration file defines the functions *up* and *down*, the first of which defines how the database should be modified when the migration is performed. The function *down* tells you how to undo the migration if there is a need to do so.

Our migration contains three operations, the first creates a *notes* table, the second creates a *users* table and the third adds a foreign key to the *notes* table referencing the creator of the note. Changes in the schema are defined by calling the queryInterface object methods.

When defining migrations, it is essential to remember that unlike models, column and table names are written in snake case form:

```
await queryInterface.addColumn('notes', 'user_id', {
  type: DataTypes.INTEGER,
  allowNull: false,
  references: { model: 'users', key: 'id' },
})
```

copy

So in migrations, the names of the tables and columns are written exactly as they appear in the database, while models use Sequelize's default camelCase naming convention.

Save the migration code in the file *migrations/20211209_00_initialize_notes_and_users.js*. Migration file names should always be named alphabetically when created so that previous changes are always before newer changes. One good way to achieve this order is to start the migration file name with the date and a sequence number.

We could run the migrations from the command line using the Sequelize command line tool. However, we choose to perform the migrations manually from the program code using the Umzug library. Let's install the library

```
npm install umzug
```

copy

Let's change the file *util/db.js* that handles the connection to the database as follows:

```
const Sequelize = require('sequelize')
const { DATABASE_URL } = require('./config')
const { Umzug, SequelizeStorage } = require('umzug')
```

copy

```
const sequelize = new Sequelize(DATABASE_URL)

const runMigrations = async () => {
  const migrator = new Umzug({
    migrations: {
      glob: 'migrations/*.js',
    },
    storage: new SequelizeStorage({ sequelize, tableName: 'migrations' }),
    context: sequelize.getQueryInterface(),
    logger: console,
  })

  const migrations = await migrator.up()
  console.log('Migrations up to date', {
    files: migrations.map((mig) => mig.name),
  })
}

const connectToDatabase = async () => {
  try {
    await sequelize.authenticate()
    await runMigrations()
    console.log('connected to the database')
  } catch (err) {
    console.log('failed to connect to the database')
    console.log(err)
    return process.exit(1)
  }

  return null
}

module.exports = { connectToDatabase, sequelize }
```

The *runMigrations* function that performs migrations is now executed every time the application opens a database connection when it starts. Sequelize keeps track of which migrations have already been completed, so if there are no new migrations, running the *runMigrations* function does nothing.

Now let's start with a clean slate and remove all existing database tables from the application:

```
username => drop table notes;
username => drop table users;
username => \d
Did not find any relations.
```

Let's start up the application. A message about the migrations status is printed on the log

```
INSERT INTO "migrations" ("name") VALUES ($1) RETURNING "name";
Migrations up to date { files: [ '20211209_00_initialize_notes_and_users.js' ] }
```

```
database connected
```

If we restart the application, the log also shows that the migration was not repeated.

The database schema of the application now looks like this

```
postgres=# \d
                List of relations
 Schema |     Name     |   Type   |     Owner
--------+--------------+----------+----------------
 public | migrations   | table    | username
 public | notes        | table    | username
 public | notes_id_seq | sequence | username
 public | users        | table    | username
 public | users_id_seq | sequence | username
```

So Sequelize has created a *migrations* table that allows it to keep track of the migrations that have been performed. The contents of the table look as follows:

```
postgres=# select * from migrations;
                   name
-------------------------------------------
 20211209_00_initialize_notes_and_users.js
```

Let's create a few users in the database, as well as a set of notes, and after that we are ready to expand the application.

The current code for the application is in its entirety on GitHub, branch *part13-6*.

Admin user and user disabling

So we want to add two boolean fields to the *users* table

- `admin` tells you whether the user is an admin
- `disabled` tells you whether the user is disabled from actions

Let's create the migration that modifies the database in the file *migrations/20211209_01_admin_and_disabled_to_users.js*:

```
const { DataTypes } = require('sequelize')

module.exports = {
  up: async ({ context: queryInterface }) => {
    await queryInterface.addColumn('users', 'admin', {
      type: DataTypes.BOOLEAN,
```

```
      defaultValue: false
    })
    await queryInterface.addColumn('users', 'disabled', {
      type: DataTypes.BOOLEAN,
      defaultValue: false
    })
  },
  down: async ({ context: queryInterface }) => {
    await queryInterface.removeColumn('users', 'admin')
    await queryInterface.removeColumn('users', 'disabled')
  },
}
```

Make corresponding changes to the model corresponding to the *users* table:

```
User.init({
  id: {
    type: DataTypes.INTEGER,
    primaryKey: true,
    autoIncrement: true
  },
  username: {
    type: DataTypes.STRING,
    unique: true,
    allowNull: false
  },
  name: {
    type: DataTypes.STRING,
    allowNull: false
  },
  admin: {
    type: DataTypes.BOOLEAN,
    defaultValue: false
  },
  disabled: {
    type: DataTypes.BOOLEAN,
    defaultValue: false
  },
}, {
  sequelize,
  underscored: true,
  timestamps: false,
  modelName: 'user'
})
```

When the new migration is performed when the code restarts, the schema is changed as desired:

```
username-> \d users
                                  Table "public.users"
  Column  |          Type          | Collation | Nullable |              Default
```

```
----------+------------------------+-----------+----------+-------------------------------
-----
 id       | integer                |           | not null |
nextval('users_id_seq'::regclass)
 username | character varying(255) |           | not null |
 name     | character varying(255) |           | not null |
 admin    | boolean                |           |          |
 disabled | boolean                |           |          |
Indexes:
    "users_pkey" PRIMARY KEY, btree (id)
    "users_username_key" UNIQUE CONSTRAINT, btree (username)
Referenced by:
    TABLE "notes" CONSTRAINT "notes_user_id_fkey" FOREIGN KEY (user_id) REFERENCES
users(id)
```

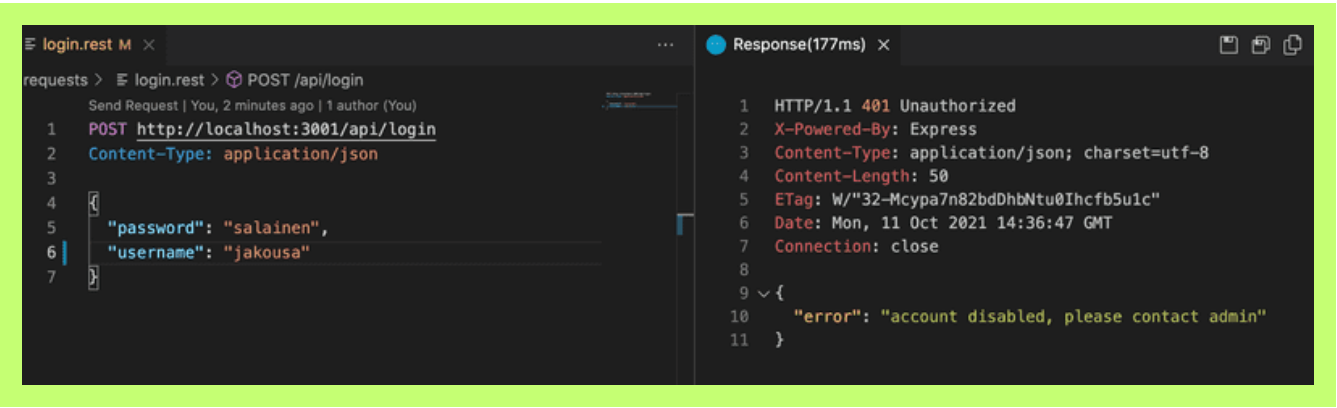
Now let's expand the controllers as follows. We prevent logging in if the user field *disabled* is set to *true*:

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

  const user = await User.findOne({
    where: {
      username: body.username
    }
  })

  const passwordCorrect = body.password === 'secret'

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

  if (user.disabled) {
    return response.status(401).json({
      error: 'account disabled, please contact admin'
    })
  }

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

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

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

Let's disable the user *jakousa* using his ID:

```
username => update users set disabled=true where id=3;
UPDATE 1
username => update users set admin=true where id=1;
UPDATE 1
username => select * from users;
 id | username |       name       | admin | disabled
----+----------+------------------+-------+----------
  2 | lynx     | Kalle Ilves      |       |
  3 | jakousa  | Jami Kousa       | f     | t
  1 | mluukkai | Matti Luukkainen | t     |
```

And make sure that logging in is no longer possible



Let's create a route that will allow an admin to change the status of a user's account:

```
const isAdmin = async (req, res, next) => {
  const user = await User.findByPk(req.decodedToken.id)
  if (!user.admin) {
    return res.status(401).json({ error: 'operation not allowed' })
  }
  next()
}

router.put('/:username', tokenExtractor, isAdmin, async (req, res) => {
  const user = await User.findOne({
    where: {
      username: req.params.username
    }
  })

  if (user) {
    user.disabled = req.body.disabled
    await user.save()
    res.json(user)
  } else {
```

```
    res.status(404).end()
  }
})
```

There are two middleware used, the first called *tokenExtractor* is the same as the one used by the note-creation route, i.e. it places the decoded token in the *decodedToken* field of the request-object. The second middleware *isAdmin* checks whether the user is an admin and if not, the request status is set to 401 and an appropriate error message is returned.

Note how *two middleware* are chained to the route, both of which are executed before the actual route handler. It is possible to chain an arbitrary number of middleware to a request.

The middleware *tokenExtractor* is now moved to *util/middleware.js* as it is used from multiple locations.

```
const jwt = require('jsonwebtoken')
const { SECRET } = require('./config.js')

const tokenExtractor = (req, res, next) => {
  const authorization = req.get('authorization')
  if (authorization && authorization.toLowerCase().startsWith('bearer ')) {
    try {
      req.decodedToken = jwt.verify(authorization.substring(7), SECRET)
    } catch{
      return res.status(401).json({ error: 'token invalid' })
    }
  } else {
    return res.status(401).json({ error: 'token missing' })
  }
  next()
}

module.exports = { tokenExtractor }
```

An admin can now re-enable the user *jakousa* by making a PUT request to `/api/users/jakousa`, where the request comes with the following data:

```
{
    "disabled": false
}
```

As noted in the end of Part 4, the way we implement disabling users here is problematic. Whether or not the user is disabled is only checked at `login`, if the user has a token at the time the user is disabled, the user may continue to use the same token, since no lifetime has been set for the token and the disabled status of the user is not checked when creating notes.

Before we proceed, let's make an npm script for the application, which allows us to undo the previous migration. After all, not everything always goes right the first time when developing migrations.

Let's modify the file *util/db.js* as follows:

```
const Sequelize = require('sequelize')
const { DATABASE_URL } = require('./config')
const { Umzug, SequelizeStorage } = require('umzug')

const sequelize = new Sequelize(DATABASE_URL, {
  dialectOptions: {
    ssl: {
      require: true,
      rejectUnauthorized: false
    }
  },
});

const connectToDatabase = async () => {
  try {
    await sequelize.authenticate()
    await runMigrations()
    console.log('connected to the database')
  } catch (err) {
    console.log('failed to connect to the database')
    return process.exit(1)
  }

  return null
}

const migrationConf = {
  migrations: {
    glob: 'migrations/*.js',
  },
  storage: new SequelizeStorage({ sequelize, tableName: 'migrations' }),
  context: sequelize.getQueryInterface(),
  logger: console,
}

const runMigrations = async () => {
  const migrator = new Umzug(migrationConf)
  const migrations = await migrator.up()
  console.log('Migrations up to date', {
    files: migrations.map((mig) => mig.name),
  })
}
const rollbackMigration = async () => {
  await sequelize.authenticate()
  const migrator = new Umzug(migrationConf)
  await migrator.down()
}

module.exports = { connectToDatabase, sequelize, rollbackMigration }
```

Let's create a file *util/rollback.js*, which will allow the npm script to execute the specified migration rollback function:

```
const { rollbackMigration } = require('./db')

rollbackMigration()
```

and the script itself:

```
{
  "scripts": {
    "dev": "nodemon index.js",
    "migration:down": "node util/rollback.js"
  },
}
```

So we can now undo the previous migration by running `npm run migration:down` from the command line.

Migrations are currently executed automatically when the program is started. In the development phase of the program, it might sometimes be more appropriate to disable the automatic execution of migrations and make migrations manually from the command line.

The current code for the application is in its entirety on GitHub, branch *part13-7*.

Exercises 13.17-13.18.

Exercise 13.17.

Delete all tables from your application's database.

Make a migration that initializes the database. Add *created_at* and *updated_at* timestamps for both tables. Keep in mind that you will have to add them in the migration yourself.

NOTE: be sure to remove the commands *User.sync()* and *Blog.sync()*, which synchronizes the models' schemas from your code, otherwise your migrations will fail.

NOTE2: if you have to delete tables from the command line (i.e. you don't do the deletion by undoing the migration), you will have to delete the contents of the *migrations* table if you want your program to perform the migrations again.

Exercise 13.18.

Expand your application (by migration) so that the blogs have a year written attribute, i.e. a field *year* which is an integer at least equal to 1991 but not greater than the current year. Make sure the application gives an appropriate error message if an incorrect value is attempted to be given for a year written.

Many-to-many relationships

We will continue to expand the application so that each user can be added to one or more *teams*.

Since an arbitrary number of users can join one team, and one user can join an arbitrary number of teams, we are dealing with a many-to-many relationship, which is traditionally implemented in relational databases using a *connection table*.

Let's now create the code needed for the teams table as well as the connection table. The migration (saved in file *20211209_02_add_teams_and_memberships.js*) is as follows:

```
const { DataTypes } = require('sequelize')

module.exports = {
  up: async ({ context: queryInterface }) => {
    await queryInterface.createTable('teams', {
      id: {
        type: DataTypes.INTEGER,
        primaryKey: true,
        autoIncrement: true
      },
      name: {
        type: DataTypes.TEXT,
        allowNull: false,
        unique: true
      },
    })
    await queryInterface.createTable('memberships', {
      id: {
        type: DataTypes.INTEGER,
        primaryKey: true,
        autoIncrement: true
      },
      user_id: {
        type: DataTypes.INTEGER,
        allowNull: false,
        references: { model: 'users', key: 'id' },
      },
      team_id: {
        type: DataTypes.INTEGER,
        allowNull: false,
        references: { model: 'teams', key: 'id' },
      },
```

```
    })
  },
  down: async ({ context: queryInterface }) => {
    await queryInterface.dropTable('teams')
    await queryInterface.dropTable('memberships')
  },
}
```

The models contain almost the same code as the migration. The team model in *models/team.js*:

```
const { Model, DataTypes } = require('sequelize')

const { sequelize } = require('../util/db')

class Team extends Model {}

Team.init({
  id: {
    type: DataTypes.INTEGER,
    primaryKey: true,
    autoIncrement: true
  },
  name: {
    type: DataTypes.TEXT,
    allowNull: false,
    unique: true
  },
}, {
  sequelize,
  underscored: true,
  timestamps: false,
  modelName: 'team'
})

module.exports = Team
```

The model for the connection table in *models/membership.js*:

```
const { Model, DataTypes } = require('sequelize')

const { sequelize } = require('../util/db')

class Membership extends Model {}

Membership.init({
  id: {
    type: DataTypes.INTEGER,
    primaryKey: true,
    autoIncrement: true
  },
```

```
    userId: {
      type: DataTypes.INTEGER,
      allowNull: false,
      references: { model: 'users', key: 'id' },
    },
    teamId: {
      type: DataTypes.INTEGER,
      allowNull: false,
      references: { model: 'teams', key: 'id' },
    },
  }, {
    sequelize,
    underscored: true,
    timestamps: false,
    modelName: 'membership'
  })

  module.exports = Membership
```

So we have given the connection table a name that describes it well, *membership*. There is not always a relevant name for a connection table, in which case the name of the connection table can be a combination of the names of the tables that are joined, e.g. *user_teams* could fit our situation.

We make a small addition to the *models/index.js* file to connect teams and users at the code level using the belongsToMany method.

```
const Note = require('./note')
const User = require('./user')
const Team = require('./team')
const Membership = require('./membership')

Note.belongsTo(User)
User.hasMany(Note)

User.belongsToMany(Team, { through: Membership })
Team.belongsToMany(User, { through: Membership })

module.exports = {
  Note, User, Team, Membership
}
```

Note the difference between the migration of the connection table and the model when defining foreign key fields. During the migration, fields are defined in snake case form:

```
await queryInterface.createTable('memberships', {
  // ...
  user_id: {
    type: DataTypes.INTEGER,
    allowNull: false,
    references: { model: 'users', key: 'id' },
```

```
  },
  team_id: {
    type: DataTypes.INTEGER,
    allowNull: false,
    references: { model: 'teams', key: 'id' },
  }
})
```

in the model, the same fields are defined in camel case:

```
Membership.init({
  // ...
  userId: {
    type: DataTypes.INTEGER,
    allowNull: false,
    references: { model: 'users', key: 'id' },
  },
  teamId: {
    type: DataTypes.INTEGER,
    allowNull: false,
    references: { model: 'teams', key: 'id' },
  },
  // ...
})
```

Now let's create a couple of teams from the psql console, as well as a few memberships:

```
insert into teams (name) values ('toska');
insert into teams (name) values ('mosa climbers');
insert into memberships (user_id, team_id) values (1, 1);
insert into memberships (user_id, team_id) values (1, 2);
insert into memberships (user_id, team_id) values (2, 1);
insert into memberships (user_id, team_id) values (3, 2);
```

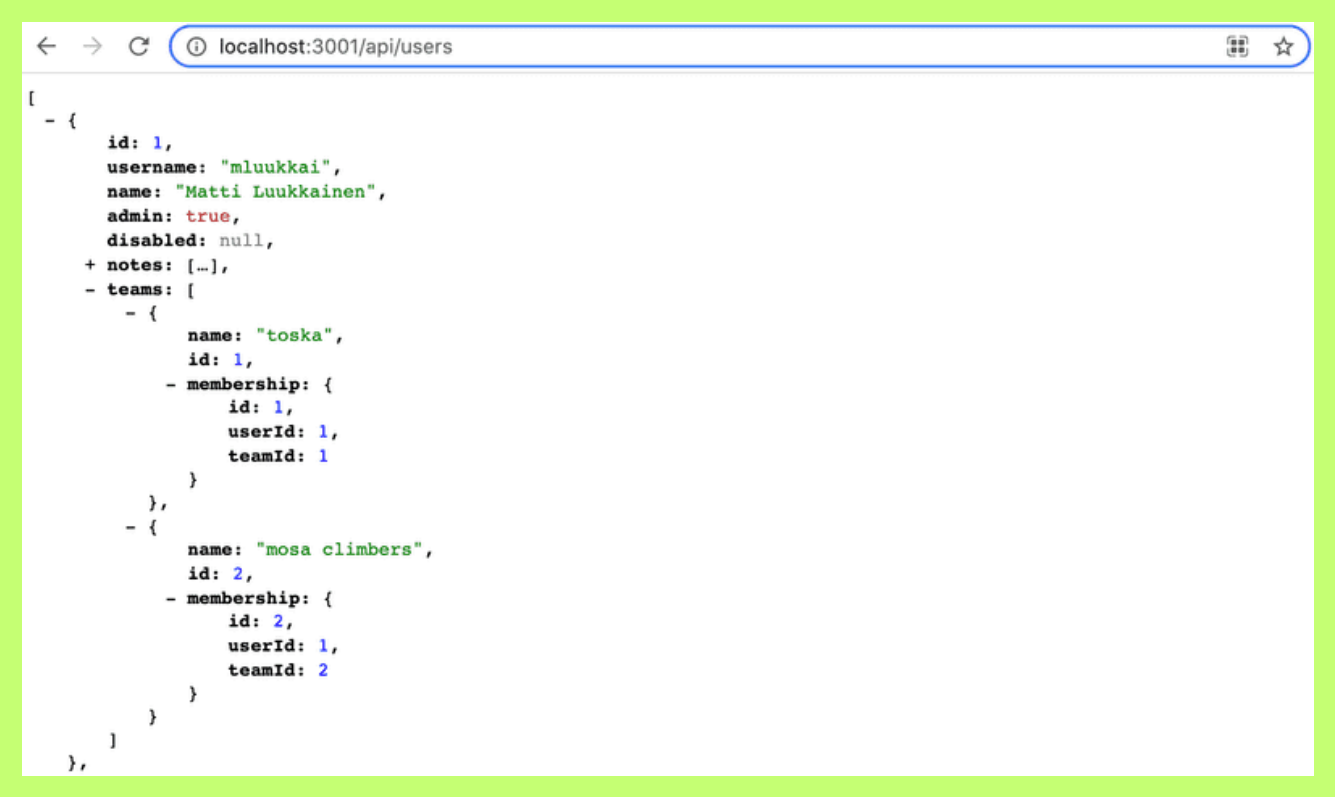
Information about users' teams is then added to route for retrieving all users

```
router.get('/', async (req, res) => {
  const users = await User.findAll({
    include: [
      {
        model: Note,
        attributes: { exclude: ['userId'] }
      },
      {
        model: Team,
        attributes: ['name', 'id'],
      }
```

```
    ]
  })
  res.json(users)
})
```

The most observant will notice that the query printed to the console now combines three tables.

The solution is pretty good, but there's a beautiful flaw in it. The result also comes with the attributes of the corresponding row of the connection table, although we do not want this:



By carefully reading the documentation, you can find a solution:

```
router.get('/', async (req, res) => {
  const users = await User.findAll({
    include: [
      {
        model: Note,
        attributes: { exclude: ['userId'] }
      },
      {
        model: Team,
        attributes: ['name', 'id'],
        through: {
          attributes: []
        }
      }
    ]
  })
```

```
  res.json(users)
})
```

The current code for the application is in its entirety on GitHub, branch *part13-8*.

Note on the properties of Sequelize model objects

The specification of our models is shown by the following lines:

```
User.hasMany(Note)
Note.belongsTo(User)

User.belongsToMany(Team, { through: Membership })
Team.belongsToMany(User, { through: Membership })
```

These allow Sequelize to make queries that retrieve, for example, all the notes of users, or all members of a team.

Thanks to the definitions, we also have direct access to, for example, the user's notes in the code. In the following code, we will search for a user with id 1 and print the notes associated with the user:

```
const user = await User.findByPk(1, {
  include: {
    model: Note
  }
})

user.notes.forEach(note => {
  console.log(note.content)
})
```

The *User.hasMany(Note)* definition therefore attaches a *notes* property to the *user* object, which gives access to the notes made by the user. The *User.belongsToMany(Team, { through: Membership }))* definition similarly attaches a *teams* property to the *user* object, which can also be used in the code:

```
const user = await User.findByPk(1, {
  include: {
    model: team
  }
})

user.teams.forEach(team => {
  console.log(team.name)
})
```

Suppose we would like to return a JSON object from the single user's route containing the user's name, username and number of notes created. We could try the following:

```
router.get('/:id', async (req, res) => {
  const user = await User.findByPk(req.params.id, {
    include: {
        model: Note
      }
    }
  )

  if (user) {
    user.note_count = user.notes.length
    delete user.notes
    res.json(user)

  } else {
    res.status(404).end()
  }
})
```

copy

So, we tried to add the *noteCount* field on the object returned by Sequelize and remove the *notes* field from it. However, this approach does not work, as the objects returned by Sequelize are not normal objects where the addition of new fields works as we intend.

A better solution is to create a completely new object based on the data retrieved from the database:

```
router.get('/:id', async (req, res) => {
  const user = await User.findByPk(req.params.id, {
    include: {
        model: Note
      }
    }
  )

  if (user) {
    res.json({
      username: user.username,
      name: user.name,
      note_count: user.notes.length
    })

  } else {
    res.status(404).end()
  }
})
```

copy

Revisiting many-to-many relationships

Let's make another many-to-many relationship in the application. Each note is associated to the user who created it by a foreign key. It is now decided that the application also supports that the note can be associated with other users, and that a user can be associated with an arbitrary number of notes created by other users. The idea is that these notes are those that the user has *marked* for himself.

Let's make a connection table *user_notes* for the situation. The migration, that is saved in file *20211209_03_add_user_notes.js* is straightforward:

```
const { DataTypes } = require('sequelize')

module.exports = {
  up: async ({ context: queryInterface }) => {
    await queryInterface.createTable('user_notes', {
      id: {
        type: DataTypes.INTEGER,
        primaryKey: true,
        autoIncrement: true
      },
      user_id: {
        type: DataTypes.INTEGER,
        allowNull: false,
        references: { model: 'users', key: 'id' },
      },
      note_id: {
        type: DataTypes.INTEGER,
        allowNull: false,
        references: { model: 'notes', key: 'id' },
      },
    })
  },
  down: async ({ context: queryInterface }) => {
    await queryInterface.dropTable('user_notes')
  },
}
```

Also, there is nothing special about the model:

```
const { Model, DataTypes } = require('sequelize')

const { sequelize } = require('../util/db')

class UserNotes extends Model {}

UserNotes.init({
  id: {
    type: DataTypes.INTEGER,
    primaryKey: true,
```

```
    autoIncrement: true
  },
  userId: {
    type: DataTypes.INTEGER,
    allowNull: false,
    references: { model: 'users', key: 'id' },
  },
  noteId: {
    type: DataTypes.INTEGER,
    allowNull: false,
    references: { model: 'notes', key: 'id' },
  },
}, {
  sequelize,
  underscored: true,
  timestamps: false,
  modelName: 'user_notes'
})

module.exports = UserNotes
```

The file *models/index.js*, on the other hand, comes with a slight change to what we saw before:

```
const Note = require('./note')
const User = require('./user')
const Team = require('./team')
const Membership = require('./membership')
const UserNotes = require('./user_notes')

Note.belongsTo(User)
User.hasMany(Note)

User.belongsToMany(Team, { through: Membership })
Team.belongsToMany(User, { through: Membership })

User.belongsToMany(Note, { through: UserNotes, as: 'marked_notes' })
Note.belongsToMany(User, { through: UserNotes, as: 'users_marked' })

module.exports = {
  Note, User, Team, Membership, UserNotes
}
```

Once again *belongsToMany* is used, which now links users to notes via the *UserNotes* model corresponding to the connection table. However, this time we give an *alias name* for the attribute formed using the keyword `as`, the default name (a user's *notes*) would overlap with its previous meaning, i.e. notes created by the user.

We extend the route for an individual user to return the user's teams, their own notes, and other notes marked by the user:

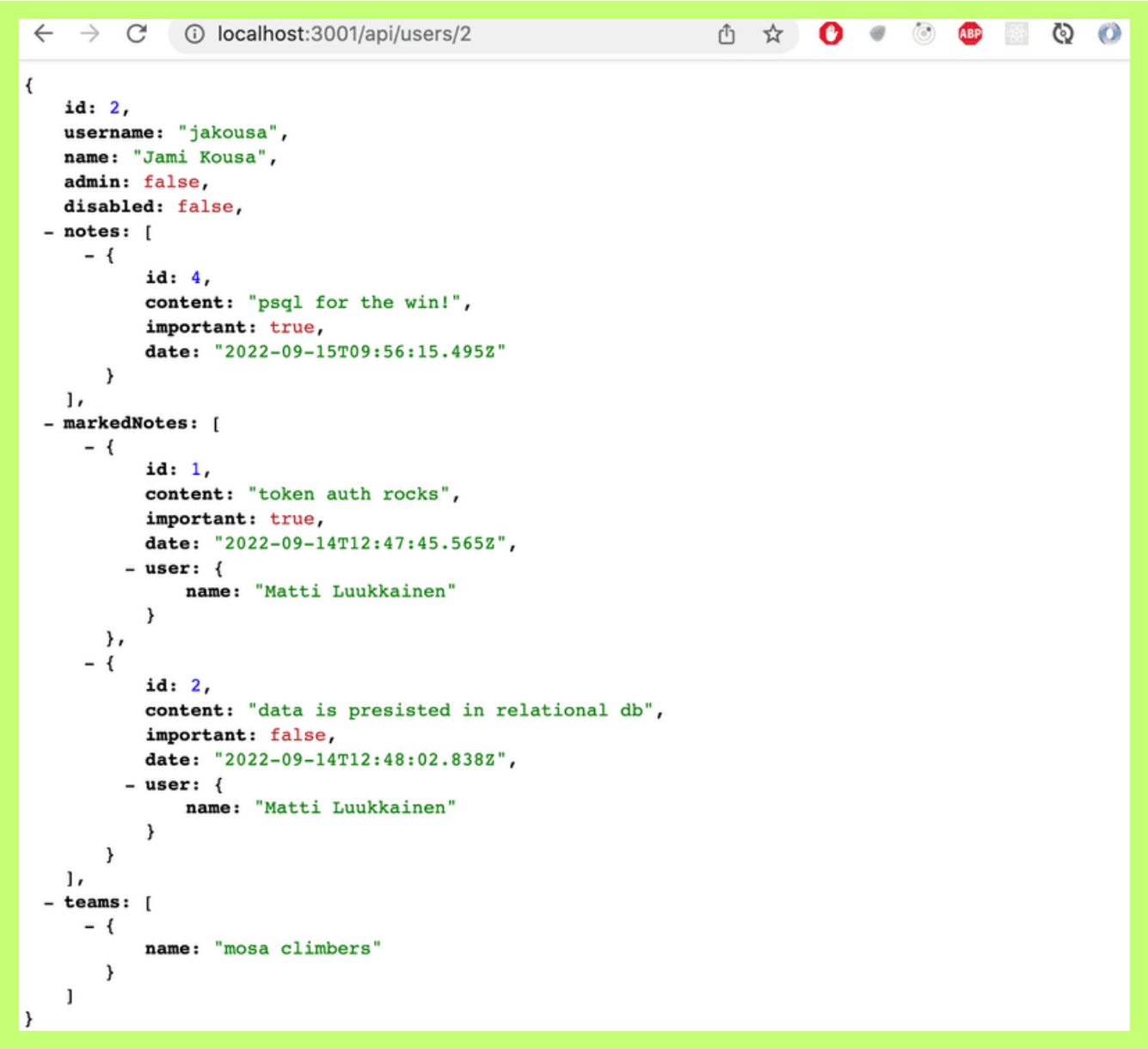
```
router.get('/:id', async (req, res) => {
  const user = await User.findByPk(req.params.id, {
    attributes: { exclude: [''] } ,
    include:[{
        model: Note,
        attributes: { exclude: ['userId'] }
      },
      {
        model: Note,
        as: 'marked_notes',
        attributes: { exclude: ['userId']},
        through: {
          attributes: []
        }
      },
      {
        model: Team,
        attributes: ['name', 'id'],
        through: {
          attributes: []
        }
      },
    ]
  })

  if (user) {
    res.json(user)
  } else {
    res.status(404).end()
  }
})
```

In the context of the include, we must now use the alias name *marked_notes* which we have just defined with the *as* attribute.

In order to test the feature, let's create some test data in the database:

```
insert into user_notes (user_id, note_id) values (1, 4);
insert into user_notes (user_id, note_id) values (1, 5);
```

The end result is functional:

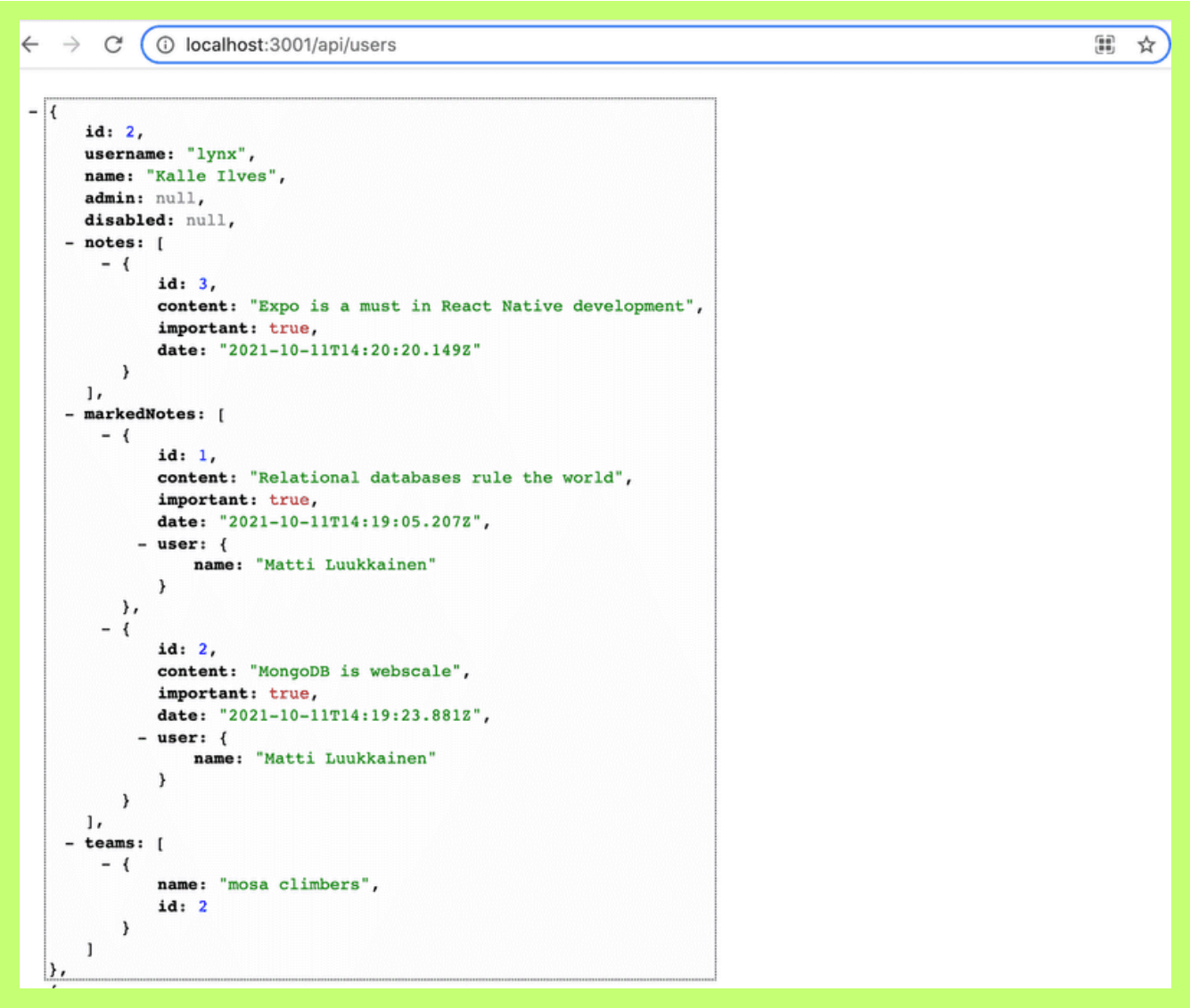


What if we wanted to include information about the author of the note in the notes marked by the user as well? This can be done by adding an *include* to the marked notes:

```
router.get('/:id', async (req, res) => {
  const user = await User.findByPk(req.params.id, {
    attributes: { exclude: [''] } ,
    include:[{
        model: Note,
        attributes: { exclude: ['userId'] }
      },
      {
        model: Note,
        as: 'marked_notes',
        attributes: { exclude: ['userId']},
        through: {
          attributes: []
```

```
        },
        include: {
          model: User,
          attributes: ['name']
        }
      },
      {
        model: Team,
        attributes: ['name', 'id'],
        through: {
          attributes: []
        }
      },
    ]
  })

  if (user) {
    res.json(user)
  } else {
    res.status(404).end()
  }
})
```

The end result is as desired:



The current code for the application is in its entirety on GitHub, branch *part13-9*.

Exercises 13.19.-13.23.

Exercise 13.19.

Give users the ability to add blogs on the system to a *reading list*. When added to the reading list, the blog should be in the *unread* state. The blog can later be marked as *read*. Implement the reading list using a connection table. Make database changes using migrations.

In this task, adding to a reading list and displaying the list need not be successful other than directly using the database.

Exercise 13.20.

Now add functionality to the application to support the reading list.

Adding a blog to the reading list is done by making an HTTP POST to the path */api/readinglists*, the request will be accompanied with the blog and user id:

```
{
  "blogId": 10,
  "userId": 3
}
```

Also modify the individual user route `GET /api/users/:id` to return not only the user's other information but also the reading list, e.g. in the following format:

```
{
  name: "Matti Luukkainen",
  username: "mluukkai@iki.fi",
  readings: [
    {
      id: 3,
      url: "https://google.com",
      title: "Clean React",
      author: "Dan Abramov",
      likes: 34,
      year: null,
    },
    {
      id: 4,
      url: "https://google.com",
      title: "Clean Code",
      author: "Bob Martin",
      likes: 5,
      year: null,
    }
  ]
}
```

At this point, information about whether the blog is read or not does not need to be available.

Exercise 13.21.

Expand the single-user route so that each blog in the reading list shows also whether the blog has been read *and* the id of the corresponding join table row.

For example, the information could be in the following form:

```
{
  name: "Matti Luukkainen",
  username: "mluukkai@iki.fi",
  readings: [
    {
      id: 3,
      url: "https://google.com",
      title: "Clean React",
      author: "Dan Abramov",
      likes: 34,
      year: null,
      readinglists: [
        {
          read: false,
          id: 2
        }
      ]
    },
    {
      id: 4,
      url: "https://google.com",
      title: "Clean Code",
      author: "Bob Martin",
      likes: 5,
      year: null,
      readinglists: [
        {
          read: false,
          id: 3
        }
      ]
    }
  ]
}
```

Note: there are several ways to implement this functionality. This should help.

Note also that despite having an array field *readinglists* in the example, it should always just contain exactly one object, the join table entry that connects the book to the particular user's reading list.

Exercise 13.22.

Implement functionality in the application to mark a blog in the reading list as read. Marking as read is done by making a request to the `PUT /api/readinglists/:id` path, and sending the request with

```
{ "read": true }
```

The user can only mark the blogs in their own reading list as read. The user is identified as usual from the token accompanying the request.

Exercise 13.23.

Modify the route that returns a single user's information so that the request can control which of the blogs in the reading list are returned:

- `GET /api/users/:id` returns the entire reading list
- `GET /api/users/:id?read=true` returns blogs that have been read
- `GET /api/users/:id?read=false` returns blogs that have not been read

Concluding remarks

The state of our application is starting to be at least acceptable. However, before the end of the section, let's look at a few more points.

Eager vs lazy fetch

When we make queries using the *include* attribute:

```
User.findOne({
  include: {
    model: note
  }
})
```

The so-called eager fetch occurs, i.e. all the rows of the tables attached to the user by the join query, in the example the notes made by the user, are fetched from the database at the same time. This is often what we want, but there are also situations where you want to do a so-called `lazy fetch`, e.g. search for user related teams only if they are needed.

Let's now modify the route for an individual user so that it fetches the user's teams only if the query parameter *teams* is set in the request:

```
router.get('/:id', async (req, res) => {
  const user = await User.findByPk(req.params.id, {
    attributes: { exclude: [''] } ,
    include:[{
        model: note,
        attributes: { exclude: ['userId'] }
      },
      {
        model: Note,
        as: 'marked_notes',
```

```
        attributes: { exclude: ['userId']},
        through: {
          attributes: []
        },
        include: {
          model: user,
          attributes: ['name']
        }
      },
    ]
  })

  if (!user) {
    return res.status(404).end()
  }

  let teams = undefined
  if (req.query.teams) {
    teams = await user.getTeams({
      attributes: ['name'],
      joinTableAttributes: []
    })
  }
  res.json({ ...user.toJSON(), teams })
})
```

So now, the *User.findByPk* query does not retrieve teams, but they are retrieved if necessary by the *user* method *getTeams*, which is automatically generated by Sequelize for the model object. Similar *get-* and a few other useful methods are automatically generated when defining associations for tables at the Sequelize level.

Features of models

There are some situations where, by default, we do not want to handle all the rows of a particular table. One such case could be that we don't normally want to display users that have been *disabled* in our application. In such a situation, we could define the default scopes for the model like this:

```
class User extends Model {}

User.init({
  // field definition
}, {
  sequelize,
  underscored: true,
  timestamps: false,
  modelName: 'user',
  defaultScope: {
    where: {
      disabled: false
    }
  },
  scopes: {
```

```
    admin: {
      where: {
        admin: true
      }
    },
    disabled: {
      where: {
        disabled: true
      }
    }
  }
})

module.exports = User
```

Now the query caused by the function call *User.findAll()* has the following WHERE condition:

```
WHERE "user". "disabled" = false;
```

copy

For models, it is possible to define other scopes as well:

```
User.init({
  // field definition
}, {
  sequelize,
  underscored: true,
  timestamps: false,
  modelName: 'user',
  defaultScope: {
    where: {
      disabled: false
    }
  },
  scopes: {
    admin: {
      where: {
        admin: true
      }
    },
    disabled: {
      where: {
        disabled: true
      }
    },
    name(value) {
      return {
        where: {
          name: {
            [Op.iLike]: value
          }
```

copy

```
        }
      }
    },
  }
})
```

Scopes are used as follows:

```
// all admins
const adminUsers = await User.scope('admin').findAll()

// all inactive users
const disabledUsers = await User.scope('disabled').findAll()

// users with the string jami in their name
const jamiUsers = User.scope({ method: ['name', '%jami%'] }).findAll()
```

It is also possible to chain scopes:

```
// admins with the string jami in their name
const jamiUsers = User.scope('admin', { method: ['name', '%jami%'] }).findAll()
```

Since Sequelize models are normal JavaScript classes, it is possible to add new methods to them.

Here are two examples:

```
const { Model, DataTypes, Op } = require('sequelize')

const Note = require('./note')
const { sequelize } = require('../util/db')

class User extends Model {
  async number_of_notes() {
    return (await this.getNotes()).length
  }
  static async with_notes(limit){
    return await User.findAll({
      attributes: {
        include: [[ sequelize.fn("COUNT", sequelize.col("notes.id")), "note_count" ]]
      },
      include: [
        {
          model: Note,
          attributes: []
        },
      ],
      group: ['user.id'],
```

```
      having: sequelize.literal(`COUNT(notes.id) > ${limit}`)
    })
  }
}

User.init({
  // ...
})

module.exports = User
```

The first of the methods *numberOfNotes* is an *instance method*, meaning that it can be called on instances of the model:

```
const jami = await User.findOne({ name: 'Jami Kousa'})
const cnt = await jami.number_of_notes()
console.log(`Jami has created ${cnt} notes`)
```

Within the instance method, the keyword *this* therefore refers to the instance itself:

```
async number_of_notes() {
  return (await this.getNotes()).length
}
```

The second of the methods, which returns those users who have at least *X*, the number specified by the parameter, amount of notes is a *class method*, i.e. it is called directly on the model:

```
const users = await User.with_notes(2)
console.log(JSON.stringify(users, null, 2))
users.forEach(u => {
  console.log(u.name)
})
```

Repeatability of models and migrations

We have noticed that the code for models and migrations is very repetitive. For example, the model of teams

```
class Team extends Model {}

Team.init({
  id: {
    type: DataTypes.INTEGER,
    primaryKey: true,
```

```
    autoIncrement: true
  },
  name: {
    type: DataTypes.TEXT,
    allowNull: false,
    unique: true
  },
}, {
  sequelize,
  underscored: true,
  timestamps: false,
  modelName: 'team'
})

module.exports = Team
```

and migration contain much of the same code

```
const { DataTypes } = require('sequelize')

module.exports = {
  up: async ({ context: queryInterface }) => {
    await queryInterface.createTable('teams', {
      id: {
        type: DataTypes.INTEGER,
        primaryKey: true,
        autoIncrement: true
      },
      name: {
        type: DataTypes.TEXT,
        allowNull: false,
        unique: true
      },
    })
  },
  down: async ({ context: queryInterface }) => {
    await queryInterface.dropTable('teams')
  },
}
```

Couldn't we optimize the code so that, for example, the model exports the shared parts needed for the migration?

However, the problem is that the definition of the model may change over time, for example the *name* field may change or its data type may change. Migrations must be able to be performed successfully at any time from start to end, and if the migrations are relying on the model to have certain content, it may no longer be true in a month or a year's time. Therefore, despite the "copy paste", the migration code should be completely separate from the model code.

One solution would be to use Sequelize's command line tool, which generates both models and migration files based on commands given at the command line. For example, the following command

would create a *User* model with *name*, *username*, and *admin* as attributes, as well as the migration that manages the creation of the database table:

```
npx sequelize-cli model:generate --name User --attributes
name:string,username:string,admin:boolean
```

copy

From the command line, you can also run rollbacks, i.e. undo migrations. The command line documentation is unfortunately incomplete and in this course we decided to do both models and migrations manually. The solution may or may not have been a wise one.

Exercise 13.24.

Exercise 13.24.

Grand finale: towards the end of part 4 there was mention of a token-criticality problem: if a user's access to the system is decided to be revoked, the user may still use the token in possession to use the system.

The usual solution to this is to store a record of each token issued to the client in the backend database, and to check with each request whether access is still valid. In this case, the validity of the token can be removed immediately if necessary. Such a solution is often referred to as a *server-side session*.

Now expand the system so that the user who has lost access will not be able to perform any actions that require login.

You will probably need at least the following for the implementation

- a boolean value column in the user table to indicate whether the user is disabled
  - it is sufficient to disable and enable users directly from the database
- a table that stores active sessions
  - a session is stored in the table when a user logs in, i.e. operation `POST /api/login`
  - the existence (and validity) of the session is always checked when the user makes an operation that requires login
- a route that allows the user to "log out" of the system, i.e. to practically remove active sessions from the database, the route can be e.g. `DELETE /api/logout`

Keep in mind that actions requiring login should not be successful with an "expired token", i.e. with the same token after logging out.

You may also choose to use some purpose-built npm library to handle sessions.

Make the database changes required for this task using migrations.

Submitting exercises and getting the credits

Exercises of this part are submitted just like in the previous parts, but unlike parts 0 to 7, the submission goes to an own course instance. Remember that you have to finish all the exercises to pass this part!

Once you have completed the exercises and want to get the credits, let us know through the exercise submission system that you have completed the course:

Note that you need a registration to the corresponding course part for getting the credits registered, see here for more information.

Propose changes to material

Part 13b

Previous part

About course

Course contents

FAQ

Partners

Challenge


UNIVERSITY OF HELSINKI


HOUSTON