```
{() => fs}
```

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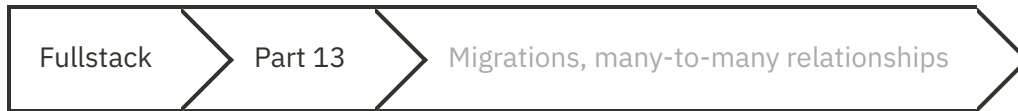
c Migrations, many-to-many relationships

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
}
```

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

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

```javascript
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                                                          copy
                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;                                   copy
                    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')                              copy

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({                                                          copy
  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                                                  copy
                        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) => {                         copy
  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 })
})
```
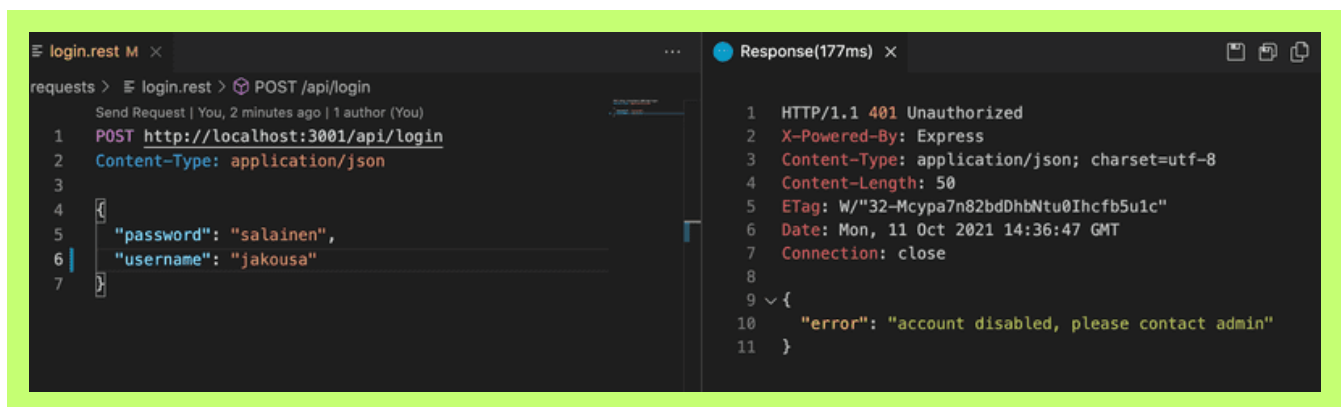
Let's disable the user *jakousa* using his ID:

```
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')                                        copy
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:

```
{                                                                          copy

    "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:

```javascript
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*.

Tasks 13.17-13.18.

Task 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.

Task 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({                                              copy
  // ...
  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');                     copy
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) => {                          copy
  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:

```
←  →  C   ⓘ localhost:3001/api/users                                                      ▦  ☆
[
  - {
      id: 1,
      username: "mluukkai",
      name: "Matti Luukkainen",
      admin: true,
      disabled: null,
    + notes: […],
    - teams: [
        - {
            name: "toska",
            id: 1,
          - membership: {
              id: 1,
              userId: 1,
              teamId: 1
            }
        },
        - {
            name: "mosa climbers",
            id: 2,
          - membership: {
              id: 2,
              userId: 1,
              teamId: 2
            }
        }
      ]
  },
```

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:

```
                                                                                    copy
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:

```
                                                                                    copy
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:

```
                                                                                    copy
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()
  }
})
```

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()
  }
})
```

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) => {                                              copy
  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:
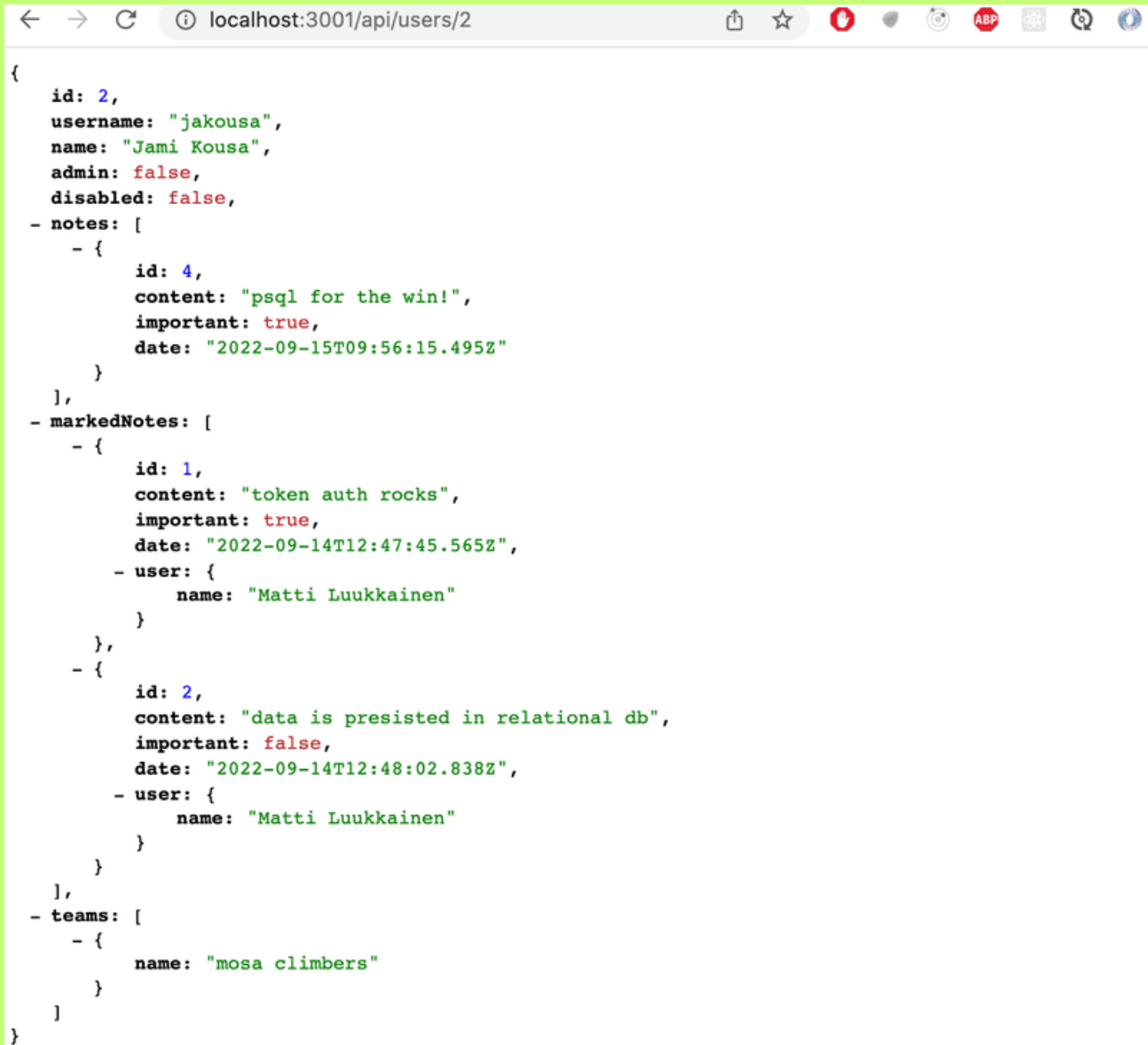
```
insert into user_notes (user_id, note_id) values (1, 4);                              copy
insert into user_notes (user_id, note_id) values (1, 5);
```

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:

```
- {
    id: 2,
    username: "lynx",
    name: "Kalle Ilves",
    admin: null,
    disabled: null,
  - notes: [
      - {
          id: 3,
          content: "Expo is a must in React Native development",
          important: true,
          date: "2021-10-11T14:20:20.149Z"
        }
    ],
  - markedNotes: [
      - {
          id: 1,
          content: "Relational databases rule the world",
          important: true,
          date: "2021-10-11T14:19:05.207Z",
          - user: {
              name: "Matti Luukkainen"
            }
        },
      - {
          id: 2,
          content: "MongoDB is webscale",
          important: true,
          date: "2021-10-11T14:19:23.881Z",
          - user: {
              name: "Matti Luukkainen"
            }
        }
    ],
  - teams: [
      - {
          name: "mosa climbers",
          id: 2
        }
    ]
},
```

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

Tasks 13.19.-13.23.

Task 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.

Task 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
        }
      ]
    }
  ]
}
```

copy

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 }
```

copy

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 {}                                                          copy

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()
```

`copy`

It is also possible to chain scopes:

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

`copy`

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'],
```

`copy`

```
      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'})          copy
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() {                                        copy
  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)                           copy
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 {}                                      copy

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')                                        copy

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
```

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.

Task 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