Veebiarendus

Back-End arendus

Martti Raavel

martti.raavel@tlu.ee

Tänased teemad

- Meenutame eelmist loengut
- JOIN laused MySQL-is
- Vigade haldus Express API-s
- Logimine Express rakenduses

Millest rääksimi eelmisel korral?

JOIN laused MySQL-is

MySQL JOIN-id on vahend, mis võimaldab andmeid erinevatest tabelitest pärida ja ühendada. JOIN-id võimaldavad kombineerida ridu kahest või enamast tabelist, mis on omavahel seotud primaar- ja võõrvõtmete kaudu.

JOIN-ide tüübid

- `INNER JOIN`
- `LEFT JOIN`
- `RIGHT JOIN`
- `FULL JOIN`

`INNER JOIN`

`INNER JOIN` tagastab ainult need read, millel on mõlemas tabelis vastavus.

```
SELECT column_name(s)
  FROM left_table
  INNER JOIN right_table
  ON left_table.column_name = right_table.column_name;
```

> Vasak ja parem tabel tähistavad tabelite järjekorda päringus.

`INNER JOIN` kasutamine

Tegevused koos kasutajate andmetega:

```
SELECT users.firstName, users.lastName, todos.title, todos.description, todos.is_done
  FROM todos
  INNER JOIN users
  ON todos.user_id = users.id
WHERE todos.deleted_at IS NULL;
```

`LEFT JOIN`

`LEFT JOIN` tagastab kõik read vasakust tabelist ja vastavad read paremast tabelist. Kui vastavust pole, siis täidetakse parema tabeli väljad NULL-idega.

```
SELECT column_name(s)
  FROM left_table
  INNER JOIN right_table
  ON left_table.column_name = right_table.column_name;
```

`LEFT JOIN` kasutamine

Tegevused koos kasutajate andmetega:

```
SELECT users.firstName, users.lastName, todos.title, todos.description, todos.is_done
  FROM todos
  LEFT JOIN users
  ON todos.user_id = users.id
WHERE todos.deleted_at IS NULL;
```

> Nagu näha, on `LEFT JOIN` kasutamine sarnane `INNER JOIN` -iga, vahe tekib sisse siis, kui tegemist on andmetega, millel pole vastavust.

`RIGHT JOIN`

`RIGHT JOIN` tagastab kõik read paremast tabelist ja vastavad read vasakust tabelist. Kui vastavust pole, siis täidetakse vasaku tabeli väljad NULL-idega.

`FULL JOIN`

FULL JOIN tagastab kõik read, kui on vastavus vasakus või paremas tabelis. MySQL-is ei ole otsest `FULL JOIN` toetust, kuid seda saab saavutada `UNION` abil.

Vigade haldus Express API-s

Vigade haldamine on oluline osa iga tarkvara arendamisel. Vigade haldamine on protsess, mis aitab tuvastada, jälgida ja lahendada vigu tarkvaras. Meie API on hetkel tehtud selliselt, et vigu haldame põhiliselt kontrollerites, kus moodustame kliendi jaoks vastava vastuse koos veateatega ja tagastame selle.

Sellise lähenemise probleem on selles, et kui meil on palju kontrollereid, siis peame igas kontrolleris vigade haldamiseks kirjutama sama koodi. Lisaks, kui me tahame ühel hetkel hakata veateateid logima, siis peame iga kontrolleri jaoks logimise koodi kirjutama.

Vigade halduse vahevara

Express API-s on võimalik teha nii, et vigade haldus on eraldatud kontrolleritest ja on koondatud ühte kohta. Selleks kasutame Express-i vahevara (middleware).

Vahevara kasutamine

Vigade halduse vahevara sisaldab erinevalt tavalisest vahevarast nelja parameetrit: `err`, `req`, `res` ja `next`. Kui teiste parameetritega oleme juba varem tuttavad, siis `err` on uus parameeter, mis sisaldab veaobjekti. Veaobjekt on Expressis spetsiaalne objekt, mis sisaldab veakoodi, veateadet ja muud viga kirjeldavat infot. Kui me käivitame `next()` funktsiooni koos veaobjektiga, siis see edastatakse viga otse veahalduse vahevarale.

```
const errorHandler = (err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send('Something broke!');
};

app.use(errorHandler);
```

> Vea objekt sisaldab muuhulgas ka nn `stack` omadust, mis sisaldab vea põhjustanud funktsioonide järjekorda ja on väga kasulik vea põhjuste tuvastamisel.

Veahalduse vahevara registreerimine

Vigade halduse vahevara tuleb registreerida kõige viimase vahevarana. Lisaks tuleb meil tegelikult muuta ka meie kontrollerite koodi, et vigade haldus toimiks - peame need muutma omakorda vahevaraks, et saaksime vea korral selle `next()` funktsiooni abil veahalduse vahevarale saata.

```
app.use('/todos', todosRouter);

app.use(errorHandler);
```

Kontrollerite muutmine vahevaraks - alg ne kontroller

```
const getAllComments = async (req, res) => {
  const comments = await commentsService.getAllComments();
  if(!comments) {
    return res.status(500).json({
      success: false,
      message: 'Something happened while fetching comments',
    });
  }
  return res.status(200).json({
    success: true,
    message: 'All comments',
    comments,
  });
};
```

Kontrollerite muutmine vahevaraks - muudetud kontroller

```
const getAllComments = async (req, res, next) => {
  try {
    const comments = await commentsService.getAllComments();
    if(!comments) {
      const error = new Error('Something happened while fetching comments');
      error.status = 500;
      throw error;
    }
    return res.status(200).json({
      success: true,
      message: 'All comments',
      comments,
    });
  } catch (error) {
    return next(error);
  }
};
```

Mis juhtub, kui viga tekib?

Kui nüüd kontrolleris tekib viga (või see viga tagastatakse teenuse poolt), siis me moodustame uue veaobjekti koos enda poolt moodustatud veateatega. Seejärel lisame sellele veaobjektile ka veakoodi ja 'viskame' selle `catch` ploki poolt kinnipüüdmiseks, mis omakorda edastab selle `next()` funktsiooni abil veahalduse vahevarale. Vahevara saab vea kätte ja edastab selle kliendile. Sellisel viisil käivad meil kõik vead läbi ühe keskse koha, kus saame neid logida ja käsitleda.

Nüüd tuleks veel üle vaadata ka teenused

Kuna teenused tagastavad meil hetkel vea tekkimise korral lihtsalt `null`-i või `false`, siis on meil selle info põhjal keeruline otsustada, kas tegemist on tõelise veaga või mitte. Seega peaksime ka teenustes tegema muudatusi, et nad viskaksid vea korral veaobjekti, millel on veakood ja veateade.

Teenuse muutmine

```
const getAllComments = async () => {
  try {
    const comments = await Comment.findAll();
    return comments;
  } catch (error) {
    const err = new Error('Something happened while fetching comments');
    err.status = 500;
    throw err;
  }
};
```

Logimine

Logimine on protsess, kus salvestatakse rakenduse tegevusi ja sündmusi, et neid hiljem analüüsida. Logimine on oluline osa tarkvaraarendusest, sest see aitab meil mõista, mis meie rakenduses toimub ja kuidas seda parandada.

Samuti võib logimise abil tuvastada pahatahtlikke tegevusi ja ründeid, mis võivad meie rakendust ohustada.

Logimine Express rakenduses

Siiani oleme oma rakenduses logimist kasutanud väga vähesel määral. Kui meil tekib viga, siis logime selle konsooli ja saadame kliendile vastava veateate. Kuid see pole piisav, sest me ei saa hiljem analüüsida, mis viga tekkis ja miks see tekkis, sest ei ole reaalne, et meil on alati võimalik konsooli jälgida. Samuti on meil keeruline konsoolist otsida konkreetset viga, kui meil on palju logisid.

Winston logger

Winston on logimise teek, mis võimaldab meil logida erinevaid sündmusi ja tegevusi meie rakenduses. Winston on väga paindlik ja võimaldab meil logida erinevatesse kohtadesse, näiteks faili, andmebaasi või konsooli.

Winston paigaldamine

Winstoni paigaldamiseks kasutame npm-i:

```
npm install winston
```

Winston kasutamine

Winstoni kasutamiseks peame looma loggeri. Logger on objekt, mis sisaldab logimise seadeid, nagu logi tase, logi formaat ja transpordid, mis määravad, kuhu logitakse.

```
// logger.js
const { createLogger, format, transports } = require('winston');
const { combine, timestamp, printf, errors } = format;

// Kohandatud logi formaat
const logFormat = printf(({ level, message, timestamp, stack }) => {
  return `${timestamp} ${level}: ${stack || message}`;
});

const logger = createLogger({
  level: 'info', // Logi tase - info, error, warn, debug
  format: combine(
    timestamp(),
    errors({ stack: true }), // Logi veateated koos stack trace'iga
    logFormat // Kohandatud logi formaat
  ),
  transports: [ // Transpordid - kuhu logitakse
    new transports.Console(), // Logi konsooli
    new transports.File({ filename: 'logs/combined.log' }), // Salvesta kõik logid faili nimega 'combined.log'
    new transports.File({ filename: 'logs/errors.log', level: 'error' }), // Salvesta ainult veateated faili nimega 'errors.log'
  ],
});

module.exports = logger;
```

Winston kasutamine - vigade logimine

Kõigepealt soovime logida kõik vead eraldi faili, et neid hiljem analüüsida. Selleks lisame oma vigade halduse vahevarale logimise.

```
const logger = require('./logger');

const errorHandling = (err, req, res, next) => {
  logger.error(err.message, { stack: err.stack });
  res.status(err.status || 500).json({
    success: false,
    message: err.message || 'Internal Server Error',
  });
}
```

Nüüd kuvatakse kõik veateated nii konsoolis, kui ka failis `errors.log`.

Morgan

Morgan on logimise teek, mis on spetsiaalselt loodud HTTP päringute logimiseks. Morgan on väga paindlik ja võimaldab meil logida erinevaid andmeid, nagu päringu meetod, URL, vastuse staatuskood ja muud.

Morgan paigaldamine

Morgani paigaldamiseks kasutame npm-i:

```
npm install morgan
```

Morgan kasutamine

Morgani kasutamiseks lisame selle oma Express rakendusse. Morgani kasutamiseks peame määrama, millist logi formaati soovime kasutada ja kuhu soovime logida. Meie rakenduses kasutame `combined` logi formaati ja kasutame logimiseks omakorda Winstoni loggerit.

```
const morgan = require('morgan');
const logger = require('./logger');

app.use(morgan('combined', { stream: { write: message => logger.info(message.trim()) }}));
```

Kodune töö

- Rakenda oma API-s vigade haldus
- Logi kõik päringud ja vead faili