VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Sami Nuutinen

Full stack -kehitys Reactilla ja Nodella

Tekniikka
2023

VAASAN AMMATTIKORKEAKOULU
IT-Insinööri

TIIVISTELMÄ

Tekijä	Sami Nuutinen
Opinnäytetyön nimi	Full stack -kehitys Reactilla ja Nodella
Vuosi	2023
Kieli	suomi
Sivumäärä	60
Ohjaaja	Harri Lehtinen

Reactin ja Noden tuntemuksesta on paljon hyötyä ohjelmistokehityksessä. JavaScript-ohjelmistokehyksiä ja -kirjastoja on useita moneen eri tarkoitukseen. React soveltuu erinomaisesti frontend-kehitykseen ja se on laaja-alaisesti käytössä myös yritysmaailmassa.

Full stack -kehitykseen perehtyminen onnistuu parhaiten käytännön kautta. Node.js mahdollistaa JavaScript koodin suorittamisen palvelimella. React frontendin ja Node backendin lisäksi myös tietokantayhteys on tarpeellinen tiedon varastoimiseksi Full stack -kehitysprojektissa.

Reactin kaltaiseen JavaScript-kirjastoon tutustuminen auttaa ymmärtämään JavaScriptiä kokonaisuutena. Tämä mahdollistaa myös muiden JavaScript-kirjastojen ja ohjelmistokehyksien helpomman omaksumisen.

Tähän opinnäytetyöhön liittyy myös projektiluontoisesti toteutettu tietovisasovellus. Backend on toteutettu Noden avulla ja frontend Reactin avulla. Viimeiset luvut kuvaavat projektin kehityksen vaiheita.

Avainsanat ohjelmistokehitys, JavaScript, full stack -kehitys, frontend, backend

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
IT-Engineer

ABSTRACT

Author	Sami Nuutinen
Title	Full stack development using React and Node.js
Year	2023
Language	Finnish
Pages	60
Name of Supervisor	Harri Lehtinen

It is useful in software development to have some knowledge of React and Node. There is a large quantity of JavaScript libraries and frameworks available for a multitude of purposes. React is excellent for frontend development and it is widely used in the corporate life.

To get acquainted with full stack development is best achieved through practical means. Node.js makes it possible to run JavaScript on server side. In addition to React front end and a Node.js backend a database connection is also necessary to store data in full stack development.

To get familiar with a JavaScript library like React helps in understanding JavaScript as a whole. This also makes it easier to learn the “dos and dont’s” of other JavaScript libraries and frameworks.

A quiz web application project is also related to this thesis. Backend has been developed using Node.js and frontend using React. Last chapters describe the development of the project.

Keywords software development, JavaScript, Full Stack development, frontend, backend

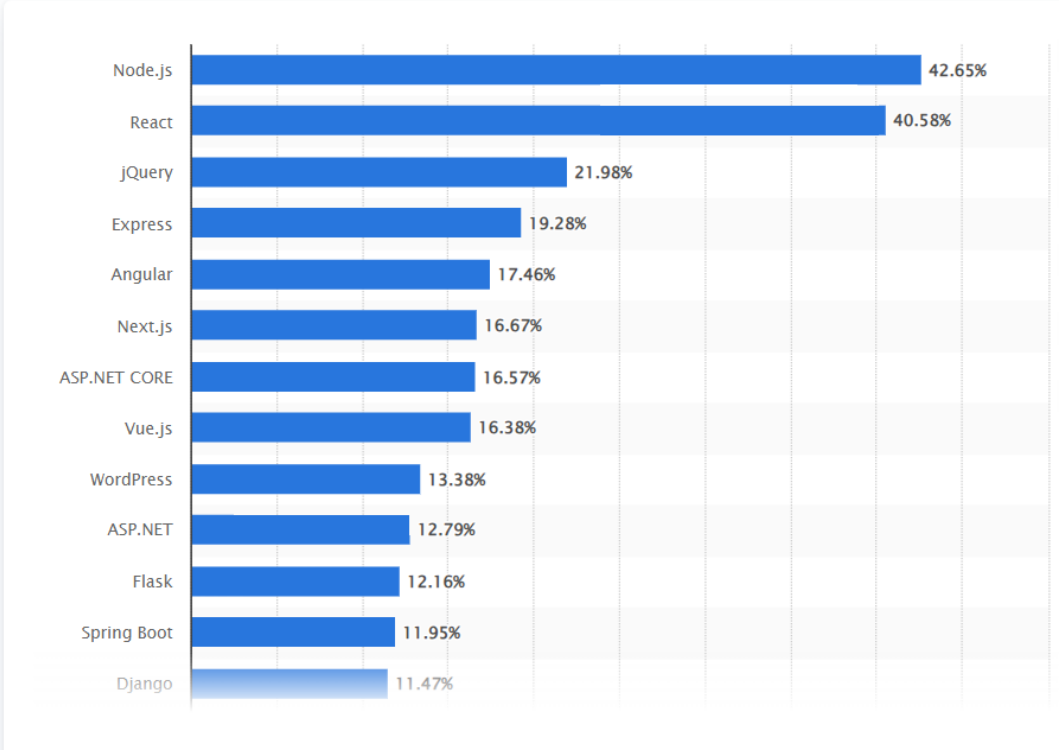
SISÄLLYS

KUVIO- JA TAULUKKOLUETTELO

1 JOHDANTO

Ohjelmistokehittäjälle tarjolla olevien työkalujen määrä on suuri ja valikoima kasvaa yhä suuremmaksi vuosi vuodelta. Pääasiassa kehitysprojekteissa käytettävät ohjelmointikielet, ohjelmistokehykset ja kirjastot valikoituvat käyttötarkoituksen mukaan. Jotkut soveltuvat esimerkiksi frontend-kehitykseen paremmin kuin toiset ja jotkut taas on tarkoitettu backend-kehitykseen sopiviksi.

Reactin ja Noden käyttö yhdessä mahdollistaa JavaScriptiin perustuvan full stack kehityksen. React soveltuu erinomaisesti frontend-kehitykseen ja Node.js backend-kehitykseen. Tämän osoittaa niiden laaja-alainen käyttö kehittäjien keskuudessa maailmanlaajuisesti, kuten kuvassa 1 oleva pylväsdiagrammi osoittaa.



Kuva 1. Käytetyimmät ohjelmistokehykset web-kehityksessä 2023 [1].

1.1 Full stack -kehitys käsitteenä

Full stack -kehittäjällä tarkoitetaan ohjelmistokehittäjää, jolla on hallussa frontend- ja backend-kehitystyö. Full stack -kehittäjän tulee siis omata iso kirjo osaamista erilaisista teknologioista, työkaluista ja ohjelmointikielistä. Full stack -kehittäjä on webkehityksen moniosaaja.

Backend ja frontend voidaan toteuttaa eri kielillä, jotka kommunikoivat REST-rajapinnan välityksellä. JavaScriptin lisäksi esimerkiksi PHP, Python ja Java soveltuvat hyvin backend kehitykseen. On kuitenkin monella tapaa hyödyllistä ja kehitystyötä helpottavaa, jos sekä backend ja frontend on kehitetty käyttäen esimerkiksi JavaScriptiä.

1.2 Frontend

Frontendillä tarkoitetaan verkkosivun käyttäjälle näkyvää osaa, eli esimerkiksi painikkeita, linkkejä ja lomakkeita. Frontendiin kuuluu myös verkkosivulle CSS-tyylikieltä (Cascading Style Sheets), tai muuta vastaavaa käyttämällä määritellyt tyylit. Näihin kuuluvat esimerkiksi navigointipalkin ulkoasu ja asettelu.

HTML (HyperText Markup Language) on isossa osassa frontend-kehityksessä muiden ohjelmointikielten tukemana ja sen perusteiden ymmärtäminen on olennaista. Myös React-komponentit palauttavat HTML:ää. Käyttäjän vuorovaikutus verkkosivun toiminnallisuuteen tapahtuu frontendissä, kun taas tiedonkäsittely ja esimerkiksi tietokantaan tallentaminen tai hakeminen tapahtuu backendissä.

1.3 Backend

Backendillä tarkoitetaan verkkosivun käyttäjälle ”näkymätöntä” osaa eli palvelimia ja tietokantayhteyksiä. Kun käyttäjä esimerkiksi painaa painiketta kirjautuakseen sisään, käyttäjän validointi tapahtuu palvelimella. Käyttäjätunnusta ja salasanaa verrataan tietokannasta haettuun hashattuun eli tiivistettyyn salasanaan.

Salasanan hashaaminen on tietoturvaa koskeva, tietomurron varalta suoritettava toimenpide. Palvelin on kuitenkin vastuussa monesta muustakin asiasta. Node.js-palvelimelle voi esimerkiksi määrittää reittejä, joiden avulla on mahdollista hakea mm. tietoa tietokannasta ja palauttaa frontendiin käyttäjälle tiettyyn urliin tehtävän pyynnön avulla.

Frontend siis pääasiassa renderöi palvelimelta haettua tietoa käyttäjän nähtäväksi ja antaa käyttäjälle mahdollisuuden päivittää tai luoda uutta tietoa painikkeiden ja lomakkeiden avulla. Backend pääasiassa käsittelee frontendiltä tai tietokannasta tullutta tietoa palvelimelle tehtyjen pyyntöjen määrittelemällä tavalla ja lähettää vastauksen frontendille. Näiden muodostaman kokonaisuuden hallitsevaa ohjelmistokehittäjää voidaan kutsua full stack -kehittäjäksi.

1.4 Yleistä JavaScriptistä

JavaScriptin tarina alkoi vuonna 1995, kun NetScape palkkasi Brendan Eichin kehittämään helppokäyttöistä ohjelmointikieltä dynaamisempien verkkosivujen luomiseksi. Lopputuloksena oli Mocha, joka nimettiin myöhemmin LiveScriptiksi ja myöhemmin, kun sopimus syntyi NetScapen ja Sunin välillä, JavaScriptiksi. Siitä markkinoitiin avustavaa kieltä Javalle, joka suorittaisi pienempiä toimenpiteitä selaimessa Javan huolehtiessa isommista kokonaisuuksista [2.]

JavaScript on kasvanut vuosikymmenten saatossa avoimen lähdekoodin JavaScript-kirjastojen ja ohjelmistokehysten ansiosta merkittäväksi ohjelmointikieleksi webkehityksessä. Sen heikko tyypitys ja dynaamisuus mahdollistaa yksinkertaisten verkkosivujen nopean ja tehokkaan tuottamisen. Esimerkiksi Javaan verrattuna JavaScript-syntaksi on joustava ja ohjelmointikielenä se soveltuu hyvin moderniin webkehitykseen monipuolisuutensa, sekä matalan oppimiskäyrän ansiosta.

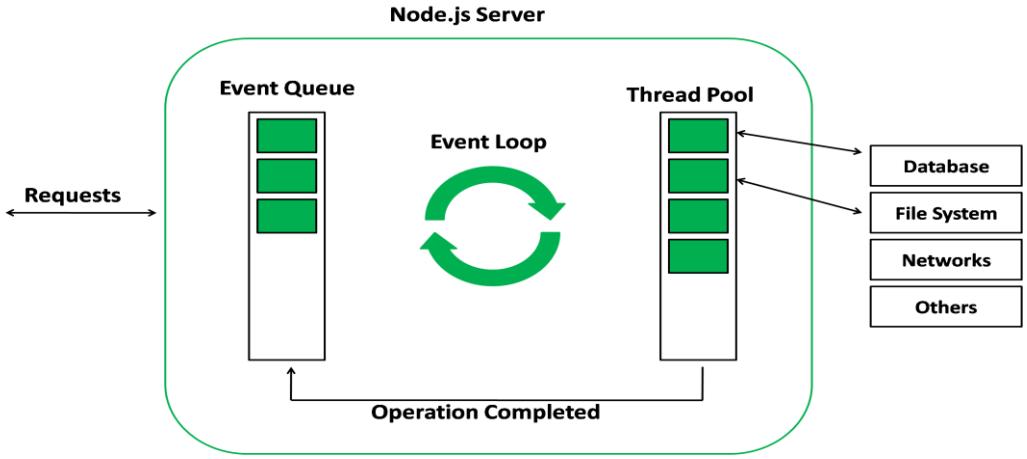
2 BACKEND NODELLA

2.1 Node.js

Siinä missä Java tarvitsee kääntäjän koodin kääntämiseksi konekieliseksi suorittamista varten, JavaScript tulkittuna kielenä tarvitsee tulkin. Frontendissä tämä on yleensä sisäänrakennettu selaimeen. Backendissä alustariippumattoman ajoympäristön JavaScriptin suorittamiseen tarjoaa Node.js V8 JavaScript-moottorin avulla, jonka logo on kuvassa 2.

Kuva 2. Node.js logo [3].

Aluksi JavaScriptiä käytettiin pääasiassa frontend puolella HTML:n yhteydessä, <*script*>-merkinnän avulla [4]. Ratkaisun JavaScriptin käyttämiseen backendissä tarjosi Ryan Dahl vuonna 2009 Node.js:n muodossa. Se yhdistää V8 JavaScript Chrome-moottorin, matalan tason I/O-rajapinnan (Input/Output) toiminnallisuuksia ja tapahtumasilmukan, jota havainnollistaa kuva 3 [5.]

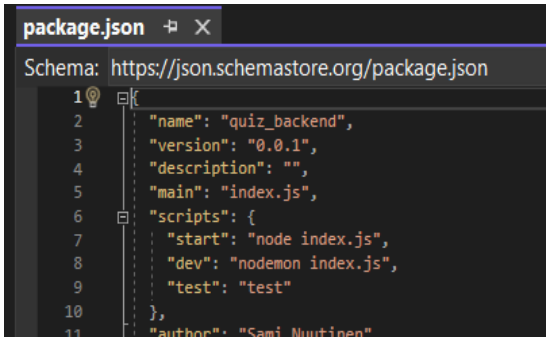


Kuva 3. Node.js:n tapahtumasilmukan toimintaperiaate [6].

Tapahtumasilmukka mahdollistaa ns. non-blocking I/O operaatioiden suorittamisen Nodessa huolimatta JavaScriptin yksisäikeisyydestä. Käynnistettäessä tapahtumasilmukka alustetaan, annettu syötekoodi käsitellään ja tapahtumasilmukan prosessointi aloitetaan. Operaatioiden sysääminen järjestelmän ytimelle mahdollistavat non-blocking-operaatioiden toteutumisen, eli samanaikaisten operaatioiden suorittamisen asynkronisesti odottamatta muiden operaatioiden käsittelyn päättymistä [7.]

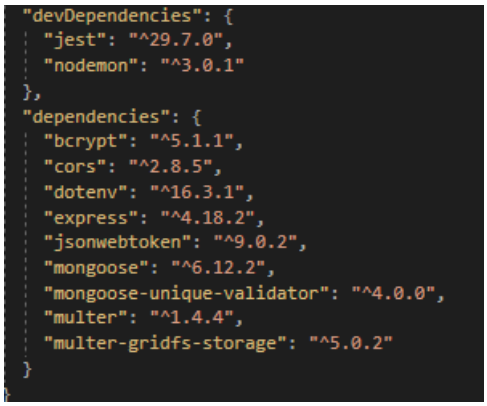
2.2 Npm (Node package manager)

Noden asennuksen yhteydessä mukana tulee oletuksena myös eräs tärkeä työkalu. Npm on paketinhallintatyökalu, jonka ansiosta tarpeellisten kirjastojen asentaminen ja poistaminen sekä esimerkiksi npm scriptien eli kustomoitujen komentojen määrittely on helppoa. Kirjastojen viimeisimmän saatavilla olevan version asennus onnistuu komentorivillä komennolla ”npm install <kirjaston nimi>” ja komentoja, kuten tiedoston index.js käynnistäminen nodemon työkalua käyttämällä voidaan kirjoittaa projektikansiossa olevaan tiedostoon package.json kuvan 4 esimerkin mukaisesti.



Kuva 4. Npm scriptejä package.json-tiedostossa. Komennolla ”npm run dev” voidaan suorittaa dev-kohtaan kirjoitettu komento komentoriviltä.

Suuntaamalla komentorivillä haluttuun kansioon, antamalla komento npm init ja vastaamalla esitettyihin kysymyksiin, voidaan luoda Node.js-projektin runkona toimiva package.json tiedosto. On mahdollista asentaa kaikki package.json tiedostossa listatut paketit, kuten projektissa käytettävät kirjastot npm:n luomaan node_modules kansioon käyttämällä projektikansiossa komentoriviltä komentoa npm install. Versiotiedot ja riippuvuusmäärittelyt on listattu package.json tiedostoon kuvan 5 esimerkin mukaisesti.



Kuva 5. Riippuvuuksien lista package.json-tiedostossa.

Backendin toteuttaminen yksinään Noden HTTP-moduulin palvelimen avulla onnistuu kyllä, mutta on sovelluksen kasvaessa vaivalloista. Noden pääasiallinen tehtävä full stack -kehityksessä on toimia palvelinpuolen JavaScriptin suoritusympäristönä. Palvelimen toteutuksessa apuna voi käyttää esimerkiksi Express-ohjelmistokehystä.

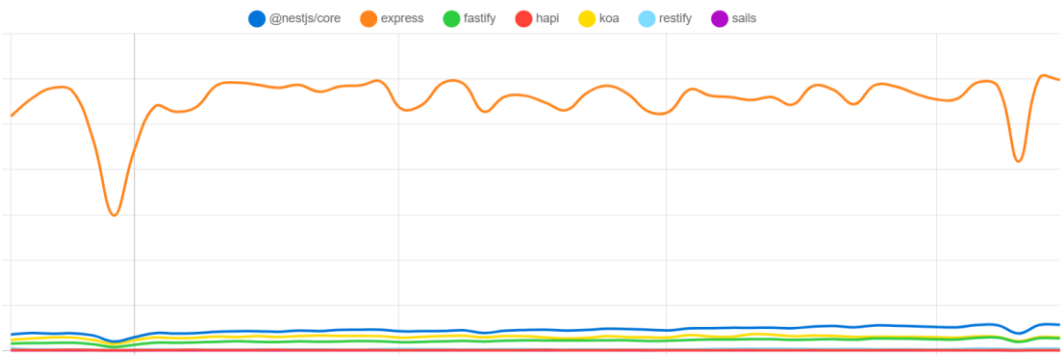
2.3 Nvm (Node version manager)

Yksi npm:n yhteydessä käytettävä hyödyllinen työkalu on nvm. Sen avulla on mahdollista hallinnoida olemassaolevia Node.js-asennuksia helposti komentoriviltä. Jos esimerkiksi jokin tarpeellinen kirjasto aiheuttaa yhteensopivuusongelmia käytössä olevan Node-version kanssa, nvm:n avulla voi helposti myös vaihtaa käytössä olevaa Node-versiota toiseen asennettuun versioon.

Nvm mahdollistaa useampien Node-versioiden olemassaolon käyttäjän koneella samanaikaisesti. Uusien Node-versioiden asentaminen ja asennusten poistaminen on myös mahdollista sen avulla. Jos käytössä on useampi Node versio niiden välillä vaihtaminen onnistuu komennolla nvm use.

2.4 Express

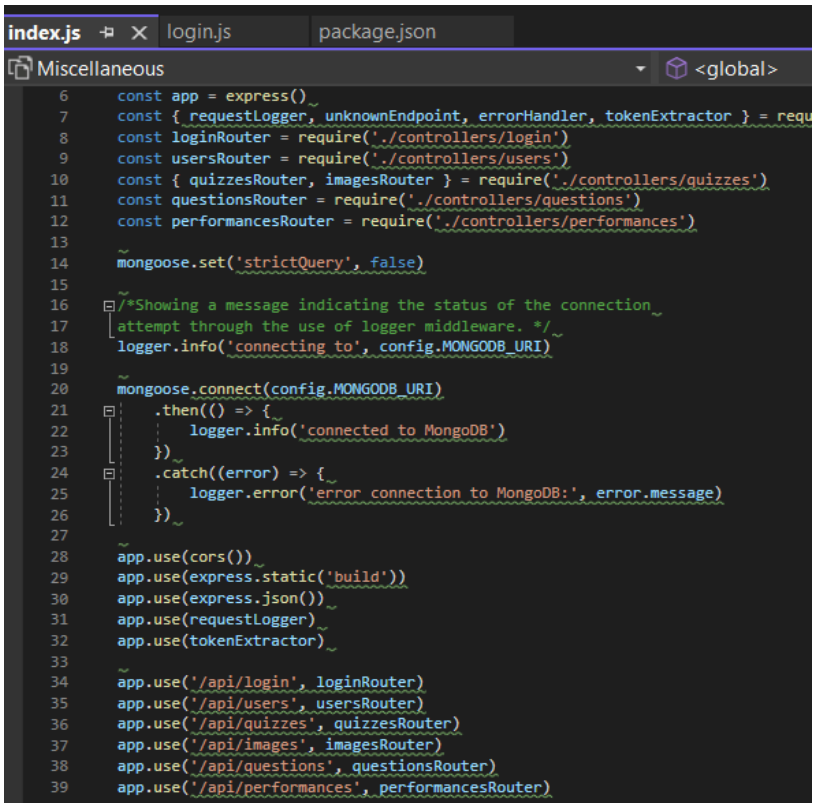
Express on tarkoitettu helpottamaan backend-kehitystä Nodella ja REST-rajapinnan luominen backendiin onnistuu melko vaivattomasti sen avulla. Expressillä voi määritellä reittejä ja vastata frontendistä tulleisiin http-pyyntöihin esimerkiksi palauttamalla jsonwebtoken-kirjaston avulla luotu valtuutustoken kirjautumisen yhteydessä. Se on vastaavien joukossa tällä hetkellä selkeästi suosituin ratkaisu kuten kuvan 6 käyrä osoittaa.



Kuva 6. Express-ohjelmistokehyksen lataukset viimeisen vuoden aikana verrattuna muihin vastaaviin [8].

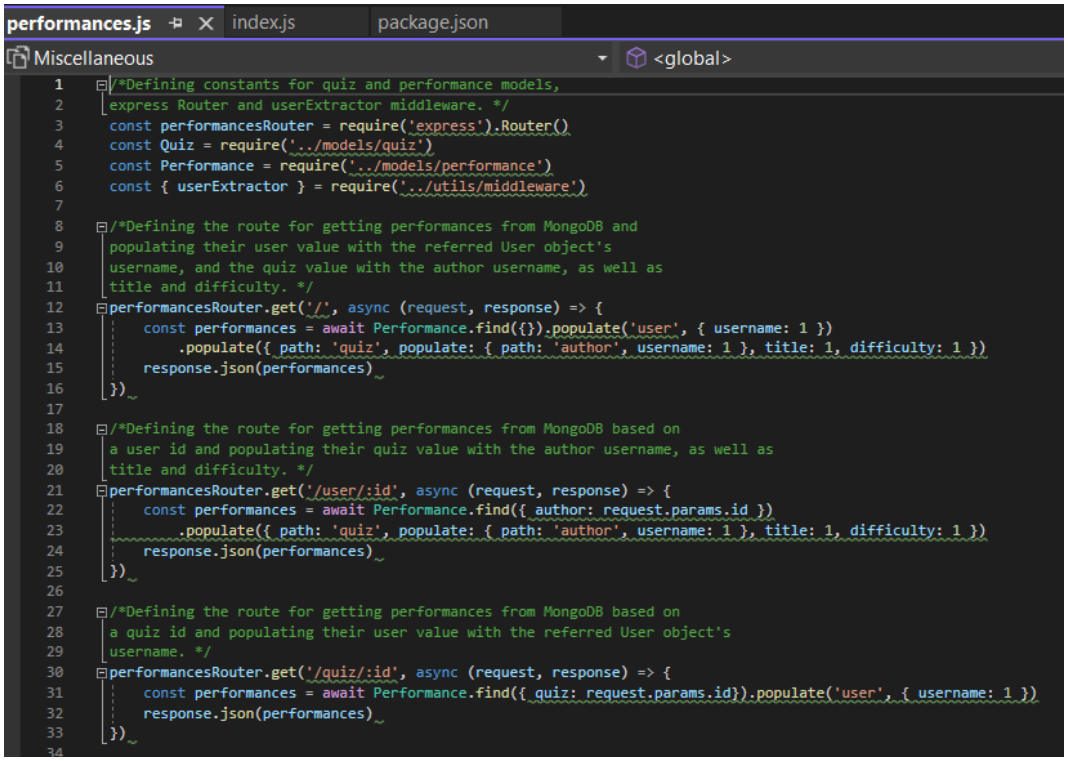
Reittien määrittely onnistuu Expressin Router funktion avulla. Yksi tapa on luoda .js-päätteinen JavaScript-tiedosto controllers-kansioon ja määritellä tiedostossa

Router, halutut reitit esimerkiksi usersRouter.post POST-metodille, toiminnallisuus vastaanotettaessa pyyntö reittiä vastaavaan urliin ja määritelty reitti vietäväksi palvelimen muille osille module.exports-määrittelyssä. Määritelty Router voidaan sitten tuoda index.js-tiedostoon ja ottaa käyttöön require- ja use-määrittelyn avulla kuvan 7 esimerkin mukaisesti.

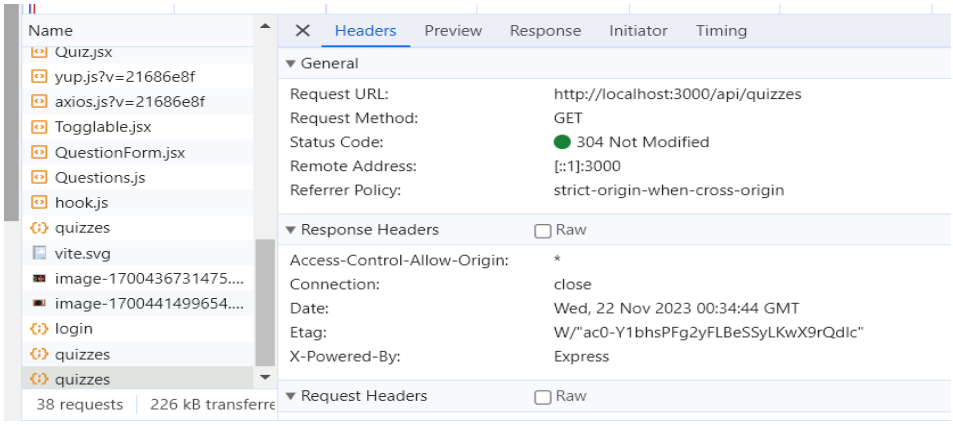


Kuva 7. Reittien käyttöönottoja index.js tiedostossa.

Router-funktioita voidaan käyttää useampia eri urleissa. Jokainen const-määrittelyllä varustettu vakiomuuttuja, joka käyttää Expressin Router funktiota, voi pitää sisällään monta määriteltyä reittiä. Jos esimerkiksi vakiomuuttuja usersRouter toimisi urlissa ”/api/users”, palvelin kuuntelisi oletusarvoisesti porttia 3000 ja usersRouterissa olisi määritelty polku usersRouter.get(”/:id”), metodia GET käyttävä http-pyyntö urliin ”http//:localhost:3000/api/users/5” kohdistuisi määriteltyyn usersRouter.get-polkuun, joka vastaanottaisi pyynnön mukana muuttujan id-arvolla 5, kuvassa 8 olevan performances.js-tiedoston määrittelyjen mukaisesti ja pyynnön tietoja voi tarkkailla selaimessa kuvan 9 esimerkin tapaan.



Kuva 8. Esimerkki reittien määrittelystä get-pyynnöille eri urleissa id-parametrilla ja ilman.



Kuva 9. Lista, jossa mm. frontendiltä backendille tehtyjä pyyntöjä ja valitun yksittäisen pyynnön otsikkokentät.

Palvelinta luotaessa backendiin tulee luultavasti tarve käyttää myös cors (Cross-Origin-Resource-Sharing) middlewarea. Esimerkiksi jos frontendin urlista ”http://localhost:3000” tai ”https://localhost:3000” tulisi pyyntö backendiin osoitteessa ”http://localhost:3001”, ei tietoturvamekanismi ”same-origin-policy” sallisi portista 3000 tulevan tai https-protokollaa käyttävän pyynnön käsittelyä.

Cors-middlewaren asentaminen onnistuu komentoriviltä npm install-komennolla ja sen käyttöönotto ratkaisee tämän ongelman.

2.5 MongoDB

Tietokanta on myös erittäin oleellinen backend-kehityksessä. Sen tarkoitus on säilyttää suuria määriä tietoja tietoturvallisesti, kuten esimerkiksi yrityksen asiakastietoja tai tuotteiden tietoja. Backendin välityksellä tietokannasta haettavia tietoja käsitellään frontendissä ja uutta frontendilta saatua tietoa tallennetaan tietokantaan.

SQL-tietokantojen (Structured Query Language) lisäksi on mahdollista käyttää myös ”NoSQL”-tietokantoja. SQL- eli relaatiotietokannat perustuvat strukturoidun tiedon tallentamiseen ja hakemiseen SQL-kyselykieltä käyttäen. Relaatiotietokantoihin tallennetaan strukturoitua tietoa taulukoina, joille voi määritellä taulukoiden välisiä suhteita perusavainten ja viiteavainten avulla.

Taulukko 1. Erilaisia datan määrittelyjä [9].

	Jäsentämätön tieto	**Puolirakenteinen tieto (Semi-structured data)**	**Strukturoitu, eli jäsennelty tieto**
Ominaisuudet	Ei määriteltyä tietomallia. (Data model) Hankala hakea.	Löyhärakenteinen (Loosely-coupled) tietomalli.	Selkeästi määritelty tietomalli. Helppo hakea.
Esimerkki	Kuvatiedosto	Puhelinkeskuksen lokitiedot	Taulukkolaskentaohjelma (Spreadsheet)
Varastointi	Tietoallas (Data lake)	Järjestetty meta tageilla.	Relaatiotietokanta

Siinä missä relaatiotietokannat käsittelevät tietoa riveistä ja sarakkeista koostuvina taulukoina, NoSQL:ksi luettava dokumenttitietokanta MongoDB käsittelee

tietoa erillisinä dokumentteina. MongoDB:lle luodaan skeema, jossa määritellään tallennettavan dokumentin tyyppi ja sen sisältämät kentät. MongoDB:tä edustaa kuvassa 10 oleva logo ja sen skeemoille on mahdollista luoda viitekenttiä, joiden avulla yhden skeeman kenttä voi saada arvokseen toisessa skeemassa määritellyn olion kuvassa 11 esitetyn esimerkin tapaan.

Kuva 10. MongoDB:n logo [10].

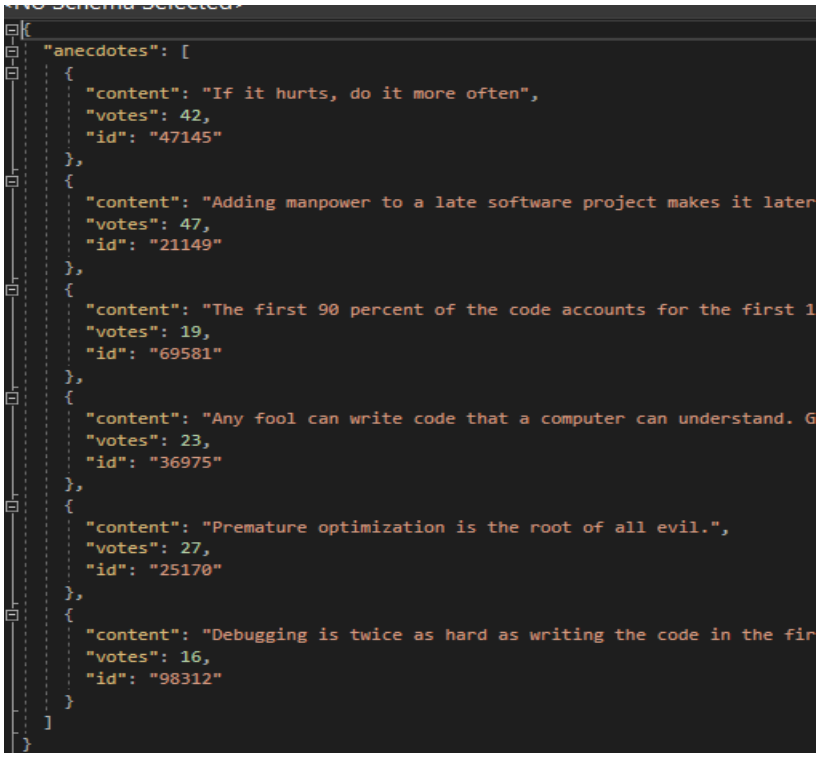
```
const quizSchema = new mongoose.Schema({
    title: {
        type: String,
        required: [true, 'Title cannot be empty.'],
        unique: [true, 'This title is already in use by another quiz. Quiz titles must be unique.']
    },
    difficulty: {
        type: Number,
        default: 0
    },
    completedAt: {
        type: Number,
        default: 0
    },
    highScore: {
        type: Number,
        default: 0
    },
    highScoreSetBy: {
        type: mongoose.Schema.Types.ObjectId,
        ref: 'User',
        default: null
    },
    ratings: [
        Number
    ],
    questions: [
        {
            type: mongoose.Schema.Types.ObjectId,
            ref: 'Question'
        }
    ],
    author: {
        required: [true, 'Author cannot be empty'],
        type: mongoose.Schema.Types.ObjectId,
        ref: 'User'
    },
    image: String
})
```

Kuva 11. Esimerkki skeeman määrittelystä. Viite question-kentässä Question-skeemassa määritellystä oliosta koostuvaan taulukkoon ja author-kentässä user-skeemaa käyttävään olioon.

MongoDB-tietokantaa käytettäessä tietoa haetaan ja sitä tallennetaan JSON-(JavaScript Object Notation) muotoisena. JSON muodossa oleva data muodostuu hakasulkeiden sisällä olevista oliomäärittelyistä. Hakasulkeissa määritellään jouk-

ko kenttiä, joilla on arvo ja jotka voivat sisältää myös muita vastaavasti määriteltyjä olioita tai taulukoita.

JSON on siis toisinsanoen datansiirtoformaatti ja vaihtoehto XML-formaatille (Extensible Markup Language). Sen loi amerikkalainen Douglas Crockford ja se on johdettu JavaScript-standardista. Tästä syystä se seuraa JavaScript-oliosyntaksia kuten kuvassa 12 ja JSON muotoinen merkkijono on helposti muunnettavissa JavaScript-olioksi ja toisinpäin [11.]



Kuva 12. Esimerkki JSON-olioiden määrittelystä .db tiedostossa.

2.6 Mongoose

Mongoose on MongoDB:n kanssa käytettäväksi tarkoitettu Node.js-pohjainen ODM-kirjasto (Object Data Modeling), jonka avulla skeemojen luominen MongoDB:seen tallennettavaksi on yksinkertaista. Kirjaston asennus Node-projektin backendiin onnistuu komennolla npm install. Mongoosen avulla skeemoille on myös mahdollista luoda validointeja.

Jos esimerkiksi kentän tyypiksi määriteltäisiin Number ja sille yritettäisiin antaa arvoksi merkkijonoa String, tuloksena olisi validointivirhe, joka estäisi väärän-tyyppisten arvojen tallentamisen. JavaScriptin heikon tyypityksen huomioonottaen Mongoosen tarjoama tyyppivalidointi on varsin hyödyllinen ominaisuus. Mongoosen avulla voidaan myös estää esimerkiksi tyhjien arvojen asettaminen kentille, tai varmistaa että annettua arvoa ei ole jo tietokannassa, jos käytössä on mongoose-unique-validator-kirjasto.

Kenttiä voidaan Mongoosen avulla myös karsia pois palautettaessa dokumenttia tietokannasta. Esimerkiksi MongoDB:n dokumentin versiointiin liittyvä __v-kenttä voidaan määritellä skeemassa poistettavaksi palautuksesta haettaessa dokumentteja tai voidaan vaikkapa poimia MongoDB:n _id-kentän arvo ja asettaa se id-kentälle. Kuvassa 13 esitetty esimerkki kuvaa User-skeemaa, joka palauttaa tietokantaan tallennetun olion tiedot lukuunottamatta esimerkiksi salasanatiivistettä.

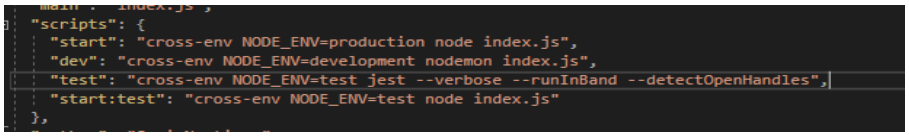
```
const userSchema = mongoose.Schema({
    username: {
        type: String,
        required: [true, 'Username cannot have an empty value.'],
        minlength: [3, 'Username must contain at least 3 characters.'],
        unique: [true, 'Username is already in use.']
    },
    passwordHash: String,
    quizzes: [
        {
            type: mongoose.Schema.Types.ObjectId,
            ref: 'Quiz'
        }
    ],
})

//Defining the mongoose-unique-validator plugin.
userSchema.plugin(uniqueValidator)

/*Define the properties of the objects that are returned by the toJSON method.
Exclude the _id value as well as the MongoDB version field __v and the
passwordHash. Also transform the value of _id from object to a string */
userSchema.set('toJSON', {
    transform: (document, returnedObject) => {
        returnedObject.id = returnedObject._id.toString()
        delete returnedObject._id
        delete returnedObject.__v
        delete returnedObject.passwordHash
    }
})

//Export the User object model.
const User = mongoose.model('User', userSchema)

module.exports = User
```

Kuva 13. Esimerkki skeeman validoinnista ja tarpeettomien kenttien karsimisesta.

2.7 Jest ja Supertest

Backend-testaamista varten voidaan luoda MongoDB tietokantaan pääohjelmasta erillinen collection eli kokoelma dokumenttejä. Sen osoite voidaan määritellä

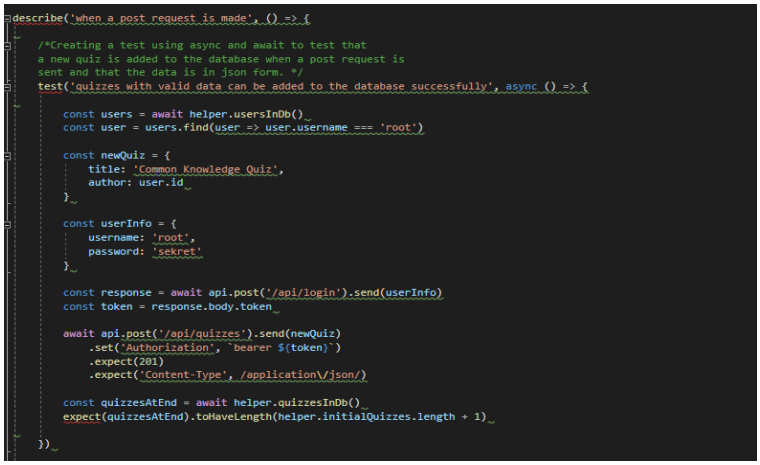
.env-tiedostossa olevaan ympäristömuuttujaan. Skriptin avulla voidaan kuvan 14 esimerkin mukaisesti määritellä komento, jonka avulla sovellusta käytetään tietyssä tilassa ja NODE_ENV-ympäristömuuttujan ollessa ”test” ohjelmoidaan backend käyttämään testitietokantaa.



Kuva 14. Esimerkki test-tilaa käyttävän käynnistysskriptin määrittelystä.

Jest on JavaScript yksikkötestaukseen tarkoitettu ohjelmistokehys. Sen avulla voi testata Nodella kehitetyn backendin yksittäisten osien, kuten esimerkiksi olioiden luomisen tai hakemisen toimintaa. Testit toimivat describe-, test- ja expect-komentojen avulla.

Describe määrittelee koodilohkon, jonka sisällä voidaan määritellä useampia testejä. Describe-lohkolle voidaan antaa kuvaus, joka kuvailee lohkon testejä ja voidaan myös määritellä esimerkiksi beforeEach-metodi, jossa voidaan alustaa tietokannan tila ennen jokaista testiä. Test-määrittelyllä määritellään testin sisältö ja expect-komennolla haluttu lopputulos, kuten kuvan 15 esimerkissä.



Kuva 15. Esimerkki Jestin ja Supertestin avulla määritellystä testistä.

Supertest on Node.js-rajapintojen testaamiseen tarkoitettu testikirjasto. Sen avulla on mahdollista kääriä express-sovellus ns. ”superagent”-olioksi, jota käyttämällä voidaan tehdä testeissä http-pyyntöjä backendiin [12]. Käyttämällä Supertest-kirjastoa Jestin kanssa voidaan testata rajapinnan toiminnallisuutta ja esimerkiksi määriteltyjen reittien toimintaa.

Describe-lohkot auttavat luomaan selkeyttä testien rakenteeseen. Testien suorituksen yhteydessä tulosteiden tulee olla selkeitä ja helposti tulkittavissa. Jest kertoo käyttäjälle mitkä testit onnistuvat, mitkä epäonnistuvat ja mistä syystä. Esimerkiksi jos taulukon sisältö poikkeaa odotetusta, tulostuu myös taulukon sisältö konsoliin kuvassa 16 esitettyyn tapaan.

```
C:\Users\San\VAMK\Quiz_backend>npm run test

> quiz_backend@0.0.1 test
> cross-env NODE_ENV=test jest --verbose --runInBand --detectOpenHandles

 PASS  tests/quiz_api.test.js (8.418 s)
  when there is initially some quizzes saved
    √ correct number of quizzes is returned (2262 ms)
    √ the idenfitying field for a quiz is id and not _id (568 ms)
  when a post request is made
    √ quizzes with valid data can be added to the database successfully (452 ms)
    √ if the added quiz is not given a value for completedAt it is set to zero (1211 ms)
    √ if the added quiz is not given a value for title, a response is invoked with status code 400. (885 ms)
    √ quizzes cannot be added if the token is missing or invalid (460 ms)
  when there is initially one user at db
    √ creation succeeds with a fresh username (291 ms)
    √ correct number of users is returned (125 ms)
  when a post request is made to add a user
    √ if necessary data is missing or faulty, the user cannot be added (946 ms)
    √ user related quizzess and quiz related users are defined and populated correctly. (230 ms)

Test Suites: 1 passed, 1 total
Tests:       10 passed, 10 total
Snapshots:   0 total
Time:        8.495 s, estimated 20 s
Ran all test suites.
```

Kuva 16. Esimerkki testien suorituksesta Jestillä ja lopputuloksesta kertovasta tulosteesta konsolissa.

2.8 TDD (Test Driven Development)

TDD on ohjelmistokehitystekniikka, jossa jatkuva testaaminen on tärkeässä roolissa. Siinä luodaan testitapaus uudelle toiminnallisuudelle ennen varsinaisen

koodin kirjoittamista. Näin varmistutaan, että ohjelma toimii odotetulla tavalla kehitysprosessin aikana.

Jos oltaisiin esimerkiksi luomassa kirjautumistoiminto käyttäjälle, voitaisiin kirjoittaa testi, joka vertaa käyttäjätunnusta ja salasanaa tietokannasta löytyviin tietoihin. Jos testiä ajetaan ilman että toimintoa on ohjelmoitu, testi epäonnistuu. Tarkoituksena onkin testata jatkuvasti kehitettävää ohjelmakoodia ja puutteiden ilmetessä korjata sitä niin kauan, että luotu toiminto läpäisee testin kun sille annetaan oikeat käyttäjätiedot.

2.9 DRY & KISS (Don't Repeat Yourself) (Keep It Simple, Stupid)

Ohjelmistokehityksestä puhuttaessa usein käytettyjä termejä ovat mm. ”KISS” ja ”DRY”. KISS-periaatteen ideana on pitää koodin kielellinen asu sekä toiminnallinen logiikka niin selkänä ja yksinkertaisena kuin mahdollista mm. muuttujien ja funktioiden selkeällä ja kuvaavalla nimeämisellä. DRY-periaate taas viittaa samankaltaista toiminnallisuutta sisältävän koodin kirjoittamisen välttämiseen, esimerkiksi luomalla uusi funktio toiminnallisuuden toteuttamista varten ja kutsumalla sitä aina kun tarpeellista [13.]

Esimerkiksi, jos kehitettäisiin autokaupan omistajalle verkkosivuja, muuttuja ”AuToNMERkinNiMiIlMAnMaLLia” tai ”AM”, ei olisi KISS-periaatteen mukainen, eikä ”EtSiAutoMerKilläJaMalliLlaTietoKannasta()” tai ”EAMJMTK()” olisi KISS-periaatteen mukainen funktion nimi. Muuttujat ”autonMerkki”, ”autonMalli” ja funktio ”Etsi()” tai ”EtsiMerkilläJaMallilla()” taas noudattaisivat KISS-periaatetta. KISS-periaatteessa on kyse paljon muustakin kuin muuttujien tai funktioiden nimistä ja esimerkiksi jokaisen erimerkkisen auton määrien laskeminen erikseen ei noudata DRY-periaatetta, kun sitä varten voitaisiin luoda yksi uudelleenkäytettävä funktio, joka saa parametrinä auton merkin ja palauttaa sen merkkisten autojen määrän.

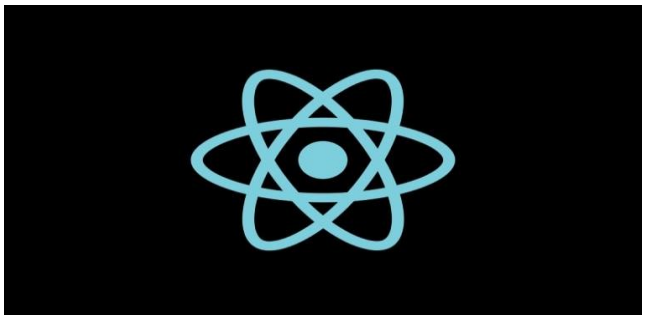
3 FRONTEND REACTILLA

3.1 Yleistä Reactista

React on avoimen lähdekoodin JavaScript kirjasto, joka on tarkoitettu frontend kehitykseen. [14] Vuonna 2011 Facebookille työskentelevä insinööri Jordan Walke kehitti FaxJS-nimellä tunnetun prototyypin Reactista. Vuonna 2013 Reactista tuli avoimen lähdekoodin kirjasto.

React itsessään on hyödyllinen työkalu, mutta ydinkirjaston lisäksi tarjolla on myös paljon kirjastoja, jotka tukevat React-kehitystä monin eri tavoin. On olemassa myös React Native-ohjelmistokehys, jota käytetään Android-sovellusten luomiseen. Cypress-kirjaston avulla voi toteuttaa ns. end-to-end testausta, eli koko sovelluksen toiminnan testaamista frontendistä backendiin ja tietokanta-operaatioihin.

Redux-kirjastolla saa aikaan tehokkaampia tilanhallintaratkaisuja. Näiden lisäksi mm. Apollo Client-kirjasto yhdessä Apollo Serverin kanssa mahdollistaa GraphQL-kyselykielen käyttämisen tiedonkäsittelyyn frontendissä ja backendissä perinteisen RESTful-lähestymistavan sijaan tiedonkulun tehostamiseksi. React, jonka logo esiintyy kuvassa 17 sallii CSS-tyylien käyttämisen ja ns. inline-tyylien määrittelyn, mutta React bootstrap-kirjasto tarjoaa valmiiksi määritellyillä tyyleillä räätälöityjä React-komponentteja parantamaan frontendin ulkoasua.



Kuva 17. Reactin logo [15].

Itse Reactissa kuitenkin yksi tärkeimmistä asioita on komponentit. Ne ovat uudelleenkäytettäviä JavaScript-funktioita, jotka voivat sisältää muuttujia ja funktioita ja palauttavat HTML-koodia, sekä exportataan eli viedään tarpeen ollen omasta tiedostostaan muiden komponenttien importattavaksi eli käyttöönotettavaksi/tuotavaksi. Komponentit ovat avainasemassa minkä tahansa frontendin luomisessa Reactilla.

Javan tyypitetyn määrittelyn sijaan JavaScriptissa voidaan muuttuja määritellä const-, let- tai var-muuttujaksi. Const tarkoittaa vakiomuuttujaa, jolle ei enää alustamisen jälkeen voi asettaa arvoa. Let- ja var-muuttujille voi antaa uuden arvon alustamisen jälkeen ja ne eroavat toisistaan siten että let-muuttuja on käytettävissä vain lohkossa, jossa se on määritelty, kun taas var-muuttuja koko funktiossa.

Yksi tärkeä asia Reactissa on myös propsit. Ns. parent-komponentti, jossa komponentti renderöidään, voi välittää muuttujia ja funktoita renderöitäville komponenteille ns. propsien avulla. Kuvan 18 tapaan määriteltyjen propsien arvojen muuttuessa myös vastaanottava komponentti renderöidään uudelleen ja se vastaanottaa uudet arvot.

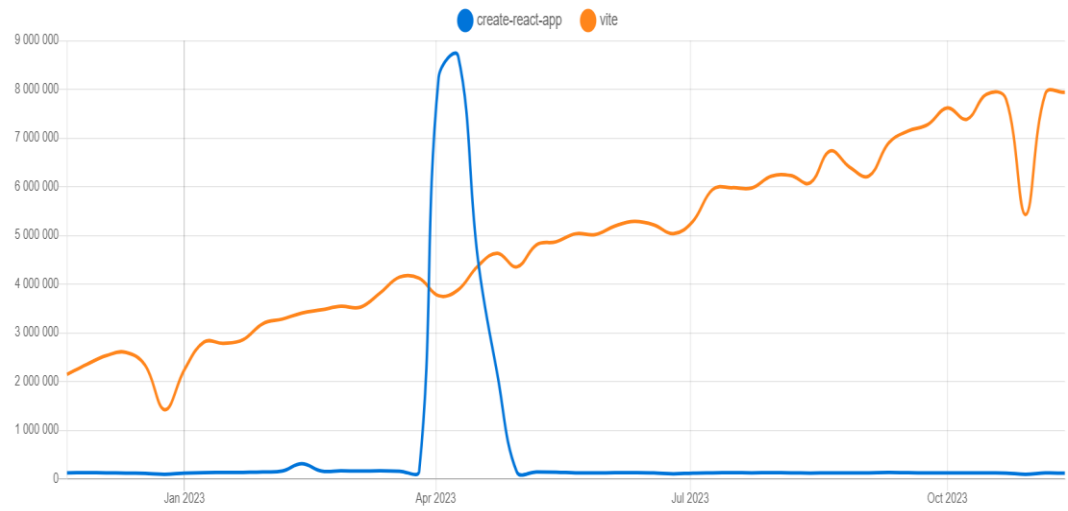
```
const LoginForm = ({ loginMethod }) => {

    const username = useField('text')
    const password = useField('password')

    //Defining a method to handle logging in and resetting the input fields.
    const handleLogin = async (event) => {
        event.preventDefault()
        await loginMethod(username.objectProps.value, password.objectProps.value)
        username.reset()
        password.reset()
    }

    return (
        <form onSubmit={handleLogin}>
            <h2>Log in to Quizmos: </h2>
            <div>
                Username:       
                <input name='userInput' {...username.objectProps} />
            </div>
            <br />
            <div>
                Password:       
                <input type='password' name='pwdInput' {...password.objectProps} />
            </div>
            <br />
            <div>
                <button id='loginBtn' type="submit">Login</button>
            </div>
        </form>
    )
}

//Defining prop validation for the LoginForm component with PropTypes.
LoginForm.propTypes = {
    loginMethod: PropTypes.func.isRequired
}

export default LoginForm
```

Kuva 18. Esimerkki loginMethod-propsin vastaanottavasta React komponentista LoginForm.

3.2 Vite

Viten julkaisuun vuonna 2020 asti käytössä React sovellusten luomiseksi oli create-react-app-työkalu. Sen asennus onnistui helposti npm install-komennolla, samoin kuin React projektin luominen create-react-app-työkalun avulla. Myös Viten avulla React-projektin luominen onnistuu helposti yhdellä komennolla.

Sen lisäksi, että Vite on pääsääntöisesti nopeampi kuin Webpackia käyttävä create-react-app, Viteen kuuluu myös valmis tuki JSX-syntaksille muiden joukossa. Create-react-app ei ole enää suositeltu työkalu React-sovellusten luomiselle. Viten suosio on kasvanut tasaista tahtia, kun taas create-react-app-työkalun suosio on romahtanut, kuten kuvan 19 käyrä osoittaa.



Kuva 19. Create-react-app työkalun ja Viten lataukset viimeisen vuoden aikana [16].

3.3 Hooks

Reactissa eräs tärkeä ominaisuus on myös tilanhallinta. Tämä tapahtuu funktionaalisessa komponentissa hook-komentojen avulla. Luokkakomponenteissa tilanhallinta tapahtuu tilamuuttujien sijaan komponentin tilaa ”this.state” muuttamalla.

Reactissa on useita hook-komentoja, mutta eniten käytetyt ovat useState ja useEffect. useState-hook luo tilamuuttujan ja metodin, jolla on mahdollista hallita sen tilaa, kun taas useEffect-hook suorittaa uudelleenrenderöintejä tarpeen mukaan. Valmiiden hook-komentojen lisäksi Reactissa on myös mahdollisuus määritellä omia räätälöityjä hook-komentoja.

Tilamuuttuja on siis muuttuja, jolla on tila ja metodi, jonka avulla tilaa voi muuttaa. Tilamuuttujan tilan muuttaminen aiheuttaa komponentin uudelleenrenderöinnin. Jos muuttujaan on tarpeen saada tallennettua uudelleenasetettava arvo ja sen muutoksen tulee renderöityä käyttäjälle välittömästi ilman koko sivun uudelleenlatausta selaimessa, tilamuuttuja osoittautuu hyödylliseksi.

UseEffectin käyttämiseen on olemassa useampia tapoja. Sen ensimmäisenä parametrinä annetaan toteutettava funktio ja toisena parametrinä ehdot, joiden täyttyessä funktio toteutetaan. On mahdollista esimerkiksi toteuttaa funktio vain ensimmäisen renderöinnin yhteydessä kuten kuvan 20 esimerkissä, jokaisen renderöinnin yhteydessä tai muuttujan arvon muuttuessa.

```
const Quiz = ({ quiz, mydisplay, handleDelete }) => {

    /*Defining a "state variable" for topics array and a variable for using
    the useNavigate hook. */
    const [topics, setTopics] = useState([])
    const navigate = useNavigate()

    const topicsArr = []

    /*Using the topicsArr variable and the topics state variable to
    map the topics of all questions into a single array. No topic should
    be present in the array twice. */
    useEffect(() => {
        quiz.questions.map(question => {
            if (!topicsArr.includes(question.topic)) {
                const topic = question.topic
                topicsArr.push(topic)
            }
        })
        setTopics(topicsArr)
    }, [])
```

Kuva 20. Esimerkki tilanhallinnasta ja useEffectin käytöstä. Toisena parametrina oleva tyhjä taulukko tarkoittaa, että funktio suoritetaan vain ensimmäisen renderöinnin yhteydessä. Tilamuuttuja topics taas alustetaan tyhjällä taulukolla.

3.4 Axios

Backendin kanssa kommunikoinnin toteuttamiseen on useampia vaihtoehtoja, joista yksi on Axios-kirjaston käyttäminen. Sen avulla voidaan tehdä esimerkiksi GET- tai POST-metodia käyttäviä http-pyyntöjä backendissä määritettyjä reittejä vastaaviin urleihin. GET-metodin avulla voidaan hakea tietoa palvelimelta ja POST-metodin avulla voidaan luoda uutta tietoa palvelimella, mutta on olemassa myös muita usein käytettyjä metodeja kuten PUT ja DELETE.

PUT-metodia käytetään, kun päivitetään palvelimella olevaa tietoa, esimerkiksi kun yrityksen verkkosivuja selaava asiakas haluaa muuttaa palvelimelle asiakastietoihin varastoitua sähköpostiosoitetta tai salasanaa. DELETE-metodia taas käytetään, kun halutaan poistaa palvelimelta tietoa. Sekä PUT- että DELETE-metodit tarvitsevat pääsääntöisesti muokattavan resurssin id-tunnistekentän arvon parametrina ja POST tarvitsee luotavan olion tiedot, kuten kuvan 21 esimerkissä.

```
//Defining a baseUrl variable for the request.
const baseUrl = '/api/quizzes'

let token = null

//Defining a function to set the value of the token variable.
const setToken = newToken => {

    token = `Bearer ${newToken}`
}

/*Using axios.get method to get all existing quizzes
from the Node backend and returning the response. */
const getQuizzes = async () => {
    const response = await axios.get(baseUrl)
    return response.data
}

/*Using axios.get method to get a single existing quizzes
from the Node backend with an id received as a parameter
and returning the response. */
const getQuiz = async (id) => {
    const response = await axios.get(`${baseUrl}/${id}`)
    return response.data
}

/*Using axios.post method to send a quiz object to the Node backend
and returning the response. A jsonwebtoken is also sent to the backend
in the Authorization header for authorization and the Content-Type header is
altered to allow multer to recognize image data. */
const createQuiz = async (quiz) => {

    const config = {
        headers: {
            Authorization: token,
            'Content-Type': 'multipart/form-data'
        }
    }

    const response = await axios.post(baseUrl, quiz, config)
    return response.data
}
```

Kuva 21. Esimerkki Axios-kirjaston käytöstä GET- ja POST-pyyntöjen määrittelyyn.

3.5 SPA (Single Page Application)

SPA eli yhden sivun sovellus on toteutus, joka ei lataa uutta sivua navigoitaessa sivulla. Reactilla ja Nodella luodut SPA:t toimivat siten että tietoa haetaan http-

pyynnöillä backendistä ja painettaessa esimerkiksi linkkiä sivulla renderöidään haettu tieto sitä varten määritellyssä komponentissa käyttäjälle frontendissä. Tämän ansiosta vältytään uuden sivun lataamiselta serveriltä.

Tunnettuja yhden sivun sovelluksia ovat mm. Gmail, Google Maps, Airbnb, Netflix, Pinterest ja PayPal [17]. Yhden sivun sovellusten käyttäjäystävällisyys perustuu niiden käytön nopeuteen ja tehokkuuteen. Vain tarpeellinen tieto ladataan backendistä esimerkiksi linkkiä klikatessa.

3.6 React Router

React Router on Reactilla luodussa frontendissä navigoinnin avuksi tarkoitettu kirjasto. Se tarjoaa esimerkiksi useNavigate- ja useParams-hookit, joilla voidaan poimia parametrinä annettuja id-arvoja urlista tai siirtyä tiettyyn urliin esimerkiksi painiketta painettaessa. React Routerin avulla on myös mahdollista luoda sivulle linkkejä ja reittejä ja määritellä komponentit, jotka renderöidään reittiä vastaavaan urliin siirryttäessä kuvan 22 esimerkin mukaisesti.

```
/*Returning links with conditional rendering depending on the logged in user.
Also returning routes related to these links. "Home page view" is set to be the list of
available quizzes. */
return (
    <div>
        <div>
            <Link to='/quizzes'>All quizzes</Link>
            {user ? <Link style={padding} to='/myquizzes'>My quizzes</Link> : null}
            {!user ? <Link style={padding} to='/login'>Login</Link> : null}
            {!user ? <Link style={padding} to='/signup'>Register</Link> : null}
            {user ? <Link style={padding} to='/create'>Create a quiz</Link> : null}
            {user ? <Link style={padding} to='/' onClick={logOut}>Logout</Link> : null}
            {user ? <p>Logged in as {user.username}</p> : null}
            <br />
            <br />
            <Notification message={message} isError={error} />
        </div>
        <Routes>
            <Route path='/' element={<HomePage />} />
            <Route path='/quizzes' element={<QuizList all={true} successMsgMethod={setSuccessMessage}
                errorMsgMethod={setErrorMessage} />} />
            <Route path='/myquizzes' element={<QuizList all={false} successMsgMethod={setSuccessMessage}
                errorMsgMethod={setErrorMessage} />} />
            <Route path='/login' element={<LoginForm loginMethod={loginFnct} />} />
            <Route path='/signup' element={<SignUpForm registerMethod={signUp} successMsgMethod={setSuccessMessage}
                errorMsgMethod={setErrorMessage} />} />
            <Route path='/create' element={<CreateQuizForm errorMsgMethod={setErrorMessage} user={user}
                successMsgMethod={setSuccessMessage} createNew={true} />} />
            <Route path='/edit/:id' element={<CreateQuizForm errorMsgMethod={setErrorMessage} user={user}
                successMsgMethod={setSuccessMessage} createNew={false} />} />
            <Route path='/play/:id' element={<Play user={user}/>} />
        </Routes>
    </div>
)
```

Kuva 22. Esimerkki reittien ja linkkien määrittelystä React Routerin avulla.

Edellämainittujen lisäksi React Router tarjoaa myös esimerkiksi useMatch-hookin. Sen avulla voi verrata annettua polkua tämänhetkiseen urliin. Käytössä on myös useLocation-hook, joka antaa tietoja käyttäjän tämänhetkisestä urlista.

3.7 Local storage

Kun sivu uudelleenladataan, kaikki tilamuuttujien arvot resetoituvat eli saavat oletusarvon. Tiedon pysyvämpää tallennusta varten voidaan kuitenkin käyttää esimerkiksi local storagea, jossa tallennettu tieto säilyy myös sivun uudelleenlatauksen jälkeen. Tämä ominaisuus on hyödyllinen esimerkiksi kirjautuneen käyttäjän tietojen säilyttämiseen kuten kuvan 23 esimerkissä.

```
/*Defining a method for logging in. If the
login is successful the useNavigate hook is used
to go to the "quiz list view". If not an error message is displayed. */
const loginFnct = async (username, password) => {

    try {
        const user = await login({ username, password })
        setUser(user)
        window.localStorage.setItem('loggedUserData', JSON.stringify(user))
        QuizService.setToken(user.token)
        navigate('/quizzes')
    } catch (exception) {

        setMessage(exception.response.data.error)
        setError(true)
        setTimeout(() => {
            setMessage('')
            setError(false)
        }, 3000)
    }
}

/*Defining a method for logging out. The user "state variable"
is set to null and the loggedUserData item is removed from localStorage. */
const logOut = () => {

    setUser(null)
    window.localStorage.removeItem('loggedUserData')

}
```

Kuva 23. Esimerkki local storagen käytöstä kirjautumis- ja uloskirjautumisfunktioissa.

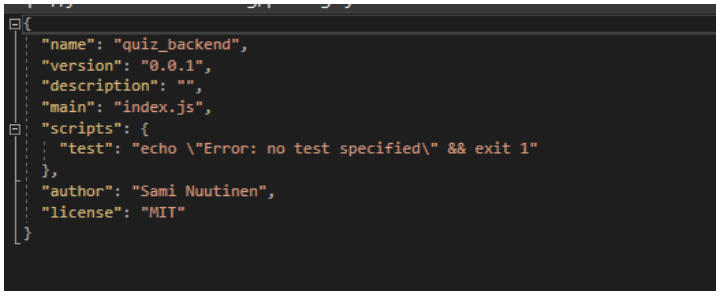
Komennolla window.localStorage.setItem voidaan tallentaa local storageen avain-arvo-pari. Komennolla window.localStorage.removeItem voidaan poistaa local storagesta haluttu avain-arvo-pari. React Nativessa voidaan local storagen sijaan käyttää samaa tarkoitusta palvelevaa AsyncStorage API-vaihtoehtoa.

4 TIETOVISASOVELLUS - BACKEND

Nodea ja Reactia käyttämällä voidaan luoda web-sovellus JavaScriptilla. Esimerkiksi sovellus, joka mahdollistaa räätälöityjen tietovisojen luomisen ja pelaamisen. Perusteellinen perehtyminen full stack -kehitykseen onnistuu parhaiten käytännön kautta, joten tässä ja seuraavassa osiossa kuvataan tietovisasovelluksen luomisen vaiheita.

4.1 Backendin luominen

Koska frontendin tarkoituksena on käsitellä backendiltä saatua tietoa, full stack -kehitys on hyvä aloittaa backendin luomisesta. Komennolla npm init luodaan package.json tiedosto, joka sisältää projektin määrittelyt. Package.json tiedostossa ”main”-tiedostoksi on määritelty index.js kuten kuvassa 24 on esitetty, joten luodaan kyseinen tiedosto.



Kuva 24. Npm init komennolla luodun package.json tiedoston sisältö.

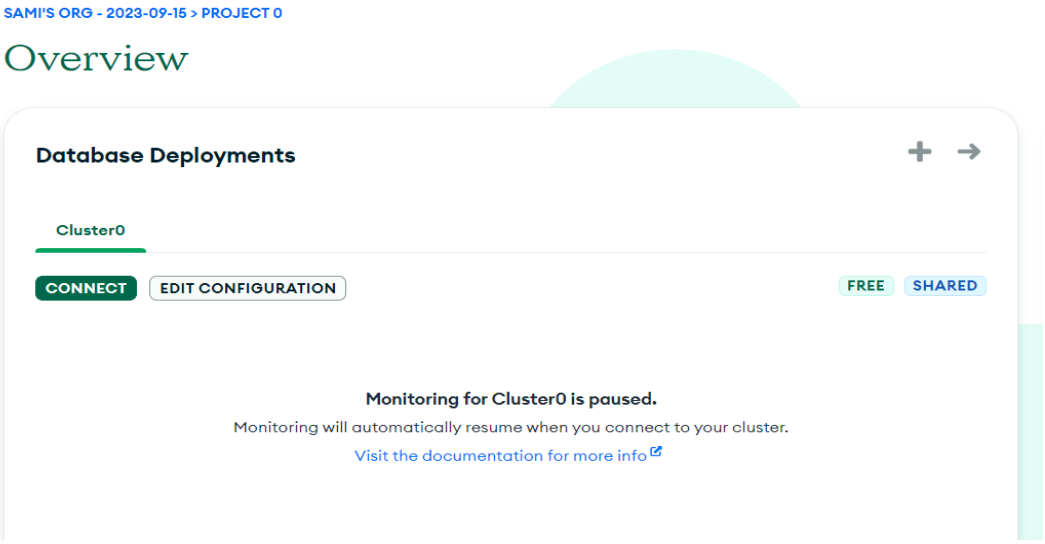
Asennetaan valmiiksi express, dotenv, cors, jest, supertest ja muut kirjastot, joita tullaan tarvitsemaan. Asennetaan myös nodemon-työkalu, joka helpottaa backend-kehitystä Nodella käynnistämällä sen automaattisesti uudelleen, kun koodiin tehdään muutoksia. Kun nodemon on asennettu on lisätään sille myös oma skripti package.json tiedostoon kuvan 25 mukaisesti.

```
"scripts": {
  "start": "node index.js",
  "dev": "nodemon index.js",
  "test": "echo \"Error: no test specified\" && exit 1"
},
```

Kuva 25. Lisätyt skriptit package.json tiedostossa.

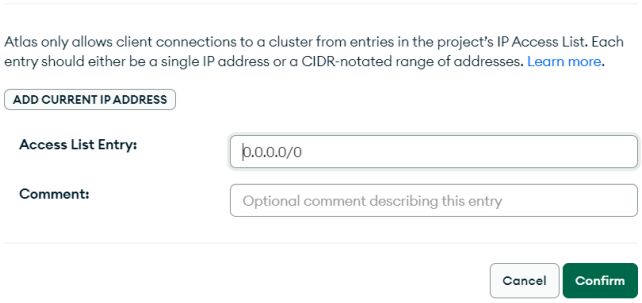
4.2 MongoDB tietokannan luominen

Seuraava tärkeä asia on tietokannan luominen. Tähän tarkoitukseen käytetään MongoDB:tä. Ensin luodaan käyttäjä MongoDB:seen ja käyttäjälle kuvan 26 kaltainen ”klusteri”, johon sovellus voidaan yhdistää.

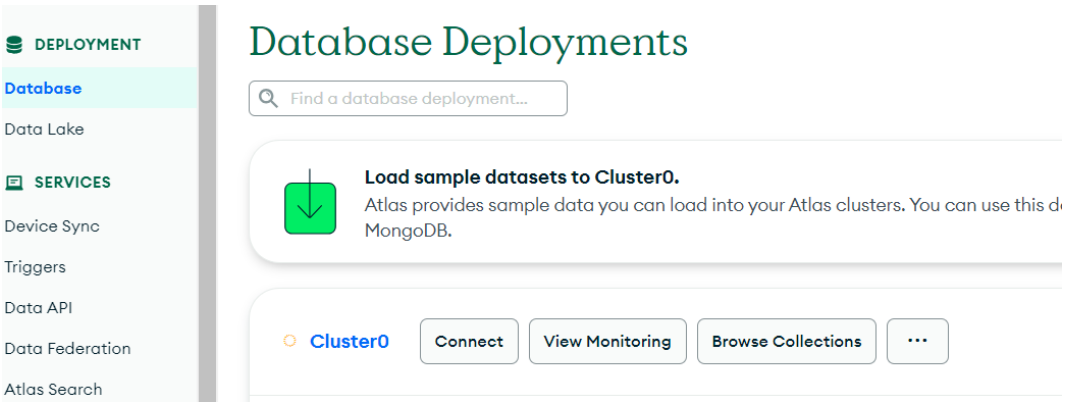


Kuva 26. MongoDB klusteri. Connect nappia painamalla voidaan valita ajuri ja selvittää uri, johon tietokantayhteys luodaan.

”Network access”-kohdasta voidaan määrittää kuvan 27 tapaan IP-osoitteet, joista yhteyden muodostaminen sallitaan. Klusteriin kuuluvia kokoelmia voi hallita kuvassa 28 esitetyn painikkeen avulla ja kohdasta ”Database access” voidaan tietokannalle luoda myös käyttäjä ja salasana. Näitä käytetään backendissä ympäristömuuttujien ja dotenv-kirjaston avulla, kun luodaan sovellukseen MongoDB-yhteys urin ja kuvan 29 tapaan sen osoittaman kokoelman avulla.



Kuva 27. Osoitteen 0.0.0.0/0 salliminen sallii pääsyn mistä tahansa IP-osoitteesta.



Kuva 28. Painamalla "Browse Collections"-painiketta päästään selaamaan klustereiden sisältämiä kokoelmia ja niiden sisältöä sekä luomaan uusia.

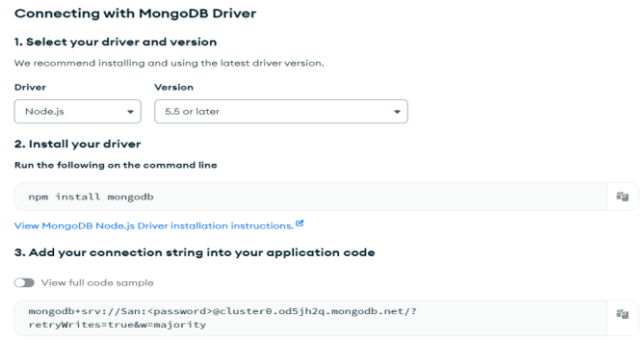
```
mongodb.net/quizApp?retryWrites=true&
```

Kuva 29. Käytettävän kokoelman nimi tulee määrittää .env-tiedoston MONGODB_URI-muuttujan merkkijonossa ennen kysymysmerkkiä quizApp-tekstin kohdalle.

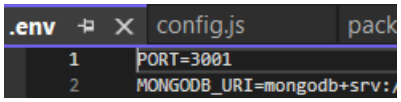
4.3 Ympäristömuuttujien määrittely

Luodaan .env-tiedosto projektin juureen eli index.js-tiedoston sisältävään kansioon. Kuvan 30 esimerkin kaltainen, MongoDB-yhteyden luomista varten määritelty URI tallennetaan ympäristömuuttujaan luodussa .env-tiedostossa kuvan 31 tapaan ja sen password-kohta korvataan ”Database access”-linkin kautta luodun tietokantakäyttäjän salasanalla. Ympäristömuuttujat otetaan käyttöön koodissa kuvan 32 esimerkin mukaisesti, mutta jos salasana sisältää erikoismerkkejä on

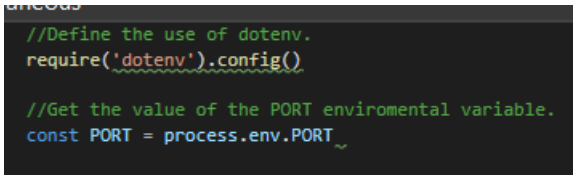
tarpeen käyttää esimerkiksi EncodeURIComponent-komentoa ja liittää näin saatu merkkijono uudelleen uriin ennen yhdistämistä.



Kuva 30. MongoDB-klusteriin yhdistämistä varten määritelty uri.



Kuva 31. PORT- ja MONGODB_URI-ympäristömuuttujien määrittely .env-tiedostossa.



Kuva 32. Ympäristömuuttujien käyttäminen koodissa.

Ympäristömuuttujia ei ole tarkoitus ladata esimerkiksi Githubiin ja tästä voidaan huolehtia .gitignore-tiedoston avulla. Luodaan kansio utils ja sinne tiedosto config.js, jossa otetaan dotenv käyttöön, muodostetaan uusi MONGODB_URI-muuttuja, jossa erikoismerkit on otettu huomioon. Kun salasana on lisätty osaksi uutta MONGODB_URIa EncodeURIComponent-komennon avulla, viedään uusi muuttuja sekä PORT-muuttuja, jotta niitä voidaan käyttää muualla sovelluksessa kuvan 33 tapaan.

```
//Form an URI to connect by using the different strings.
const MONGODB_URI = `${usrstring}${encodedstring}@${splitsecond[pwdindex]}`

//Export the values of the MONGODB_URI and PORT variables.
module.exports = {
    MONGODB_URI,
    PORT
}
```

Kuva 33. Muuttujien MONGODB_URI ja PORT exportointi eli vieminen tiedostosta config.js.

4.4 Palvelimen luominen Expressillä

Luodaan tiedosto app.js ja lisätään siihen yhteyden luominen MongoDB:seen. Testaamista varten on hyvä olla erikseen tiedostot index.js ja app.js. Yhteyden muodostaminen voidaan toteuttaa ottamalla luotu config.js-tiedosto käyttöön app.js-tiedostossa kuvan 34 tapaan ja määritellään yhteys käyttöönotetun config.js-tiedoston MONGOBD_URI-ympäristömuuttujan avulla kuvan 35 esimerkin mukaisesti.

```
const config = require('./utils/config')
const express = require('express')
const mongoose = require('mongoose')
const cors = require('cors')
const app = express()
```

Kuva 34. app.js-tiedostossa määriteltyjä importtauksia eli käyttöönottoja.

```
console.log('connecting to', config.MONGODB_URI)

mongoose.connect(config.MONGODB_URI)
    .then(() => {
        console.log('connected to MongoDB')
    })
    .catch((error) => {
        console.log(`error connection to MongoDB: ${error.message}`)
    })
```

Kuva 35. Yhteyden luominen MongoDB:seen app.js-tiedostossa.

Kun on lisätty vielä mm. muutama app.use määrittely tiedostoon app.js kuvan 36 tapaan, exportoidaan app-muuttuja tiedoston lopussa. Määritellään kuunneltava portti tiedostossa index.js tiedoston config.js sekä PORT-ympäristömuuttujan avulla kuvan 37 esimerkin mukaisesti ja otetaan käyttöön app.js. Projektikansio on kuvan 38 esimerkin mukainen nyt kun tietokantayhteys on määritelty ja pal-

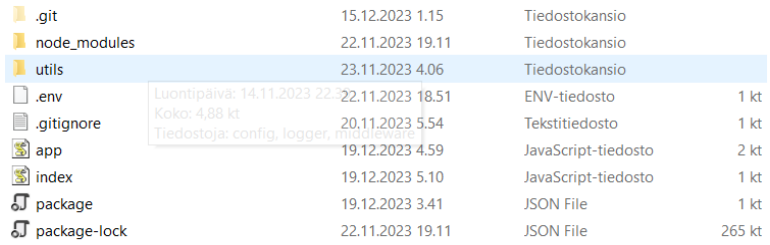
velin kuuntelee ympäristömuuttujan osoittamaa porttia, mutta backend ei sisällä vielä yhtäkään skeemaa eikä reittiä.

```
app.use(cors())
app.use(express.static('build'))
app.use(express.json())
```

Kuva 36. app.use määrittelyt app.js-tiedostossa.

```
app.listen(config.PORT, () => {
  console.log(`Server running on port ${config.PORT}`)
})
```

Kuva 37. Kuunneltavan portin määrittely tiedostossa config.js määritellyn muuttujan PORT avulla ja käyttöön otetun app.js tiedoston avulla.



Kuva 38. Projektikansion sisältö.

4.5 Skeemojen määrittely ja validointi

Skeemojen määrittelyssä ensimmäinen askel on miettiä, mitä sovelluksessa on tarpeen tallentaa luotuun MongoDB-tietokantaan. Käyttäjien tiedoille, yksittäisen tietovisan tiedoille ja kysymyksille on oltava oma skeemansa. Luodaan siis projektin juureen kansio models, johon luodaan user.js-, quiz.js- ja question.js-tiedostot.

Vaikka vastaukset voitaisiin tallentaa myös taulukkona merkkijonoja, luodaan myös niille tiedostoon answer.js erillinen skeema ja tallennetaan ne tietokantaan omina olioinaan. Luodaan myös performance.js-tiedosto ja siihen skeema yksittäistä visan suorituskertaa koskevia tietoja varten. Näin voidaan luoda myöhemmin mahdollisuus esimerkiksi top 5-listojen tarkasteluun frontendissä.

Asennetaan Mongoose kirjaston lisäksi myös mongoose-unique-validator-plugin käyttäjänimen ainutlaatuisuuden tarkistuksen helpottamiseksi käyttäjää luotaessa. Jotkut mongoose-unique-validator versiot saattavat aiheuttaa yhteensopivuus ongelmia joidenkin Mongoose versioiden kanssa. Mongoosen asennuksen poistaminen ja uudelleenasentaminen kuvan 39 esimerkin mukaisesti korjaa ongelman, jos asennuksessa määritelty versio on yhteensopiva kuten kuvan 40 package.json-tiedostossa määritellyt versiot.

```
>npm install mongoose@7.6.5
```

Kuva 39. Asennettavan mongoose version määrittely komentoriviltä.

```
"dependencies": {
  "bcrypt": "^5.1.1",
  "cors": "^2.8.5",
  "dotenv": "^16.3.1",
  "express": "^4.18.2",
  "jsonwebtoken": "^9.0.2",
  "mongoose": "^7.6.5",
  "mongoose-unique-validator": "^4.0.0",
```

Kuva 40. Yhteensopivat mongoose ja mongoose-unique-validator versiot.

Mongoosen avulla voidaan ainakin osittain hoitaa myös validointi backendissä. Validoitaville kentille voidaan esimerkiksi määrittää tyyppi, onko kenttä vaadittu kenttä ja merkkijonon minimi- sekä maksimipituus. Jos käyttäjän syöttämät tiedot ovat virheellisiä, voidaan myös välittää virheen mukana mukautettu viesti kuten kuvan 41 esimerkin required- ja minlength-kenttien määrittelyissä.

```
const userSchema = mongoose.Schema({
    username: {
        type: String,
        required: [true, 'Username cannot have an empty value.'],
        minlength: [3, 'Username must contain at least 3 characters.'],
        unique: true
    },
    passwordHash: String,
    quizzes: [
        {
            type: mongoose.Schema.Types.ObjectId,
            ref: 'Quiz'
        }
    ],
})

//Defining the mongoose-unique-validator plugin.
userSchema.plugin(uniqueValidator, { message: 'Username is already in use.' })
```

Kuva 41. Käyttäjä skeeman validointien määrittelyt.

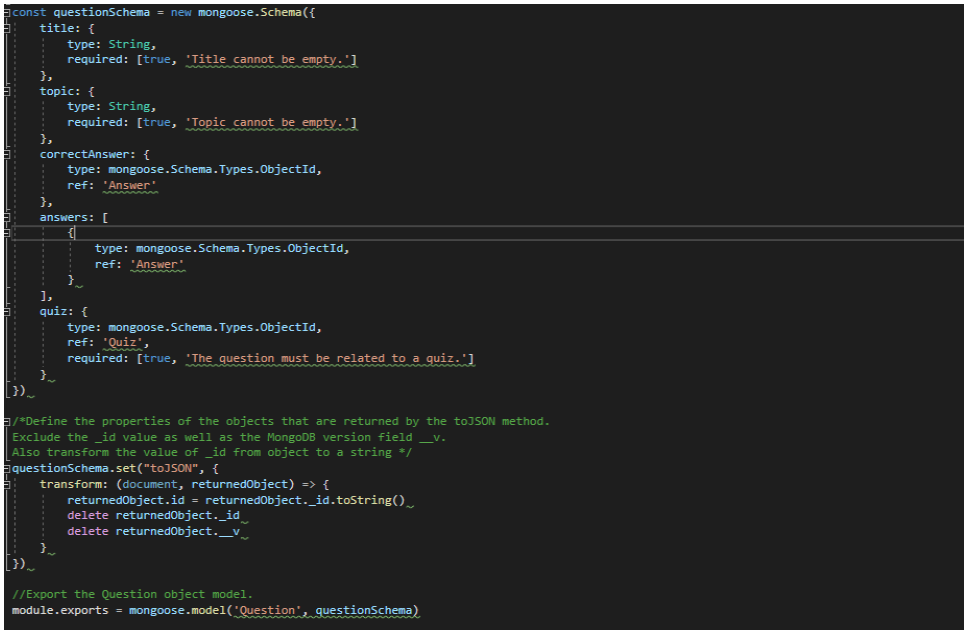
Käyttäjän tiedoille määritelty skeema sisältää käyttäjänimen lisäksi myös passwordHash kentän eli annetusta salasanasta muodostetun tiivisteen ja käyttäjän

luomiin visoihin viittaavan Quiz-olioiden tunnistekenttiä sisältävän quizzes-taulukon. Tiiviste muodostetaan reitin määrittelyn yhteydessä ja visaa tallennettaessa sen tunnistekenttä tallennetaan käyttäjän quizzes taulukkoon. Itse salasana syötteen validointia ei myöskään suoriteta backendissä, vaan se tehdään frontendissä.

Quiz-skeemalle annetaan kenttinä title-, difficulty-, completedAt-, highScore-, highScoreSetBy-, ratings-, questions-, author-, image- ja timeLimitPerQuestion-kentät. Difficulty-kentän arvona on numero, joka on oletusarvoisesti 0 kun visa luodaan ja muodostetaan myöhemmin ratings-taulukon käyttäjien arviointien keskiarvosta. CompletedAt-kenttä määrittää monenko oikean vastauksen jälkeen visa on suoritettu, highScoreSetBy- ja author- kentät sisältävät viitteen huippupisteet asettaneeseen käyttäjään sekä visan luojaan, kun taas questions sisältää taulukon viitteitä visan kysymysten tunnistekenttiin.

Image-kenttä sisältää käyttäjän lataaman, visoja selatessa näkyvän koristeellisen kuvan tiedostonimen. Kuvan lataaminen koneelta sovellukseen frontendissä on yksinkertaista, mutta sen backendiin tallentamista varten asennetaan Multer middleware. Kuvan tallentaminen Multerin avulla tapahtuu, kun määritellään reitti visan luomiseksi.

Koska tarkoituksena on, että pelaaja voi halutessaan karsia aihealueita visasta, kysymyksen skeemalla on oltava vähintään title-, topic-, answers-, quiz- ja correctAnswer-kentät. Answers-kenttä sisältää taulukon tunnisteviitteitä kysymyksen vastausvaihtoehtoihin, quiz-kenttä sisältää viitteen visaan, jolle kysymys on luotu ja correctAnswer-kenttä viitteen oikean vastausvaihtoehdon tunnisteeseen. Answers-skeema sisältää vain title- ja quiz-kentät ja performance-skeema score-, user- ja quiz-kentät ja kun skeema on määritelty se viedään sovelluksen muiden osien käyttöön kuvan 42 esimerkin mukaisesti.



```
const questionSchema = new mongoose.Schema({
    title: {
        type: String,
        required: [true, 'Title cannot be empty.']
    },
    topic: {
        type: String,
        required: [true, 'Topic cannot be empty.']
    },
    correctAnswer: {
        type: mongoose.Schema.Types.ObjectId,
        ref: 'Answer'
    },
    answers: [
        {
            type: mongoose.Schema.Types.ObjectId,
            ref: 'Answer'
        }
    ],
    quiz: {
        type: mongoose.Schema.Types.ObjectId,
        ref: 'Quiz',
        required: [true, 'The question must be related to a quiz.']
    }
})

/*Define the properties of the objects that are returned by the toJSON method.
Exclude the _id value as well as the MongoDB version field __v.
Also transform the value of _id from object to a string */
questionSchema.set("toJSON", {
    transform: (document, returnedObject) => {
        returnedObject.id = returnedObject._id.toString()
        delete returnedObject._id
        delete returnedObject.__v
    }
})

//Export the Question object model.
module.exports = mongoose.model('Question', questionSchema)
```

Kuva 42. Kysymys skeeman määrittely ja exportointi eli vieminen.

Visaa luotaessa ei haluta käyttäjän pystyvän luomaan kahta samannimistä visaa, visaan kahta samaa kysymystä tai kysymykselle kahta samaa vastausta. Tässä vaiheessa voidaan käyttää visalle unique-validatoria, mutta koska halutaan eri käyttäjien voivan luoda samannimisen visan tai kahden eri visan pystyvän sisältämään saman kysymyksen, validointi tehdään quizzes.js-tiedoston reittien määrittelyssä. Validointi voidaan myös tehdä esimerkiksi frontendin puolella.

4.6 Käyttäjänhallinta

Aloitetaan reittien määrittely luomalla projektin juureen controllers-kansio. Määritellään tiedosto login.js käyttäjien hallintaa varten, sekä performances.js-, quizzes.js-, questions.js- ja users.js-tiedostot reittien määrittelyä varten. Aloitetaan login.js- ja users.js-tiedostoista.

Käyttäjä skeeman määrittelevään tiedostoon on määriteltävä ainakin yksi reitti, mutta määritellään kolme. Määritellään aluksi Express-ohjelmistokehyksen avulla reittimäärittelyt sisältävä usersRouter, joka exportoidaan tiedoston loppuosassa. UsersRouterille määritellään yksi POST-metodia käyttävä reitti käyttäjän luomista

varten, eikä DELETE-metodia käyttävää reittiä käyttäjän poistamiseksi tällä hetkellä tarvita.

Tämän lisäksi määritellään yksi GET-metodia käyttävä reitti, joka palauttaa tietokannan kaikki käyttäjät ja toinen GET-metodia käyttävä reitti, joka hakee yksittäisen palautettavan käyttäjän id:n avulla. Id vastaanotetaan pyynnön mukana ja populate-metodin avulla palautetaan quizzes-taulukon sisällöstä title-kenttien arvot. POST-metodia käyttävän reitin määrittelyssä pyynnössä vastaanotetusta sanasanasta luodaan tiiviste bcrypt-kirjaston hash-funktion sekä määritellyn saltRounds-muuttujan avulla ja vain tiiviste tallennetaan tietokantaan.

Mitä isompi arvo saltRounds-muuttujalle annetaan, sitä turvallisempi tiivistefunktion palauttama tiiviste on. Tiivistefunktion toiminta ei ole yksiselitteistä ja liian suuren arvon antaminen saltRounds-muuttujalle pidentää tiivistealgoritmin vasteaikaa. Tämä taas haittaa sovelluksen toimintaa ja aiheuttaa käyttäjissä turhautumista.

Myös login.js-tiedostossa otetaan käyttöön bcrypt-kirjasto ja määritellään sekä exportoidaan loginRouter. Tämän lisäksi otetaan käyttöön jsonwebtoken-kirjasto, jonka avulla luodaan token käyttäjien todennusta ja valtuutusta varten. Sitä ei näytetä käyttäjille, mutta käytetään jatkossa esimerkiksi visojen poisto- ja lisäysoperaatioiden valtuuttamisessa.

Token luodaan ympäristömuuttujan SECRET ja käyttäjätietojen avulla. Ympäristömuuttujan SECRET sisältämällä merkkijonolla ei ole merkitystä. LoginRouterin POST-metodia käyttävällä reitillä varmistetaan annetun salasanan oikeellisuus bcrypt-kirjaston compare-funktion avulla ja palautetaan onnistuneen varmistuksen jälkeen jsonwebtoken-kirjaston metodilla sign luotu token vastauksen mukana kuvan 43 esimerkin mukaisesti.

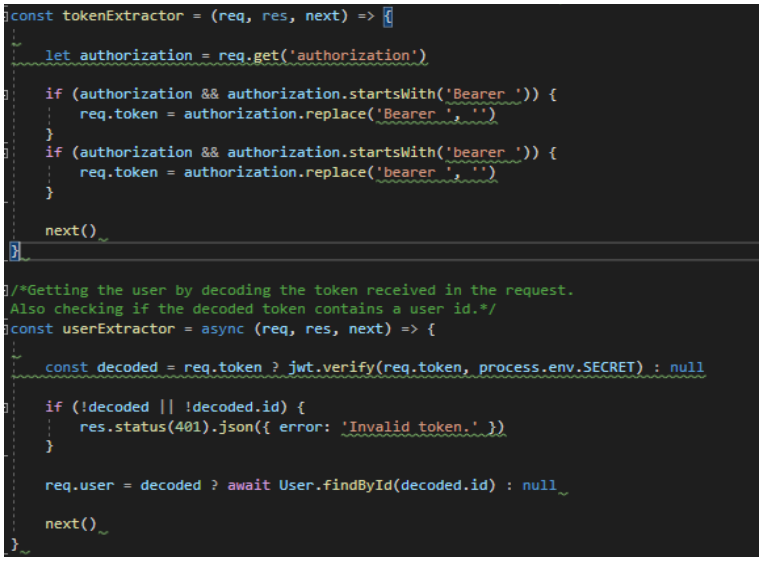
```
loginRouter.post('/', async (request, response) => {
    const { username, password } = request.body

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

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

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

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

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

Kuva 43. Salasanan tarkistus ja tokenin palauttaminen loginRouterin POST-metodia käyttävän reitin määrittelyssä.

4.7 Middlewaret ja Multer

Luodaan seuraavaksi utils-kansioon tiedosto middleware.js. Tähän tiedostoon voidaan määritellä middlewareja, joita voidaan käyttää pyyntöjä vastaanotettaessa. Määritellään middlewaret tokenExtractor ja userExtractor, joilla on req-, res- ja next-parametrit.

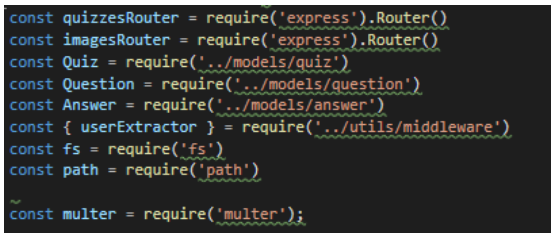
TokenExtractor middleware tarkistaa pyynnön Authorization otsikkokentän, karsii ylimääräisen tekstin, asettaa sen pyynnön token kentän arvoksi ja siirtää pyynnön käsittelyvastuun seuraavalle middlewarelle next parametrin avulla. UserExtractor-middleware vahvistaa pyynnön käsitellyn tokenin oikeellisuuden jsonwebtoken-kirjaston avulla, purkaa sen koodauksen ja asettaa tokenista saadun id:n avulla pyynnön user kentän arvoksi id:tä vastaavan käyttäjän tiedot. Middlewaret voidaan määritellä kuvan 44 esimerkin mukaisesti ja näiden middlewarejen avulla voidaan välttää saman koodin toistoa jokaisen tokenia käyttävän pyynnön määrittelyssä.



```
const tokenExtractor = (req, res, next) => {

    let authorization = req.get('authorization')

    if (authorization && authorization.startsWith('Bearer ')) {
        req.token = authorization.replace('Bearer ', '')
    }
    if (authorization && authorization.startsWith('bearer ')) {
        req.token = authorization.replace('bearer ', '')
    }

    next()
}

/*Getting the user by decoding the token received in the request.
Also checking if the decoded token contains a user id.*/
const userExtractor = async (req, res, next) => {

    const decoded = req.token ? jwt.verify(req.token, process.env.SECRET) : null

    if (!decoded || !decoded.id) {
        res.status(401).json({ error: 'Invalid token.' })
    }

    req.user = decoded ? await User.findById(decoded.id) : null

    next()
}
```

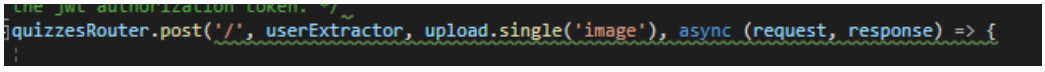
Kuva 44. TokenExtractor ja userExtractor middlewarejen määrittely.

Visan luomiseen tarvitaan myös asennettua Multer middlewarea kuvien tallentamiseksi. Luodaan tässä vaiheessa kansioon controllers uusi kansio uploads johon kuvat tallennetaan ja otetaan käyttöön multer quizzes.js-tiedoston alussa kuvan 45 tapaan. Koska on tarkoituksena, että vain kirjautunut käyttäjä voi luoda uuden tietovisan, on userExtractor middleware otettava käyttöön quizzes.js-tiedoston POST-metodia käyttävän reitin määrittelyssä kuvan 46 esimerkin mukaisesti.



```
const quizzesRouter = require('express').Router()
const imagesRouter = require('express').Router()
const Quiz = require('../models/quiz')
const Question = require('../models/question')
const Answer = require('../models/answer')
const { userExtractor } = require('../utils/middleware')
const fs = require('fs')
const path = require('path')

const multer = require('multer');
```

Kuva 45. Importtaukset eli käyttöönotot quizzes.js-tiedostossa.



```
the jwt authorization token. */
quizzesRouter.post('/', userExtractor, upload.single('image'), async (request, response) => {
```

Kuva 46. Middlewarejen käyttöönotto POST-metodia käyttävällä reitillä.

Kuvat voitaisiin tallentaa myös MongoDB:seen, mutta määritellään seuraavaksi Multeria käyttävä upload-metodi, joka tallentaa kuvat uploads-kansioon. Se käyt-

tää kuvan 47 esimerkin mukaisesti multer.diskStorage-metodia, jossa määritetään pyynnön mukana vastaanotetulle tiedostolle kohdesijainti sekä tiedostonimi. Tietokantaan tallennetaan vain tiedostonimi, jolla kuva voidaan hakea imagesRouterin GET-metodia käyttävän reitin määrittelyssä uploads-kansiosta pyynnön kentässä ”file.filename” vastaanotetun tiedostonimen avulla kuvan 48 esimerkin esittämällä tavalla.

```
const upload = multer({
    storage: multer.diskStorage({
        destination: (req, file, cb) => {
            cb(null, './controllers/uploads')
        },

        1 reference
        filename: function (req, file, callback) {
            callback(null, file.fieldname + '-' + Date.now() +
                path.extname(file.originalname))
        }
    })
});
```

Kuva 47. Upload metodin määrittely tiedostossa quizzes.js.

```
imagesRouter.get('/:fileName', function (request, res) {

    res.sendFile(`./uploads/${request.params.fileName}`, { root: __dirname });
});
```

Kuva 48. ImagesRouterin GET-metodia käyttävän reitin määrittely.

4.8 Reittien määrittely ja http-metodit

Lopulta quizzes.js-tiedoston tulee sisältää kaikki CRUD-operaatiot (Create, Read, Update, Delete) eli reitti kaikkien visojen ja yksittäisen visan hakemista varten GET-metodilla, reitti kuvan hakemista varten GET-metodilla, reitti visan poistamista varten DELETE-metodilla, reitti visan päivittämistä varten PUT-metodilla, reitti visan luomista varten POST-metodilla ja reitti kysymyksen lisäämistä varten POST-metodilla. DELETE-metodia käyttävän reitin tulee visan lisäksi poistaa myös kuva uploads-kansiosta ja poistettavaan Quiz-olioon viittaava tunniste visan luoneen käyttäjän quizzes-taulukosta JavaScriptin splice-metodin avulla kuvan 49 esimerkin tapaan. Kysymyksen visaan lisäävän reitin on erotuttava POST-metodia käyttävästä visan luovasta reitistä, joten polkua on muutettava ja sen tulee sisäl-

tää myös validointi, joka tuottaa itse määritellyn virheen, jos visa sisältää jo kysymyksen samalla otsikolla kuvan 50 esimerkin mukaisesti.

```
const quizToDelete = await Quiz.findById(request.params.id)

const user = request.user

if (user.id.toString() !== quizToDelete.author.toString()) {
    return response.status(401).json({ error: 'Unauthorized.' })
}

if (quizToDelete !== null && quizToDelete !== undefined) {

    if (quizToDelete.image !== null && quizToDelete.image !== undefined) {
        const filePath = `${__dirname}/uploads/${quizToDelete.image}`

        fs.unlink(filePath, (error) => {
            if (error) {
                response.status(404).json(error).end()
            }
        })
    }

    /*Deleting the quiz with the received id and removing the reference to
    the deleted quiz from the user.quizzes array and saving the user. */
    const index = user.quizzes.indexOf(quizToDelete.id)

    if (index > -1) {
        user.quizzes.splice(index, 1)
    }

    const questions = quizToDelete.questions

    await Answer.deleteMany({ question: { $in: questions } })
    await Question.deleteMany({ quiz: quizToDelete.id.toString() })

    await Quiz.findOneAndRemove(quizToDelete)

    await user.save()

    response.status(204).end()
} else {
    response.status(400).end()
}
```

Kuva 49. DELETE-metodia käyttävän reitin määrittely quizzes.js-tiedostossa.

```
quizzesRouter.post('/:id/questions', async (request, response) => {

    const body = request.body
    const quiz = await Quiz.findById(request.params.id).populate('questions', { title: 1 })

    const question = new Question({
        title: body.title,
        topic: body.topic,
        quiz: quiz.id,
    })

    /*If the quiz is found from the MongoDB database and the quiz does
    not already contain a question with the given title, it is saved and
    the id of the question is also saved to the quiz data as a reference. */
    if (quiz) {
        if (!quiz.questions.some((question) => question.title === body.title)) {
            try {
                const savedQuestion = await question.save()

                if (quiz.questions.length === 0) {
                    quiz.questions = quiz.questions[0] = savedQuestion._id
                    await quiz.save()
                } else {
                    quiz.questions = quiz.questions.concat(savedQuestion._id)
                    await quiz.save()
                }

                response.status(201).json(savedQuestion)
            } catch (error) {
                response.status(400).json(error).end()
            }
        } else {
            response
                .status(400)
                .json({
                    error: 'Quiz already contains a question with the given title.',
                })
                .end()
        }
    } else {
        response
            .status(404)
            .json({
                error: 'Quiz does not exist. It has possibly been deleted.',
            })
            .end()
    }
})
```

Kuva 50. POST-metodia kysymyksen tallentamiseen käyttävän reitin määrittely.

Määritellään myös questions.js-tiedostoon GET-metodia käyttävä reitti, joka hakee kaikki kysymykset, DELETE-metodia käyttävä reitti, joka poistaa yksittäisen

kysymyksen ja siihen liittyvät viitteen Quiz-oliosta, sekä POST-metodia käyttävä reitti, joka vastaanottaa id:n pyynnön mukana ja luo id:tä vastaavalle kysymykselle Answer-skeemaa käyttäen vastausvaihtoehdon. Haettaessa visaa, käyttäjää, tai kysymystä GET-metodilla on tärkeää myös palauttaa Mongoosen populate-metodin avulla viitteitä vastaavat oliot ja niiden tarpeelliset kentät. Määritellään vielä performances.js-tiedostoon yksi POST-metodia käyttävä reitti ja kolme GET-metodia käyttävää reittiä, jotta voidaan helposti hakea myös tietyn käyttäjän tai tiettyä visaa koskevia suorituksia.

4.9 Testien luominen

Nyt kun reitit ja skeemat on määritelty, luodaan vielä lopuksi muutama testitapaus kansioon tests. Testaamisessa voidaan käyttää apuna Supertest-kirjastoa ja Jest-ohjelmistokehystä. Luodaan tiedostot quiz_api.test.js, teardown.js kuvan 51 esimerkin sisällöllä ja test_helper.js, sekä lisätään package.json-tiedostoon Jestin käyttöä koskevia määrittelyjä kuvan 52 tapaan.

```
module.exports = () => {
    process.exit(0)
}
```

Kuva 51. teardown.js-tiedosto sisältää vain process.exit-komennon viennin.

```
"jest": {
  "testEnvironment": "node",
  "globalTeardown": "./tests/teardown.js"
}
```

Kuva 52. Jestiä koskevia määrittelyjä package.json-tiedostossa.

test_helper.js-tiedostoa voidaan esimerkiksi käyttää koodin toiston välttämiseen määrittelemällä siihen funktio tietokannan alustamista varten tai esimerkiksi funktio, joka hakee tietokannasta kaikki käyttäjät tai visat. Eriytetään luomalla uusi tietokanta testQuizApp, määrittelemällä ympäristömuuttuja TEST_MONGODB_URI, asentamalla cross-env-kirjasto ja muokkaamalla skriptejä kuvan 53 tapaan sekä config.js-tiedoston koodia tarkistamaan NODE_ENV-ympäristömuuttujan arvo ja yhdistämään oikeaan tietokantaan. quiz_api.test.js-

tiedostoon tuodaan tarvittavat kirjastot ja skeemat kuvat 54 esimerkin mukaisesti ja siinä määritellään backendille suoritettavat testit.

```
"scripts": {
  "start": "cross-env NODE_ENV=production node index.js",
  "dev": "cross-env NODE_ENV=development nodemon index.js",
  "test": "cross-env NODE_ENV=test jest --verbose --runInBand --detectOpenHandles",
```

Kuva 53. Skriptien määrittely package.json-tiedostossa.

```
const mongoose = require('mongoose')
const bcrypt = require('bcrypt')
const supertest = require('supertest')
const helper = require('./test_helper')
const app = require('../app')
const api = supertest(app)
const Quiz = require('../models/quiz')
const User = require('../models/user')
```

Kuva 54. quiz_api.test.js-tiedoston importtaukset.

app.js-tiedosto kääritään Supertest-kirjaston avulla api-muuttujaan sijoitettavaksi ”superagentiksi”, jonka avulla voidaan tehdä http-pyyntöjä backendiin. Pyyntöjen avulla voidaan testata esimerkiksi tokeneihin perustuvan valtuutuksen tai skeemoissa määriteltyjen validointien toimivuutta. Määritellään selkeyden vuoksi toisiinsa liittyvät (esimerkiksi visan lisäämistä koskevat) testit saman describe-lohkon sisällä kuvan 55 tapaan, alustetaan tietokanta tarpeen tullen BeforeEach-funktion sisällä ja muistetaan myös sulkea tietokantayhteys lopuksi AfterAll-funktion sisällä.

```
describe('when there is initially some quizzes saved', () => {

    /*Clearing the MongoDB database quiz collection and inserting
    initialQuizzes array into it. Also clearing the MongoDB database
    User collection and inserting a User into it. */
    beforeEach(async () => {

        await Quiz.deleteMany({})
        await User.deleteMany({})

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

        const quizzes = helper.initialQuizzes

        quizzes.forEach(quiz =>
            quiz.author = savedUser.id
        )

        await Quiz.insertMany(helper.initialQuizzes)

    })

    /*Creating a test using async and await to test that
    the correct number of quizzes is returned from MongoDB
    database. */
    test('correct number of quizzes is returned', async () => {

        const response = await api.get('/api/quizzes')

        expect(response.body).toHaveLength(helper.initialQuizzes.length)
    })

    /*Creating a test using async and await to test that
    a id field exists for each of the quizzes instead of an
    _id field. */
    test('the idenfitying field for a quiz is id and not _id', async () => {

        const response = await api.get('/api/quizzes')

        response.body.forEach(quiz => {
            expect(quiz.id).toBeDefined()
        })
    })
})
```

Kuva 55. Testien määrittely describe-lohkon sisällä.

5 TIETOVISASOVELLUS - FRONTEND

5.1 Frontendin luominen

Frontendin luominen onnistuu komentorivistä Vitellä. Vite luo projektin juurikansion ja mm. package.json- ja vite.config.js-tiedostot pohjaksi frontendille. Kun projekti on luotu kuvan 56 tapaan, App.css- ja index.css-tiedostot sekä assets-kansion voi tässä tapauksessa poistaa src-kansiosta tarpeettomina.

```
C:\Users\San\VAMK>npm create vite@latest Quiz_frontend -- --template react
Need to install the following packages:
  create-vite@5.1.0
Ok to proceed? (y) Y
√ Package name: ... quiz-frontend

Scaffolding project in C:\Users\San\VAMK\Quiz_frontend...

Done. Now run:

  cd Quiz_frontend
  npm install
  npm run dev
```

Kuva 56. Frontendin luova komento. Sovellus voidaan käynnistää komennolla npm run dev, heti kun riippuvuudet on asennettu ohjeistuksen mukaan.

Koska backend käyttää porttia 3001, muodostuu urliksi ”http://localhost:3001”. Luodun frontend projektin juurikansion sisältö on kuvassa 57 esitetyn kaltainen. Lisätään nyt kuvan 58 esimerkin tapaan vite.config-tiedostoon määrittely ”proxylle”, jonka avulla voidaan reiteille lähettää pyyntöjä yksinkertaisemmin palveluita määriteltäessä.

Nimi	Muokkauspäivä	Tyyppi	Koko
public	29.12.2023 23.13	Tiedostokansio	
src	2.1.2024 1.21	Tiedostokansio	
.eslintrc.cjs	29.12.2023 23.13	TypeScript JSX File	1 kt
.gitignore	29.12.2023 23.13	Tekstitiedosto	1 kt
index	29.12.2023 23.13	Microsoft Edge HT...	1 kt
package	29.12.2023 23.13	JSON File	1 kt
README	29.12.2023 23.13	Markdown Source ...	1 kt
vite.config	29.12.2023 23.13	JavaScript-tiedosto	1 kt

Kuva 57. Vitellä luodun projektikansion sisältö.

```
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'

// https://vitejs.dev/config/
export default defineConfig({
    plugins: [react()],
    server: {
        proxy: {
            "/api": "http://localhost:3001",
        }
    }
})
```

Kuva 58. Proxyn määrittely vite.config.js-tiedostossa.

5.2 Kirjautuminen ja rekisteröityminen

Asennetaan react-router-, yup-, axios- ja prop-types-kirjastot. Tiedostoon main.jsx on myös tehtävä pieni muutos. Määritellään kuvan 59 esimerkin mukaisesti komponentti App, Router-komponentin lapsikomponenttina, jotta sitä voidaan käyttää missä tahansa sovelluksen osassa.

```
import ReactDOM from 'react-dom/client'
import {
    BrowserRouter as Router
} from 'react-router-dom'

import App from './App'

ReactDOM.createRoot(document.getElementById('root')).render(

    //Defining App as a child of Router to allow use of Routes.
    <Router>
        <App />
    </Router>
)
```

Kuva 59. App-komponentin määrittely Routerin lapsikomponenttina.

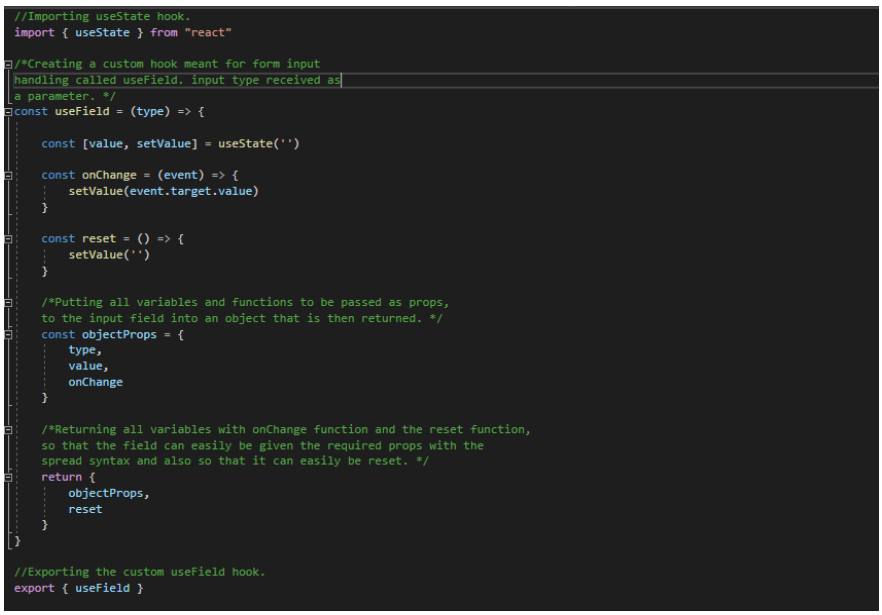
Luodaan seuraavaksi src-kansioon kansiot components, hooks ja services. Luodaan kansioon components tiedostot LoginForm.jsx ja SignUpForm.jsx ja Notification.jsx. Luodaan myös kansioon hooks tiedosto index.js ja kansioon services tiedosto Login.js.

Määritellään App.js-tiedostossa setSuccessMessage-, setErrorMessage-, signUp-, loginFnct- ja logOut-funktiot, jotka välitetään tarpeen mukaan propseina muille komponenteille. Määritellään myös tarpeelliset linkit ja reitit. Kysymysmerkkioperaattoria käyttämällä voidaan ehdollisen renderöinnin avulla näyttää Login-

tai Logout-linkki käyttäjälle riippuen siitä, onko käyttäjä kirjautunut kuvan 60 esimerkin esittämällä tavalla.

Kuva 60. Linkkien ja reittien määrittelyt App.js-tiedostossa.

Yksi tapa helpottaa lomakkeiden käsittelyä on määritellä hook-funktio tiedostoon index.js. Hookin nimi voi tässä tapauksessa olla useField ja sen avulla on mahdollista resetoida ja käyttää tekstikenttien arvoja helposti. Se voidaan määritellä kuvan 61 tapaan, viedä tiedostosta ja importoida tarpeen tullen muissa tiedostoissa.



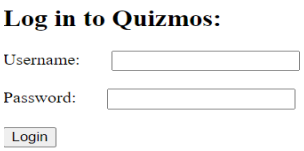
Kuva 61. useField-hookin määrittely tiedostossa index.js.

Sovellukseen tulee pystyä luomaan uusi käyttäjä. Luodaan tätä varten services-kansioon Users.js-tiedosto ja siihen POST-metodia käyttävä palvelu createUser. Exportoidaan palvelu ja käytetään sitä SignUpForm-komponentissa.

Määritellään sitten services-kansion tiedostoon Login.js axios-kirjaston avulla POST-metodia käyttävä login-palvelu, joka käyttää reittiä ”/api/login”. Importoidaan luotu palvelu App.js-tiedostossa ja käytetään sitä loginFnct-funktiossa. LoginFnct-funktio välitetään kuvan 62 esimerkin tapaan määritellylle LoginForm-komponentille propsina ja sitä kutsutaan, kun ulkoasultaan kuvan 63 kaltainen lomake luovutetaan.

```
/*Defining a LoginForm component with text and password type input
fields and a submit button. The required login method is received
from props. */
const LoginForm = ({ loginMethod }) => {

    const username = useField('text')
    const password = useField('password')

    //Defining a method to handle logging in and resetting the input fields.
    const handleLogin = async (event) => {
        event.preventDefault()
        await loginMethod(username.objectProps.value, password.objectProps.value)
        username.reset()
        password.reset()
    }

    return (
        <form onSubmit={handleLogin}>
            <h2>Log in to Quizmos: </h2>
            <div>
                Username:       
                <input name='userInput' autoComplete="off" {...username.objectProps} />
            </div>
            <br />
            <div>
                Password:       
                <input type='password' name='pwdInput' {...password.objectProps} />
            </div>
            <br />
            <div>
                <button id='loginBtn' type="submit">Login</button>
            </div>
        </form>
    )
}

//Defining prop validation for the LoginForm component with PropTypes.
```

Kuva 62. LoginForm-komponentin määrittely tiedostossa LoginForm.jsx.



Kuva 63. Kirjautumislomakkeen ulkoasu.

Määritellään myös App.js-tiedostoon linkki tekstillä ”Register” käyttäjän luomista varten. Käyttäjätietojen validointiin käytetään yup-kirjastoa ja validointiskeemaa. Validointiskeeman avulla voidaan esimerkiksi varmistaa, että salasanan uudel-

leensyöttö kentän sisältö vastaa annettua salasanaa, kuten kuvan 64 esimerkissä.

```
/*Defining a validation schema for the username, password and password reentry
using the imported yup library. */
const validationSchema = Yup.object().shape({
    username: Yup.string().min(3, 'Username must contain at least 3 characters.').
        required('Username cannot have an empty value.'),
    password: Yup.string().min(8, 'Password must contain at least 8 characters.').
        required('Password cannot have an empty value.'),
    passwordReentry: Yup.string().oneOf([Yup.ref('password'), null], 'Password re-entry is invalid.').
        required('You must confirm the password by re-entering it.'),
});

/*Defining a method to check if input for username, password and password-reentry
fields is valid. If not an error message is displayed. */
const validateInput = async () => {
    try {
        const pwd = password.objectProps.value
        const pwdReentry = passwordReentry.objectProps.value

        /*Validating the values of the username, password
        and passwordrReentry variables. */
        await validationSchema.validate({
            username: username.objectProps.value,
            password: pwd,
            passwordReentry: pwdReentry,
        }, { abortEarly: false });

        return true
```

Kuva 64. Käyttäjätietojen validointi yup-kirjaston avulla tiedostossa SignUpForm.jsx

5.3 Visojen listaaminen

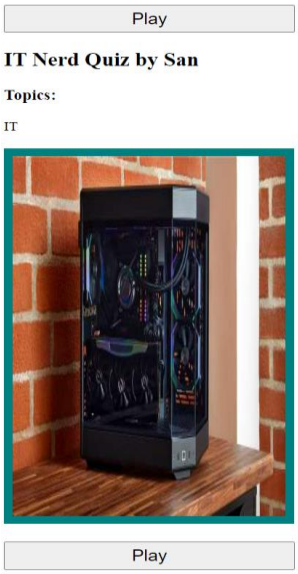
Visoja on pystyttävä myös selaamaan, poistamaan ja muokkaamaan. Tätä varten luodaan services-kansioon tiedosto Quizzes.js ja components-kansioon Quiz.jsx- sekä QuizList.jsx-tiedostot. QuizList-komponentissa määritellään deleteMethod- ja initializeQuizzes-metodit ja propsina vastaanotetaan successMsgMethod- ja errorMsgMethod-funktioiden lisäksi boolean arvo ”all”.

All-muuttujan arvo kertoo komponentille, näytetäänkö kaikki visat listassa, vai vain kirjautuneen käyttäjän luomat visat. App.js-tiedostoon määritellään myös All quizzes- ja My quizzes-reitit. All quizzes-reitin tulee mahdollistaa visan pelaaminen ja My quizzes-reitin tulee mahdollistaa visan muokkaaminen, sekä poistaminen.

Quiz.js-tiedostossa määritellään yksittäisen visan tietojen näyttäminen. Jos all-muuttujan arvo on tosi, välitetään Quiz-komponentille propsina epätosi boolean arvo ja toisinpäin, sekä kuvassa 65 esitettyyn tapaan määritellyn getQuizzes palvelun avulla haetun listan yksittäisen visan tiedot. Mydisplay-muuttujan arvo vastaanotetaan Quiz.js-tiedostossa ja se määrittää luodaanko yksittäiselle visalle ”Edit”- ja ”Delete”- painikkeet vai ”Play”-painike, kuten kuvassa 66.

```
//Defining a function to set the value of the token variable.
const setToken = newToken => {

    token = `Bearer ${newToken}`
}

/*Using axios.get method to get all existing quizzes
from the Node backend and returning the response. */
const getQuizzes = async () => {
    const response = await axios.get(baseUrl)
    return response.data
}

/*Using axios.get method to get a single existing quizzes
from the Node backend with an id received as a parameter
and returning the response. */
const getQuiz = async (id) => {
    const response = await axios.get(`${baseUrl}/${id}`)
    return response.data
}

/*Using axios.post method to send a quiz object to the Node backend
and returning the response. A jsonwebtoken is also sent to the backend
in the Authorization header for authorization and the Content-Type header is
altered to allow multer to recognize image data. */
const createQuiz = async (quiz) => {

    const config = {
        headers: {
            Authorization: token,
            'Content-Type': 'multipart/form-data'
        }
    }

    const response = await axios.post(baseUrl, quiz, config)
    return response.data
}

const deleteQuiz = async (id) => {

    const config = {
        headers: {
            Authorization: token,
            'Content-Type': 'multipart/form-data'
        }
    }

    const response = await axios.delete(`${baseUrl}/${id}`, config)
    return response.data
```

Kuva 65. Palveluiden määrittelyjä Quizzes.js-tiedostossa.



Kuva 66. Kaikki visat listaava näkymä.

5.4 Visojen luominen

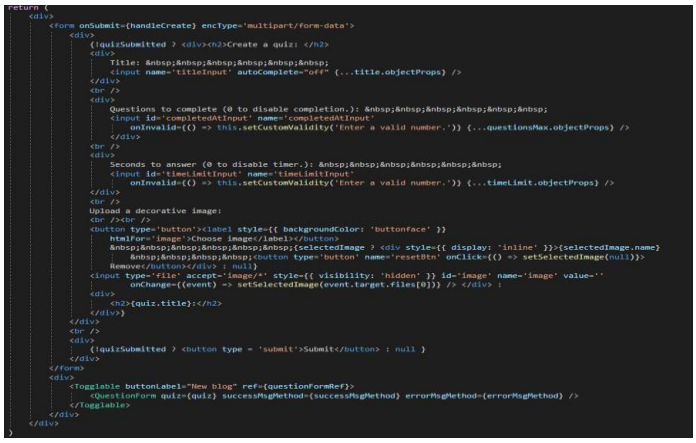
Visan luomisen mahdollistavan ”Create a quiz”-linkin tulee näkyä vain kirjautuneelle käyttäjälle. Tarvitaan CreateQuizForm-, QuestionForm- ja Togglable-

komponentit sekä Questions-palvelu. Määritellään ensimmäisenä kuvan 67 esimerkin tapaan Togglable-komponentti, jonka tarkoituksena on piilottaa lapsikomponentit tarvittaessa niiden tyylimäärittelyjen display-ominaisuuden avulla.

```
import { useState, useImperativeHandle, forwardRef } from 'react'

//Defining the Togglable component.
const Togglable = forwardRef((props, ref) => {

    /*Defining the visible "state variable" and 2 inline css styles,
    used to make the component visible and hidden. */
    const [visible, setVisible] = useState(false)

    const hideWhenVisible = { display: visible ? 'none' : '' }
    const showWhenVisible = { display: visible ? '' : 'none' }

    /*Defining a function to give the visible "state variable"
    the opposite of it's current value. */
    const toggleVisibility = () => {
        setVisible(!visible)
    }

    /*Using useImperativeHandle hook to allow other components
    to use the toggleVisibility method. */
    useImperativeHandle(ref, () => {
        return {
            toggleVisibility
        }
    })

    //Returning the necessary buttons to show and hide a child component.
    return (
        <div>
            <div style={showWhenVisible}>
                {props.children}
            </div>
        </div>
    )
})

Togglable.displayName = 'Togglable'

export default Togglable
```

Kuva 67. Togglable-komponentin määrittely Togglable.jsx-tiedostossa.

Määritellään CreateQuizForm-lomakkeen sisältö ehdollisen renderöinnin avulla ja asetetaan QuestionForm-komponentti Togglable-komponentin lapseksi. Luodaan myös viite Togglable-komponenttiin useRef-hookin avulla, jotta voidaan käyttää ToggleVisibility-funktiota komponentin ulkopuolella. CreateQuizForm-komponenttia voidaan käyttää yhdessä QuestionForm-komponentin kanssa visan muokkaamiseen.

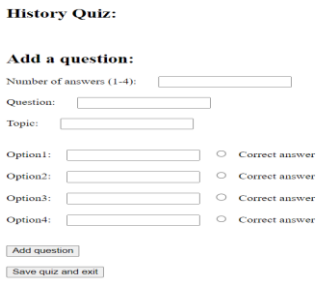


```
return (
    <div>
        <form onSubmit={handleCreate} encType='multipart/form-data'>
            <div>
                {!quizSubmitted ? <div><h2>Create a quiz: </h2>
                <div>
                    Title:       
                    <input name='titleInput' autoComplete="off" {...title.objectProps} />
                </div>
                <br />
                <div>
                    Questions to complete (0 to disable completion.):       
                    <input id='completedAtInput' name='completedAtInput'
                        onInvalid={() => this.setCustomValidity('Enter a valid number.')} {...questionsMax.objectProps} />
                    </div>
                <br />
                <div>
                    Seconds to answer (0 to disable timer.):       
                    <input id='timeLimitInput' name='timeLimitInput'
                        onInvalid={() => this.setCustomValidity('Enter a valid number.')} {...timeLimit.objectProps} />
                </div>
                <br />
                Upload a decorative image:
                <br /><br />
                <button type='button'><label style={{ backgroundColor: 'buttonface' }}
                    htmlFor='image'>Choose image</label></button>
                          {selectedImage ? <div style={{ display: 'inline' }}>{selectedImage.name}
                            <button type='button' name='resetBtn' onClick={() => setSelectedImage(null)}>
                    Remove</button></div> : null}
                <input type='file' accept='image/*' style={{ visibility: 'hidden' }} id='image' name='image' value=''
                    onChange={(event) => setSelectedImage(event.target.files[0])} /> </div> :
                <div>
                    <h2>{quiz.title}:</h2>
                </div>}
            </div>
            <br />
            <div>
                {!quizSubmitted ? <button type = 'submit'>Submit</button> : null }
            </div>
        </form>
        <div>
            <Togglable buttonLabel="New blog" ref={questionFormRef}>
                <QuestionForm quiz={quiz} successMsgMethod={successMsgMethod} errorMsgMethod={errorMsgMethod} />
            </Togglable>
        </div>
    </div>
)
```

Kuva 68. Ehdollisen renderöinnin ja Togglable-komponentin hyödyntäminen CreateQuizForm.jsx-tiedostossa.

”input type=file”-määrittelyn avulla mahdollistetaan kuvan lataaminen sovellukseen. Kun tietovisa on luotu, muutetaan quizSubmitted-muuttujan arvo ja näytetään QuestionForm-komponentti, jolle välitetään propsina luotu visaolio. Tiedostoon Questions.js määritellään myös palvelu, joka mahdollistaa vastauksen lisäämisen ja Quizzes.js-tiedostoon palvelu, joka mahdollistaa kysymyksen lisäämisen.

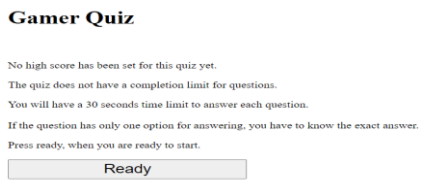
Vastausvaihtoehdot on myös validoitava frontendissä. Kysymys on lisättävä ennen vastausvaihtoehtoja, mutta vain kaikkien vaihtoehtojen sisältäessä eri arvon. Luodaan tätä varten metodi checkAnswers, joka palauttaa boolean arvon tarkistettuaan tilamuuttujia sisältävän answers-taulukon, jonka käyttäjä on syöttänyt kuvassa 69 esitettyyn lomakkeeseen.



Kuva 69. Lomake kysymysten ja vastausvaihtoehtojen luomiseen.

5.5 Visojen pelaaminen

Visoja tulee pystyä myös pelaamaan. Tätä varten luodaan tiedosto Play.jsx. Haetaan yksittäinen visa useParams-hookin avulla url:issa vastaanotetun id-arvon avulla ja luodaan ehdollisen renderöinnin avulla visalle kuvan 70 esittämä aloitusnäkymä.



Kuva 70. Visan aloitusnäkymä.

Visaa pelatessa tulee myös varmistaa, että sama kysymys ei toistu kahteen kertaan ja että vastausvaihtoehdot ovat ruudulla satunnaisessa järjestyksessä. Tämä saadaan aikaan käyttämällä Math.random-metodia, sekä questionsUsed-taulukkoa, jonka sisältöä verrataan arvottuun kysymykseen. Vastauksen tarkistamista varten tulee kuvan 71 tapaan määritellyn setRandom-metodin lisäksi määritellä myös checkAnswer-metodi, jossa kuvan 72 pelinäkymässä klikattua vastausta verrataan kysymykselle määritetyn correctAnswer-kentän arvoon.

```
    console.log(questionsUsed)
    console.log(quiz.questions.length)

  if (quiz && quiz.questions.length > questionsUsed.length) {

      random = Math.floor(Math.random() * quiz.questions.length)

      if (questionsUsed.length > 0) {
          while (questionsUsed.includes(quiz.questions[random])) {
              random = Math.floor(Math.random() * quiz.questions.length)
          }
      }

      let usedAnswers = []

      let newAnswers = []

      setQuestion(quiz.questions[random])

      setQuestionsUsed(questionsUsed.concat(quiz.questions[random]))

      if (quiz.questions[random].answers.length > 1) {

          randomAnswers = quiz ? Math.floor(Math.random() * quiz.questions[random].answers.length) : null

          for (let i = 0; i < quiz.questions[random].answers.length; i++) {
              if (usedAnswers.length > 0) {
                  randomAnswers = quiz ? Math.floor(Math.random() * quiz.questions[random].answers.length) : null

                  while (usedAnswers.includes(quiz.questions[random].answers[randomAnswers])) {
                      randomAnswers = quiz ? Math.floor(Math.random() * quiz.questions[random].answers.length) : null
                  }

                  newAnswers.push(quiz.questions[random].answers[randomAnswers])
                  usedAnswers.push(quiz.questions[random].answers[randomAnswers])

              } else {
                  randomAnswers = quiz ? Math.floor(Math.random() * quiz.questions[random].answers.length) : null
                  newAnswers.push(quiz.questions[random].answers[randomAnswers])
                  usedAnswers.push(quiz.questions[random].answers[randomAnswers])
              }
          }

          setAnswers(newAnswers)

      } else {

          newAnswers.push(quiz.questions[random].answers[0])
          setAnswers(newAnswers)
      }
  }
```

Kuva 71. Kysymyksen arpovan setRandom-funktion määrittely.

Kuva 72. Visan pelaamisen näkymä.

6 JOHTOPÄÄTÖKSET

Full Stack -kehitys vaatii tarkkuutta ja järjestelmällisyyttä koodatessa. Testaaminen on tärkeässä asemassa, koska liikkuvia osia on paljon. Backendin tulee hakea ja palauttaa tietoja onnistuneesti ja frontendin tulee renderöidä tarvittava http-pyyntöjen vastauksen mukana saatu tieto.

Backend ja frontend voidaan toteuttaa JavaScriptillä käyttäen Nodea ja Reactia. Kehitystyön avuksi on tarjolla useampia kirjastoja, jotka helpottavat full stack sovelluksen luomista. React ja Node ovat varteenotettavia vaihtoehtoja modernin web-kehityksen maailmassa.

Sovelluksen toiminnan voi myös toteuttaa esimerkiksi Apollo GraphQL:n avulla. GraphQL sallii haetun tiedon tarkemman määrittämisen parantaen mm. sovelluksen suorituskykyä. Tilanhallintaa on mahdollista myös helpottaa esimerkiksi Reduxia käyttämällä.

LÄHTEET

[1] Lionel Sujay Vailshery. (19.07.2023.) *Most used web frameworks among developers worldwide, as of 2023.* Statista | statista.com. https://www.statista.com/statistics/1124699/worldwide-developer-survey-most-used-frameworks-web/

[2] Sebastian Peyrott. (16.01.2017.) *A Brief History of JavaScript.* Auth0 | auth0.com. https://auth0.com/blog/a-brief-history-of-javascript/

[3] Google Cloud | cloud.google.com. Noudettu 21.11.2023 osoitteesta https://cloud.google.com/nodejs

[4] Kinsta. *What Is Node.js and Why You Should Use It.* Kinsta | kinsta.com. Noudettu 21.11.2023 osoitteesta https://kinsta.com/knowledgebase/what-is-node-js/

[5] Jethro Magaji. (25.08.2020.) *The History of Node.js.* Section| section.io. https://www.section.io/engineering-education/history-of-nodejs/

[6] Geeks for geeks. *Node.js Event Loop.* Geeks for geeks | geeksforgeeks.org. Noudettu 21.11.2023 osoitteesta https://www.geeksforgeeks.org/node-js-event-loop/

[7] Node.js *The Node.js Event Loop, Timers, and process.nextTick().* Node.js | nodejs.org https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick

[8] Npm trends | npmtrends.com. Noudettu 22.11.2023 osoitteesta https://npmtrends.com/@nestjs/core-vs-express-vs-fastify-vs-hapi-vs-koa-vs-restify-vs-sails

[9] Margaret Rouse. *Structured Data*. Techopedia | techopedia.com. Noudettu 24.11.2023 osoitteesta https://www.techopedia.com/definition/30363/structured-data

[10] Commvault | commvault.com. Noudettu 24.11.2023 osoitteesta https://www.commvault.com/supported-technologies/mongo-db

[11] Alexander S. Gillis. *What is JSON (JavaScript Object Notation)?* The Server-Side | theserverside.com. Noudettu 24.11.2023 osoitteesta https://www.theserverside.com/definition/JSON-Javascript-Object-Notation

[12] Full Stack Open| fullstackopen.com. Noudettu 27.11.2023 osoitteesta https://fullstackopen.com/osa4/backendin_testaaminen#super-test

[13] Michał Nerć (25.11.2017). *How to become a better programmer?* Medium | medium.com. https://medium.com/@derodu/design-patterns-kiss-dry-tda-yagni-soc-828c112b89ee

[14] Ferenc Hámori. *The History of React.js on a Timeline.* RisingStack | blog.risingstack.com. Noudettu 24.11.2023 osoitteesta https://blog.risingstack.com/the-history-of-react-js-on-a-timeline/

[15] Meta Open Source | opensource.fb.com. Noudettu 24.11.2023 osoitteesta https://opensource.fb.com/projects/react/

[16] Npm trends | npmtrends.com. Noudettu 24.11.2023 osoitteesta https://npmtrends.com/create-react-app-vs-vite

[17] Katie Lawson. *What Are Single Page Applications and Why Do People Like Them So Much?* Bloomreach | bloomreach.com. https://www.bloomreach.com/en/blog/2018/what-is-a-single-page-application