



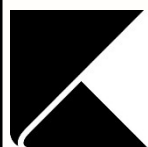
Karelia-ammattikorkeakoulu
Tradenomi, tietojenkäsittely (AMK)

REST-API:n toteutus sisältäen valtuuttamisen ja todentamisen

Nina Päivinen

Opinnäytetyö, joulukuu 2023

www.karelia.fi

**Karelia**
AMMATTIKORKEAKOULU**OPINNÄYTETYÖ**
Joulukuu 2023
Tradenomi, tietojenkäsittelyTikkarinne 9
80200 JOENSUU
+358 13 260 600 (vaihde)Tekijä(t)
Nina PäivinenNimike
REST-API:n toteutus sisältäen valtuuttamisen ja todentamisenToimeksiantaja
KARELIA-AMK**Tiivistelmä**

Opinnäytetyössä käsitellään pääaiheena back-end-sovelluksen toteuttaminen käyttäen REST-rajapintatekniikkaa, joka sisältää myös valtuuttamis- ja todentamisominaisuudet. Kirjallisessa osiossa selvitetään mikä on web-rajapinta ja miten se asiakas-palvelin-

arkkitehtuurimallissa toimii. Erilaisia todentamis- ja valtuuttamistapoja käydään kirjallisessa vaiheessa myös läpi. GraphQL- ja SOAP-rajapintatekniikat ovat myös työssä ohessa lyhyesti esittelyssä, vaikka REST on pääteema.

Sovelluskokeilussa toteutetaan tietokantaa käyttävä rajapinta aluksi pelkillä CRUD-toiminnoilla, jonka jälkeen luodaan ja yhdistetään sovellukseen valtuuttamiseen ja todentamiseen liittyvät ominaisuudet.

Lopulta REST-sovelluksen toteutuksen jälkeen vertaillaan, kuinka RESTful-periaate toteutui tässä kehitetyssä sovelluksessa.

Kieli
suomi

Sivuja 83

Asiasanat

Back-end, rajapinta, web-api, REST, tietokanta, mariadb, valtuuttaminen, todentaminen, crud, palvelin, javascript, nodejs

 Karelia UNIVERSITY OF APPLIED SCIENCES	THESIS DECEMBER 2023 Degree Programme in Xxx Tikkarinne 9 80200 JOENSUU FINLAND + 358 13 260 600 (switchboard)
Author (s) Nina Päivinen	
Title REST-API implementation including authorization and authentication Commissioned by KARELIA-AMK	
Abstract The main topic of the thesis is the implementation of a back-end application using REST interface technology, which also includes authorization and authentication features. The written section explains what a web interface is and how it works in the client-server architecture model. Different methods of authentication and authorization are also reviewed in the written phase. GraphQL and SOAP interface technologies are also showcased in the work, although REST is the main theme. In the application experiment, the interface using the database is initially implemented with only CRUD functions, after which the features related to authorization and authentication are created and combined with the application. Finally, after the implementation of the REST application, we will compare how the RESTful principle was implemented in this developed application.	
Language Finnish	Pages 83
Keywords Back-end, web-api, REST, database, mariadb, authorization, authentication, crud, server, javascript, nodejs	

Sisältö

1	Johdanto	7
2	Ohjelmointirajapinta	8
3	Rajapinnan toimintaperiaate: asiakas-palvelin-arkkitehtuuri	8
4	Rajapinnan toteutustekniikat	13
4.1	REST	13
4.1.1	Rajoitteet	15
4.1.2	Richardsonin kypsyyssmalli	16
4.2	GraphQL	17
4.3	SOAP	18
5	Valtuuttaminen	18
5.1	RBAC	19
5.2	ABAC	19
5.3	SSO	20
6	Todentaminen	20
6.1	Perustodennus	21
6.2	Bearer	21
6.3	OAuth 2.0	22
6.4	OpenID	23
6.5	JWT-todennus	23
7	Sovelluskokeilu: sovelluksen pohja	24
7.1	Tietokanta	25
7.2	Ohjelmoinnin ensivaiheet	28
7.2.1	Sovelluksen rakenne	29
7.2.2	Riippuvuudet	31
7.2.3	Tietokanta-kansio	32
8	Sovelluskokeilu: CRUD-toiminnot	34
8.1.1	Server.js päätiedosto	34
8.1.2	Routes-kansio	36
8.1.3	Models-kansio	37
8.1.4	Controllers-kansio	41
9	Testaus: CRUD-toiminnot	44
9.1.1	Testi: GET-pyyntö	44
9.1.2	Testi POST	46
9.1.3	Testi GET:ID	50
9.1.4	Testi PUT:ID	51
9.1.5	Testi DELETE:ID	53
10	Sovelluskokeilu: todentamisen ja valtuuttamisen	54
10.1.1	Sovelluksen juuri	54
10.1.2	Routes-kansio	55
10.1.3	Models-kansio	60
10.1.4	Controllers-kansio	61
10.1.5	Config-kansio	65
11	Testaus: valtuuttaminen ja todentaminen	66
11.1.1	Testi: Rekisteröi uusi käyttäjä	66
11.1.2	Testi: Roolin luonti	69
11.1.3	Testi: sisään kirjautuminen	71

11.1.4Testi: ADMIN-käyttöoikeus	72
11.1.5Testi: USER-sisältö.....	75
11.1.6Testi: PM-sisältö	75
11.1.7Testi: Näytä kaikki käyttäjät	76
12 Koirat-resurssin ja valtuuttamisen ja todentamisen yhdistäminen.....	77
13 Pohdinta.....	79
13.1 Sovelluksen kehitys mahdollisuudet	80
13.2 REST periaatteiden toteutuminen.....	80
LÄHTEET.....	82

1 Johdanto

Tässä opinnäytetyössä kirjallisessa osiossa selvitetään mikä on rajapinta eli WEB-API, kuinka se toimii asiakas-palvelin-arkkitehtuurimallissa, joka on REST-rajapintatekniikka sovellusta rakennettaessa yleisin arkkitehtuurityyli. GraphQL- ja SOAP-rajapintatekniikat esitellään myös lyhyesti opinnäytetyön ohessa, mutta REST on opinnäytetyön pääaihe ja yksistään jo aiheena hyvin laaja. Läpi käydään myös valtuuttamista ja erilaisia todentamistekniikoita, joita voidaan käyttää REST-tekniikalla olevaa sovellusta rakennettaessa.

Opinnäytetyön toiminnallisessa osiossa perehdytään sovelluskokeiluna toteuttamaan REST-tekniikkaan perustuva back-end-sovellus. Ensimmäisessä vaiheessa toteutetaan tietokanta ja koodataan pohja sovellukselle, jonka jälkeen lisätään sovellukseen CRUD-toiminnot. Viimeisenä vaiheena lisätään käyttäjän todennus- ja valtuutusominaisuudet ja yhdistetään nämä luotujen CRUD toimintojen kanssa. Sovelluskokeilussa käydään ohjelmiston rakennetta ja toimintaa läpi. Lopulta vertaillaan, kuinka RESTful-periaate toteutui tässä kehitetyssä sovelluksessa.

2 Ohjelmointirajapinta

Tietotekniikassa rajapinta eli API (eng. Application Programming Interface) voidaan hahmottaa kahden asian yhtymäkohtana, joka mahdollistaa näiden kahden tehdä viestipyyntöjä (eng. request) ja palauttaa viestivastauksia (eng. response) toisilleen. API mahdollistaa eri kohteiden tietojen vaihtoa eli keskustelua keskenään.

Rajapinta on alustasta riippumaton eli universaali, koska se mahdollistaa back-end- eli palvelinsovelluksen (eng. server) tietojen käyttämisen eri sovelluksissa tai järjestelmissä, pääasia on vain, että osataan käyttää itse rajapintaa, jonka kautta päästään käsiksi back-endin tarjoamiin tietoihin. Rajapinta voi olla myös sovelluksen sisäiseen käyttöön tarkoitettu, jolloin se vaihtaa tietoja eri kerrosten välillä sovelluksessa. Rajapintaa kutsutaan myös WEB-APIksi, koska se toimii verkossa. (Valjas 2023).

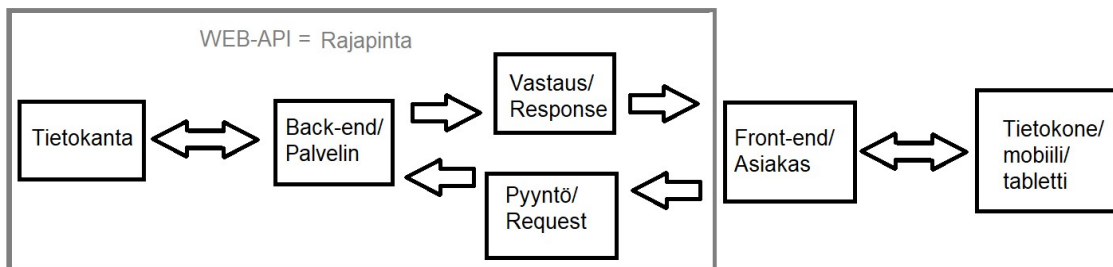
Avoin rajapinta (eng. Open API) eli julkinen rajapinta (eng. Public API) tarkoittaa sitä, että vain yhdistymällä API-sovellukseen, voi kuka tahansa voi lukea rajapinnan tarjoamia tietoja. Rajapinta mahdollistaa integraation toteuttamisen, mutta se ei itse ole rajapinta. Integraation tarkoitus on yhdistää kaksi erillistä järjestelmää yhdeksi kokonaisuudeksi. (Valjas 2023).

Rajapinnan avulla hyödynnetään back-endin puolelle rakennettua älyä, ilman että sitä tarvitsee rakentaa front-end- eli asiakaspuolelle (eng. client), jolloin kuormitetaan front-endin päätelaitteita huomattavasti vähemmän. (Matveinen 2023).

3 Rajapinnan toimintaperiaate: asiakas-palvelin-arkkitehtuuri

Asiakas-palvelin-arkkitehtuuri rajapinta voidaan karkeasti jakaa kahteen pääosaan front-endiin ja back-endiin (kuva 1). Internetin käyttäjä on

loppukäyttäjä, joka selaa verkkosivuja joko internetiselaimella tietokoneella tai mobiililaitteella, mutta näkee selauslaitteesta riippumatta verkkosivuilla aina saman front-end-ohjelmiston. (InterviewBit 2023).



Kuva 1. Asiakas-palvelin-arkkitehtuurimalli.

Web-sovelluskehys eli WF (eng. Web Software Framework) tarkoittaa ohjelmistotuotetta, joka muodostaa rungon eli ytimen sen päälle rakennettavalle tietokoneohjelmalle mm. React, Angular ja Vue.js ovat tuon front-endin rakentamiseen hyvin laajalti nykyisin käytettyjä tekniikoita. Front-end käyttää rajapintaa vuorovaikuttaakseen back-endin kanssa, pyytääkseen siltä tietoja tai palveluita. (Guru99 2023).

Backend Framework esimerkiksi Express JS, Python Django auttavat luomaan back-endin paljon vähemmällä koodi määrällä. Back-endillä on monia käyttötarkoituksia, ne on suunniteltu käsittelemään useita back-end-pyyntöjä samanaikaisesti ja myös hostaamaan eli ylläpitämään palveluita, sovelluksia ja tavallisia verkkosivustoja. Back-end on vähemmän näkyvillä, vaikka tiedämme senkin olevan siellä ja niillä on usein rakennettu kyky pystyä kommunikoimaan tietokannan kanssa. Back-endiltä saamme vain joitakin tietoja, mutta suoraan sen kanssa kosketuksissa ei käyttäjä ole, kaikki kommunikaatio tapahtuu rajapinnan avustuksella front-endin kautta. (GeeksForGeeks 2023).

Front-endin ja back.endin välinen kommunikointi alkaa siitä, kun internetin käyttäjä siirtyy selaimella johonkin URI-päätepisteeseen (eng. uniform resource identifiers, end-point), esimerkiksi <https://koirat.fi>. Yksi päätelaitteessa tapahtuva verkkosivun aukaiseminen aiheuttaa front-end-ohjelmistosta back-endille useita pyyntöjä, johon se vastaa vastauksella takaisin. Tiedot mitä

esimerkiksi voidaan hakea voi olla back-endin tietokannassa oleva tieto, tässä tapauksessa koirarotujen tiedot sijaitsevat URI-päätepisteessä <https://www.koirat.fi/rodut>, tätä polkua kutsutaan myös resurssiksi. (GeeksForGeeks 2023).

Haettava tieto on kaikki mitä verkkosivun sisällön ja toiminnallisuuden rakentamiseen tarvitaan ja voi olla muodoltaan esimerkiksi:

- kuvat (jpg, png, gif)
- mediat (wma, mp3, wav)
- teksti (plain)
- fontit
- kooditiedosto (javascript, php, css, html)
- tietokannan tietojen esittämiseen käytetyt kielet (json, xml) (Tray.io 2023).

Pyyntö toteutetaan HTTP:ssä metodia, toimintoa eli verbiä käyttämällä, joka määrittää mitä tiedoille halutaan tehdä. HTTP-metodeihin kuuluvat CRUD-metodit, joka tulee sanoista create, read, update ja delete, kun halutaan lisätä tietoa eli create, eli POST, lukea tietoa read eli GET, päivittää tietoa update eli PUT tai poistaa tietoa delete eli DELETE. Esimerkiksi front-end-sovellus voi lähettää back-endille HTTP-GET pyynnön URI-päätepisteeseen <https://www.koirat.fi/rodut>, joka palauttaa nähtäväksi tietoja tietokannasta. (Tray.io 2023)

Front-end-sovelluksen luoma viestipyyntö sisältää tietoja paljon tietoja, mutta tässä tärkeimmät otsikko ja runko osat:

Otsikko (eng. header)

HTTP-metodi (esim. PUT)

URI-päätepiste (esim. /api/koirat)

lähetettävän tiedon tyyppi (esim. content-type: application/json)

päivämäärä ja kellonaika

mahdolliset todennus- tai valtuutustiedot (esim. authorization: Basic salainen12345)

Runko (eng. body)

Runko sisältää lähetettävän datan. (esim. JSON)

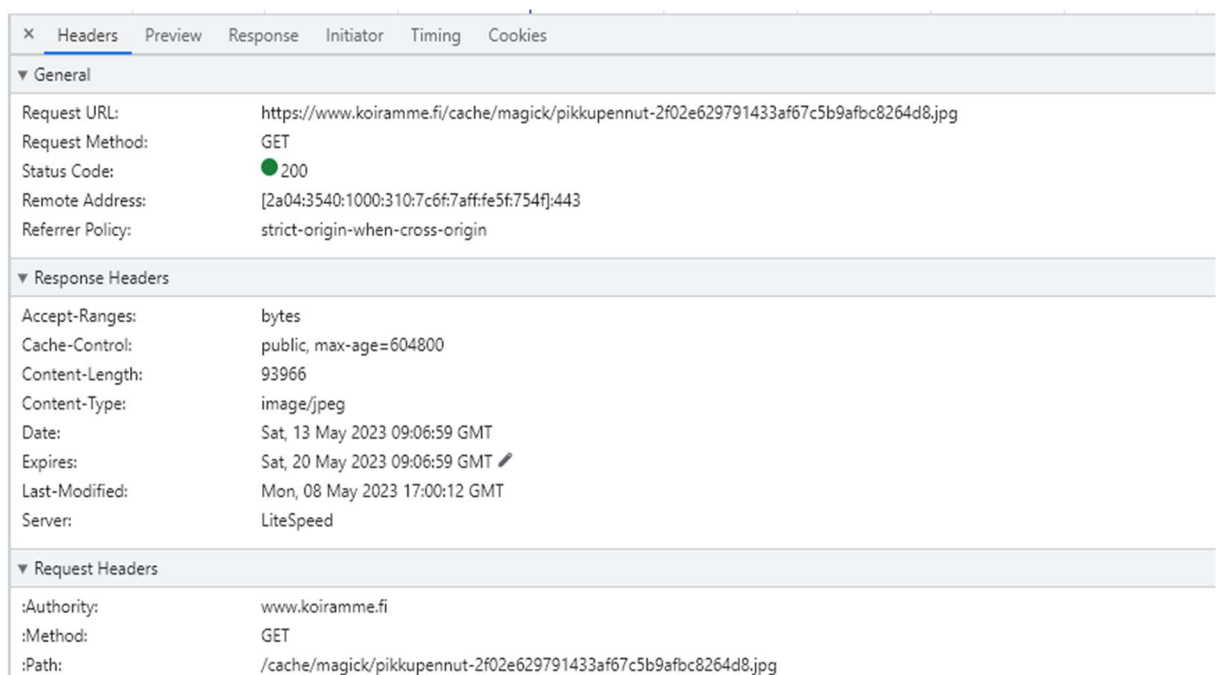
Pyynnön jälkeen back-end käsittelee pyynnön, suorittaa tarvittavat toiminnot ja palauttaa viestivastauksen, viestin rungossa on pyydetty data front-endille, jos pyyntö on vain hyväksytty. Vastaus on samanlainen, mutta sisältää lisäksi HTTP-tila eli -statuskoodin, joka kertoo front-endille, onko toiminto onnistunut vai ei ja mikä sen status on. HTTP-tilakoodit antavat arvokasta tietoa ohjelmoitsijoille (taulukko 1). (Tray.io 2023).

HTTP-tilakoodi	Selvennys	Esimerkki
100-199	Vaiheessa. Asiakkaan lähettämä pyyntö on vastaanotettu palvelimella ja käsittelyssä.	100 Continue. Kaikki kunnossa pyynnöstä tähän mennessä, asiakkaan tulee jatkaa pyyntöä.
200-299	Onnistunut. Asiakkaan lähettämä pyyntö on hyväksytty palvelimella ja käsitelty.	200 OK. Pyyntö onnistunut.
300-399	Keskeneräinen. Asiakkaan lähettämä pyyntö vaatii palvelimelle jatkotoimenpiteitä, jotta se saadaan loppuun käsiteltyä.	300 Multiple Choices. HTTP-pyyntöön seurauksena on saatavilla useampi kuin yksi HTTP-vastaus, vaaditaan asiakkaalta lisätoimia.
400-499	Asiakas virhe. Asiakkaan lähettämä pyyntö on saapunut palvelimelle, mutta sitä ei ole voitu	404 Not found. Asiakaspyynnön resurssi ei ole löydettävissä.

	käsitellä asiakkaasta johtuvista syistä loppuun asti.	
500-599	Palvelin virhe. Asiakkaan lähettämä pyyntö on saapunut palvelimelle, mutta sitä ei ole voitu käsitellä palvelimesta johtuvista syistä loppuun asti.	500 Internal Server Error Palvelin on kohdannut odottamattoman ongelman. Tämä vastaus on yleensä annettu, kun mikään muu ei ole sopiva.

Taulukko 1. HTTP-tilakoodit (Http.dev 2023).

Verkkosivulla yleensä joudutaan toteuttamaan useampi pyyntö, nämä pyynnot tapahtuvat aina yksi kerrallaan (kuva 2), eli front-end lähettää pyynnön ja back-end antaa vastauksen ja sen jälkeen otetaan seuraava pyyntö vasta käsittelyyn, eikä samanaikaisesti useampaa pyyntöä. (InterViewBit 2021).



Kuva 2. Chrome-selaimessa näkymä Developer Tools -> Network välilehdessä. Onnistunut HTTP-GET pyyntö jpeg-muotoisesta kuvasta, josta front-endille on palauttanut HTTP-tilakoodin 200.

Back-endin palauttama tieto tietokannasta voi olla monessa eri tietomuodossa, jota se antaa front-endin käyttöön, yleisimpiä ovat XML ja JSON. Tietomuodot, jotka ovat käytössä riippuu valitusta rajapintatekniikasta. JSON on kehittyneempi ja helpompi lukuisempi kuin XML-kieli, muita harvemmin kieliä on olemassa kuten TOML, CSON ja YAML. Verkkosivulla oleva sisältö voi olla dynaaminen, eli back-end luo sisällön sivulle hyödyntäen esim. tietokannassa olevaa dataa. Staattista sisältöä eli tuolloin back-endin hakemistosta peräisin oleva tieto on pelkkä tekstitiedosto. (JeffTk 2023).

4 Rajapinnan toteutustekniikat

Rajapintojentoteutustekniikat ovat erilaisia lähestymistapoja online-tiedonsiirtoon, ne määrittelevät, kuinka rakennetaan rajapintoja, jotka mahdollistavat tiedon välittämisen back-endin puolelta front-endin käyttöön. (Aloi.io 2023)

Verkkosivustot ovat kehittyneet alkuaikojen pelkistä staattisista nykyaikaisiksi dynaamisiksi ja interaktiivisiksi raskaammiksi kokonaisuuksiksi, joista monet jo käyttävät jonkinlaista rajapintaa tai useampaa. Rajapinnat toimivat hyvin oleellisessa osassa web-sovelluskehityksessä nykypäivänä. (Aloi.io 2023)

4.1 REST

Termi REST on syntynyt vuonna 2000 ja ensimmäisen kerran sitä käytti Roy Fieldingin väitöskirjassaan nimeltä Architectural Styles and the Design of Network-based Software esittelemä – arkkitehtuurityyli ohjelmointirajapintojen toteuttamiseen, jota myös arkkitehtuurimalliksi on kutsuttu. (Ics 2023).

REST-tekniikalla toteutettu rajapinta tarjoaa pääsyn resursseihin yksilöllisten URL-päätepisteiden kautta eli se perustuu resurssipohjaiseen arkkitehtuuriin, jossa käyttäjät vuorovaikutuksessa resurssien kanssa. Rajapinta joka toteutuu

käyttäen REST-tekniikka, voidaan kutsua myös RESTful rajapinnaksi. (Ics 2023).

Rajapinnan ollessa RESTllä toteutettu back-end ja front-end viestivät toisilleen sen avulla HTTP-tiedonsiirtoprotokollaa (eng. Hyper Text Transfer Protocol) käyttäen. REST-tekniikalla toteutettu pyyntö on HTTP- tai HTTPS-pyyntö, joka käyttää jotakin HTTP-metodia, johon back-end palauttaa oman vastauksen. HTTPS-pyynnöt ovat aina salattuja ja eivät näy muille, toisin kuin pelkät HTTP-muotoiset pyynnot, joita ei suositella enää nykyisin käytettäväksi. Back-endin HTTP-vastausten tilakoodit selkeyttivät myös ohjelmointia (Ics 2023).

Nykyisin REST:n käyttäjäkunta on hyvin suuri ja laajimmin tänä päivänäkin vielä käytetty rajapintatoteutustekniikka. Sen käytön oppiminen on suhteellisen helppoa, koska se käyttää standardia arkkitehtuuria ja REST-pyyntöviestit ovat pieniä ja selkeästi tulkittavia. (Altexsoft 2023).

REST tarjosi standardoidun selkeän tietomuodon JSONin ja paransi näin suorituskyykyä ja tehokkuutta, mutta se käytössä on myös mahdollisuudet käyttää muitakin tiedostomuotoja (Altexsoft 2023). Kevyiden protokollien ansiosta pidetään myös nopeana ja tehokkaana. Kyky tallentaa vastaukset välimuistiin (eng. cache) lisää RESTn suorituskyykyä. (Guru99 2023).

Yli- tai alihaku (eng. fetching) on tunnettu ongelma RESTn käytössä, tuolloin front-endin käyttöön saadaan tietoja back-endin päätepisteistä, joka antaa yleensä enemmän tietoa kuin olisi ollut tarpeen. (Altexsoft 2023).

Arkkitehtuuriset elementit jaetaan kolmeen eri pääluokkaan REST-arkkitehtuurityylissä:

- komponentit käsittelevät järjestelmän tietoa (mm. front-end-, back-end- jaottelu)
- liittimet näiden komponenttien välinen kommunikointi onnistuu liittimien avulla
- dataelementit yksilölliset URI-päätepisteet eli resurssien yksilöivä tunniste, jossa sovelluksen tieto sijaitsee. (Ics 2000).

4.1.1 Rajoitteet

REST määrittää kuusi arkkitehtuurillista rajoitusta, jotka tekevät mistä tahansa rajapinnasta REST-back-end-sovelluksen kriteerit täyttävän:

1. Asiakas-palvelin erottelu (eng. client-server separation): front-end vastaa käyttöliittymästä, kun taas back-end vastaa pyyntöjen käsittelystä ja resurssien hallinnasta. Tämä erottelu mahdollistaa sekä front-end- että back-end-komponenttien itsenäisen kehityksen ja skaalautuvuuden. Front-endin tarvitsee tietää siis vain back-endin dataelementit eli URI-päätepisteet.
2. Tilaton (eng. stateless): palvelin ei tallenna mitään front-endin viimeisimmästä HTTP-pyynnöstä. Se käsittelee jokaisen pyynnön uutena, eikä tallenna aiempia istunto tietoja, eikä historiaa. Tilattomuus kuormittaa verkkoa mahdollisesti enemmän, mutta resursseja back-end voi vapauttaa käyttöönsä nopeammin ja palautua vikatilanteista helpommin.
3. Välimuisti (eng. caching): välimuistiin tallentamisen mahdollisuus parantaa front-end-puolen suorituskykyä ja näin ollen back-endin skaalautuvuutta, koska kuormitusta on vähennetty.
4. Yhteinen käyttöliittymä (eng. uniform interface): yhtenäinen käyttöliittymä yksinkertaistaa ja erottaa arkkitehtuurin, mikä mahdollistaa jokaisen osan kehittymisen itsenäisesti RESTful-rajapinnassa.

Tämä rajoite sisältää lisäksi neljä lisäperiaatetta:

1. Resurssi pohjainen (eng. resource-based)
2. Itse kuvaavat viestit (eng. self-descriptive messages)
3. Resurssien manipulointi edustustojen kautta (eng. manipulation of resources through representations)
4. HATEOAS (eng. Hypermedia As The Engine Of Application State)
5. Kerrostettu järjestelmä (eng. layered system): RESTful-arkkitehtuuri edellyttää, että suunnittelu on jäsennelty kerroksiksi, jotka toimivat yhdessä. Tämä on etuna, jos rajapinnan laajuus muuttuu, tasoja voidaan lisätä,

muokata tai poistaa vaarantamatta muita käyttöliittymän osia eli rajapinnan skaalattavuus ja tietoturva on kerrostetun järjestelmän ansiosta parempi. Välimuisti auttaa helpottamaan kuormittumista, joka aiheutuu kerroksittaisen järjestelmän käytöstä.

6. Koodi pyynnöstä (eng. code-on-demand): tämä rajoitus on valinnainen. Ideana on sallia koodin lähettäminen rajapinnan kautta ja käyttö sovelluksessa. (Ics 2000).

Näiden kuuden yllä mainitun rajoitusten tarkoitus on taata REST-rajapinnan:

- skaalautuvuus
- suorituskyky
- luotettavuus
- ylläpidettäviä
- yhteensopivia (Ics 2000).

4.1.2 Richardsonin kypsyysmalli

Richardsonin kypsyysmallin (eng. Richardson Maturity Model) mukaisen taso määritetään sen avulla, kuinka verkkosovellukset noudattivat REST-arkkitehtuurin kuutta eri rajoitetta. Tasoja on nollasta kolmeen asti, josta kolme on korkein mahdollinen saavutettava kypsyystaso. (Rest Api Tutorial 2021).

Sovelluksen kypsyyttä tarkastellaan kolmen tekijän kautta:

- URI-päätepiisteet
- HTTP-metodit
- HATEOAS (Rest Api Tutorial 2021).

Kypsyystaso 0 palveluilla on yksi URI-päätepiiste ja ne käyttävät jotakin yhtä HTTP-verbiä, joka yleensä on POST. (Rest Api Tutorial 2021).

Kypsyystaso 1 yksi käyttää useita resursseja, mutta vain yhtä HTTP-verbiä, joka yleensä on vain POST. (Rest Api Tutorial 2021).

Kypsyystaso 2 käyttää useita eri resursseja ja HTTP-menetelmiä, mutta ei käytä HATEOASia. Tämän tason rajapinnat hyödyntävät täysin kaikkia HTTP-verbejä. Monet REST-rajapinnat täyttävät tämän tason, vaikka eivät kuitenkaan korkeinta kypsyystasoa. (Rest Api Tutorial 2021).

Kypsyystaso 3 kypsyysasteella käytetään kaikkia kolmea eli päätepistettä ja HTTPtä sekä HATEOASa. Taso kolme on Richardsoninmallin kypsien taso. HATEOS tarkoittaa, että rajapinta käyttää jossakin paluuviesteissä URI-linkkejä, jolloin rajapinnan käyttäjän ei tarvitse itse tietää kaikkia resurssien osoitteita, tämä mahdollistaa rajapinnan käyttäjän dynaamisen ohjaamisen back-endistä käsin. (Matveinen M 2023).

4.2 GraphQL

GraphQL-kyselykieli edustaa uutta rajapintatekniikkaa ja on Facebook yrityksen vuonna 2012 kehittämä, näin ollen uusien rajapintatoteutustekniikoista. Yksi GraphQLn tärkeimmistä ominaisuuksista on sen kyky noutaa useita resursseja yhdellä pyynnöllä. REST-rajapinnassa, jossa asiaan liittyviä resursseja haetaan käyttöön, vaatii se yleensä useita pyyntöjä eri URI-päätepisteisiin. GraphQLn avulla käyttäjät voivat kuitenkin määrittää tarkkaan määritellyjä kyselyitä ja hakevat kaikki tarvittavat tiedot yhdellä kertaa. Kommunikointi rajapinnan kanssa on täten hienostuneempaa ja tekee skaalaamisesta paljon yksinkertaisempaa. GraphQLää pidetään täten yleensä tehokkaampana ja joustavampana kuin REST, kun sen käytössä ei ole ylihaun ongelmaa modernimpien kyselyjen ansiosta. (Kinsta 2023).

GraphQLn etu on myös sen tyyppijärjestelmä, jonka avulla käyttäjät voivat määritellä skeeman, joka kuvaa saatavilla olevat tiedot ja toiminnot (Kinsta (2023)). Halutessa siirtyä monoliittisesta taustasovelluksesta mikropalveluarkkitehtuuriin GraphQL-rajapinta auttaa käsittelemään erilaisten mikropalvelujen välistä viestintää yhdistämällä ne yhdeksi GraphQL-skeemaksi. (Altexsoft 2023).

GraphQLn haasteina voidaan pitää, että oppiminen on haastavampaa kuin RESTn käytön, eikä sille ole tukea HTTP-välimuistin käyttöön. GraphQL ei ole myöskään hyödyllinen pienten sovellusten rakentamiseen arkkitehtuurinsa puolesta. (Kinsta 2023).

4.3 SOAP

SOAP (eng. Simple Object Access Protocol) on tiedonsiirtoprotokolla, joka on syntynyt 1999 vuonna Microsoftin kehittämänä ja se on XML-viesti pohjainen. SOAP-viestit lähetetään back-endille XML-muotoisina SOAP-kirjepyyntöinä (eng. envelope), johon back-end vastaa myös XML-muotoisena SOAP-kirjevastauksena. SOAP on käytettävissä HTTP-, SMTP-, TCP- ja muiden sovellustasonprotokollien kautta. (GeekForGeeks 2023).

Hyvä rajapintateknikka ratkaisu vahvaa tietoturvallisuutta vaativiin ratkaisuihin ja siinä on myös sisäänrakennettuja suojausominaisuuksia, kuten WS-Security. SOAP tukee useita suojausstandardeja kuten salaus, digitaaliset allekirjoitukset ja todentaminen. (Altexsoft 2023).

Nykyisin SOAPin käyttäjäyhteisö on pieni RESTn tultua yleisimmäksi. SOAP tarvitsee enemmän kaistanleveyttä käyttöä varten, koska SOAP-kirjeviestit sisältävät paljon tietoa ja sen käyttö on myös vaikeampaa. Nykyisin SOAPia pidetään jo vanhentuneena tekniikkana, se käyttääkin ainoastaan vain XML-tiedostomuotoa, eikä sille ole tukea välimuistin käyttöön. (Altexsoft 2023)

5 Valtuuttaminen

Valtuutus (eng. authorization) määrittää, mitä käyttäjät voivat tehdä järjestelmässä, kun pääsy järjestelmään on sallittu kyseiseltä käyttäjältä todentamisvaiheen jälkeen. Valtuuttamisella on tärkeä rooli järjestelmien

tietojen suojaamisessa. Kaikki tieto ei kuulu olla kaikkien todennettujen käyttäjien saatavilla joka paikassa järjestelmää, vaan valtuuttamisella halutaan rajoittaa tätä mikä tieto on tarkoitettu kenenkin saatavaksi.

Identiteetinhallinta eli IdM (eng. Identity Management) on valtuutuksen tärkeä osa ja se hallinnoi käyttäjien sähköistä henkilöidentiteettiä ja mahdollistaa pääsyn vain sallittuihin tietoihin ja järjestelmiin tai järjestelmän osiin.

5.1 RBAC

Identiteetinhallinta voidaan toteuttaa rooliperusteisena pääsynhallintana eli RBAC (eng. Role-Based Access Control), jossa tehdään käyttäjien eritasoisten roolien määrittämisellä käyttöjärjestelmään ja asettamalla näille rooleille tietyt oikeudet mitä on lupa tehdä ja nähdä kyseisessä järjestelmässä. (Atostek 2023).

Tämä yksinkertaistaa käyttöoikeuksien hallintaa, koska käyttöoikeudet myönnetään organisaation työtehtävien ja toimintojen perusteella. RBAC on yleinen isoissa organisaatioissa. (Atostek 2023).

Esimerkkejä RBAC rooleista:

- käyttäjä (eng. User) – matalat valtuudet
- PM (eng. Project Manager) – keskitason valtuudet
- ylläpitäjä (eng. Admin) – korkeat valtuudet

5.2 ABAC

RBACn identiteetinhallina vaihtoehtona voidaan pitää attribuuttiperusteista pääsyhallintaa eli ABACta (eng. Attribute-Based Access Control). ABACissa käyttöoikeuspäätökset perustuvat käyttäjään, käytettävään resurssiin ja nykyisiin ympäristöolosuhteisiin liittyviin attribuutteihin. Nämä attribuutit voivat

sisältää käyttäjän ominaisuuksia, resurssin ominaisuuksia ja muita mahdollisia tekijöitä. (Atostek 2023).

ABAC mahdollistaa hienorakeisen pääsynhallinnan ottamalla huomioon useita attribuutteja tehdessään pääsypäätöksiä, joten se on joustavampi ja dynaamisempi lähestymistapa pääsyoikeuksien hallintaan.

5.3 SSO

SSO (eng. Single Sign-On) tarkoittaa kertakirjautumista, jossa käyttäjä yhdellä kirjautumiskerralla pääsee käyttämään useita palveluita. Tämä on mahdollista luomalla käyttäjälle väliaikainen tunnistetieto tai istunto, joka toteutuu käyttäen jotakin protokollaa kuten esimerkiksi JWT-Token tai OpenId. (Atostek 2023).

Mikropalveluiden välillä tapahtuva tunnistautuminen liikkuu viestipyyntöjen mukana, jolloin käyttäjälle näyttää niin kuin hän toimisi yhden ainoan järjestelmän sisällä. (Atostek 2023).

6 Todentaminen

Todennus (eng. authentication) varmistaa resursseihin pääsyä yrittävän käyttäjän identiteetin ennen käyttöoikeuden myöntämistä. Todentamisen tärkeä merkitys sovelluksessa on, että vain valtuutetut käyttäjät tai järjestelmät voivat olla vuorovaikutuksessa suojatuiksi määriteltujen URI-päätepisteiden kanssa. (Strongdm 2023).

Todentamisella on tärkeä rooli järjestelmien tietoturvassa, koska kaikille ei kuulu pääsy järjestelmään ollenkaan. Todentamisella tarkastetaan käyttäjän identiteetti ja onko tietokannassa määritetty kyseiselle identiteetille pääsyoikeus järjestelmään. (Strongdm 2023).

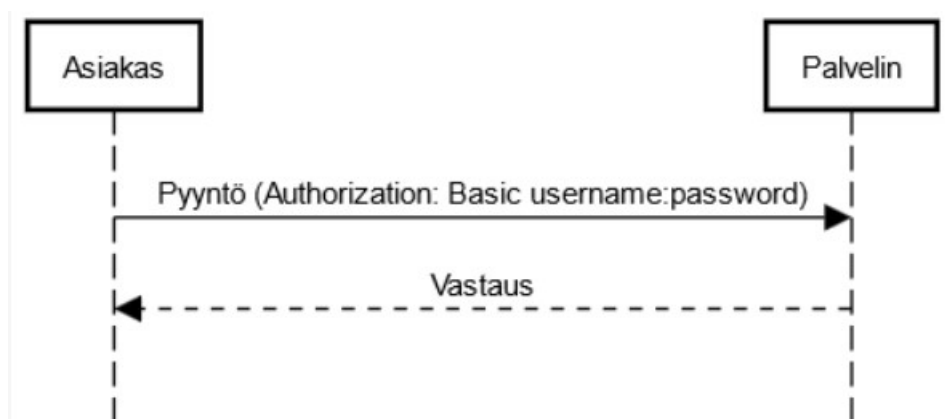
Eri rajapintatekniikat mahdollistavat kukin eri tapoja todentamiseen ja RESTssä onkin useita tapoja todentaa käyttäjä sovelluksessa. SOAPlla ja GraphQLlää on omat mahdolliset todentamistavat, joista osa on samoja kuin RESTillä.

Seuraavaksi käydään teoreettisesti läpi tunnetumpia RESTn todentamisen mahdollistavia menetelmiä. (Strongdm 2023).

6.1 Perustodennus

HTTP-valtuutus scheemoja on useita erilaisia, joista perustodennus (eng. basic HTTP-authentication-scheema) on yksi. Tällä vanhalla menetelmällä käyttäjän käyttäjätunnus ja salasana lähetetään pelkkänä tekstinä back-endille lähetyn viestipyynnön valtuutusotsikossa. Tämän jälkeen back-end tarkistaa käyttäjätunnuksen ja salasanan oikeellisuuden tietokannassa ja tunnusten ollessa oikein antaa back-end vastauksena resursseista pyydetty tiedot (kuva 3). (GeeksForGeeks 2023).

Tämä menetelmä on yksinkertainen, mutta ei kovin turvallinen, koska tunnistetiedot lähetetään pelkkänä tekstinä pyynnön mukana. Perustodennusta pidetään vanhentuneena ja ei kovin suositeltava todentamistapana nykyisin. (GeeksForGeeks 2023).

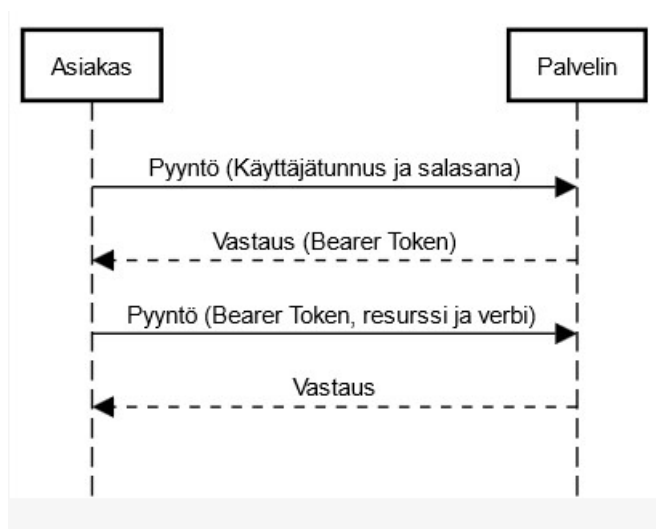


Kuva 3. Basic-pyyntö.

6.2 Bearer

Bearer-todennusjärjestelmä luotiin alun perin osana OAuth 2.0 i:tä ja on myös yksi HTTP-valtuutus scheemoista (Writer 2023). Bearer-tokenissa front-end lähettää oman käyttäjätunnus ja salasana tietonsa back-endille, joka niiden oikeellisuuden tarkistettua palauttaa merkkijonomuotoisen (eng. string) Bearer-tokenin front-endin käyttöön. (DevlosSchool 2023)

Tokenin avulla voi tehdä back-endille pyyntöjä, jos tokenin ollessa vain kelvollinen ja voimassa antaa back-end tarjoamiaan tietojansa front-end-sovelluksen käyttöön rajapinnan kautta (kuva 4). (DevlosSchool 2023)



Kuva 4. Bearer-Token-pyyntö.

6.3 OAuth 2.0

2012 vuonna OAuth versio 1.0:n korvasi OAuth 2.0 valtuutuskehys/protokolla, nykyisin sitä käytetään usein todentamiseen pääsyssä resursseihin verkko-, mobiili- ja työpöytäsovelluksissa. OAuth 2.0 standardi käyttää token-menetelmään (usein JWT- ja Bearer-tokenia) ja se mahdollistaa Googlen kautta tunnistautumisen, jonka avulla kolmannenosapuolen sovellukset voivat käyttää käyttäjätietoja resurssipalvelimelta käyttäjän puolesta ilman, että käyttäjän on luovutettava tunnistetietojaan eli käyttäjätunnusta ja salasanaa kolmannen osapuolen sovellukselle. (DevOpsSchool 2023)

OAuth 2.0 toimintaperiaate on seuraava:

- Käyttäjä siirtyy kolmannen osapuolen sovelluksen verkkosivustolle ja napsauttaa painiketta päästäkseen valtuutuksen vaativaan resurssiin.
- Kolmannen osapuolen sovellus ohjaa käyttäjän valtuutuspalvelimelle (esim. Google, Facebook) todentamaan ja myöntämään luvan.
- Käyttäjä syöttää tunnistetietonsa valtuutuspalvelimen verkkosivustolle ja myöntää kolmannen osapuolen sovellukselle luvan käyttää resurssejaan.
- Valtuutuspalvelin antaa kolmannen osapuolen sovellukselle pääsytunnuksen, jota voidaan käyttää käyttäjän resurssien käyttämiseen.
- Kolmannen osapuolen sovellus käyttää käyttöoikeustunnusta päästäkseen käsiksi käyttäjän resursseihin, mutta sillä ei ole pääsyä käyttäjän tunnistetietoihin. (DevOpsSchool 2023).

6.4 OpenID

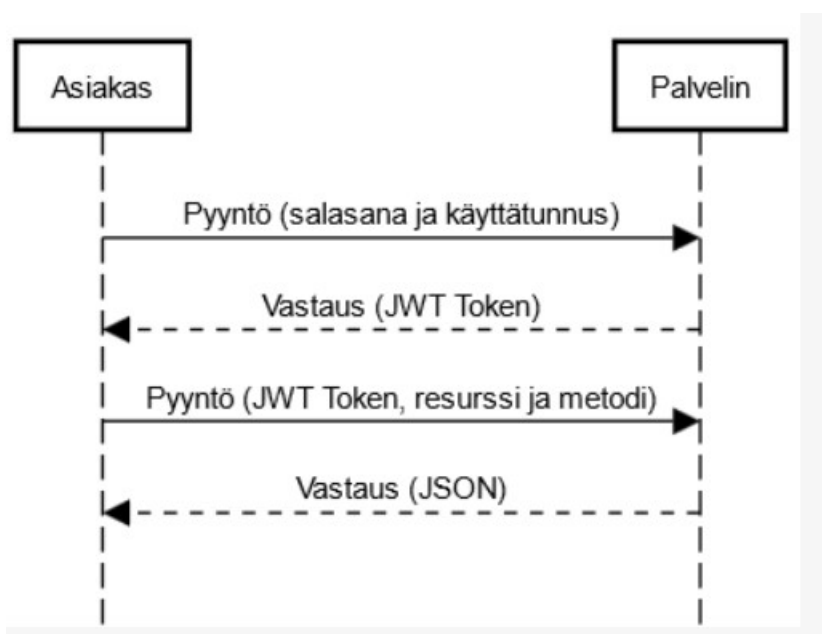
OpenID Connect (eng. OIDC) julkaistiin vuonna 2014 ja se on OAuth 2.0:n laajennus, joka lisää todennusominaisuudet valtuutuskehityksen päälle. OIDCn avulla yhteistyössä toimivat sivustot voivat todentaa käyttäjät käyttämällä kolmannen osapuolen identiteetintarjoajan palvelua, jolloin verkkosivustojen ei tarvitse tarjota omia kirjautumisjärjestelmiä käyttäjille, koska se on ulkoistettu OIDCn avulla. (Auth0 by Okta 2023)

OpenID Connect on yksinkertainen integroitavaksi perussovelluksiin, mutta siinä on myös ominaisuuksia, jotka se kelpaa turvallisuutensa puolesta myös yritysten käyttöön. Turvallisuutta lisää luetettava JWT-token, jota käytetään OpenIDn kanssa. (Auth0 by Okta 2023)

6.5 JWT-todennus

JWT (eng. JSON Web Token) voidaan käyttää REST-rajapintojen turvallisena todennusmenetelmänä. Ensin front-endin kautta käyttäjä lähettää HTTPS:llä suojattuna viestipyyntönä salasanan ja käyttäjätunnuksen back-endille, joiden ollessa kelvolliset palauttaa se palauttaa front-endin käyttöön määrääjäksi viestivastauksena JWT-tokenin.

Käyttäjä voi tokenin saatua lähettää sen back-endille pyynnön resursseihin valtuutusotsikossa olevan JWT-tokenin kanssa. Pyyntöön saatuaan back-end tarkistaa JWT-tokenin ja palauttaa pyydettyt tiedot niin kauan kuin token on vain voimassa (kuva 5). (GeeksForGeeks 2023)



Kuva 5. Onnistunut JWT-pyyntö.

7 Sovelluskokeilu: sovelluksen pohja

Tästä luvusta eteenpäin opinnäytetyössä käsitellään sovelluskokeilun toteuttamista. Ensimmäisessä sovelluskokeilua käsittelevässä luvussa 7 muodostetaan pohja tulevalle rajapinnalle ja luodaan tietokanta. Seuraava luku 8 käsittelee CRUD-toimintojen koodaamista ja luku 10 perehtyy valtuuttamiseen ja todentamiseen. Luvussa 12 lopetellaan sovelluskokeilua yhdistämällä CRUD-

toiminnot, valtuuttamis- ja todentamisominaisuudet yhdeksi kokonaisuudet. Sovellusta testataan jo sovelluskokeilun muodostamisen aikana.

Back-endin tavoite on luoda toimiva REST-rajapintatekniikkaan perustuva CRUD-toiminnot sisältävä rajapinta ja ajoympäristöksi tämän toteuttamiseen valittiin Node.js, jonka ohjelmointikielenä on JavaScript. URI-päätepisteitä resursseja varten on käytössä back-endillä useampia (taulukko 2), kuten myös niiden käyttöön olevia metodeja.

Ohjelmointiin käytän back-endin rakentamisessa Visual Studio Codea ja testaamiseen käytössä on Postman-sovellus, REST-Client VisualStudio Coden lisäosa ja Firefox-selain. En käy läpi itse Visual Studio Codea ohjelman asetusten määrittelyä optimaalisiksi.

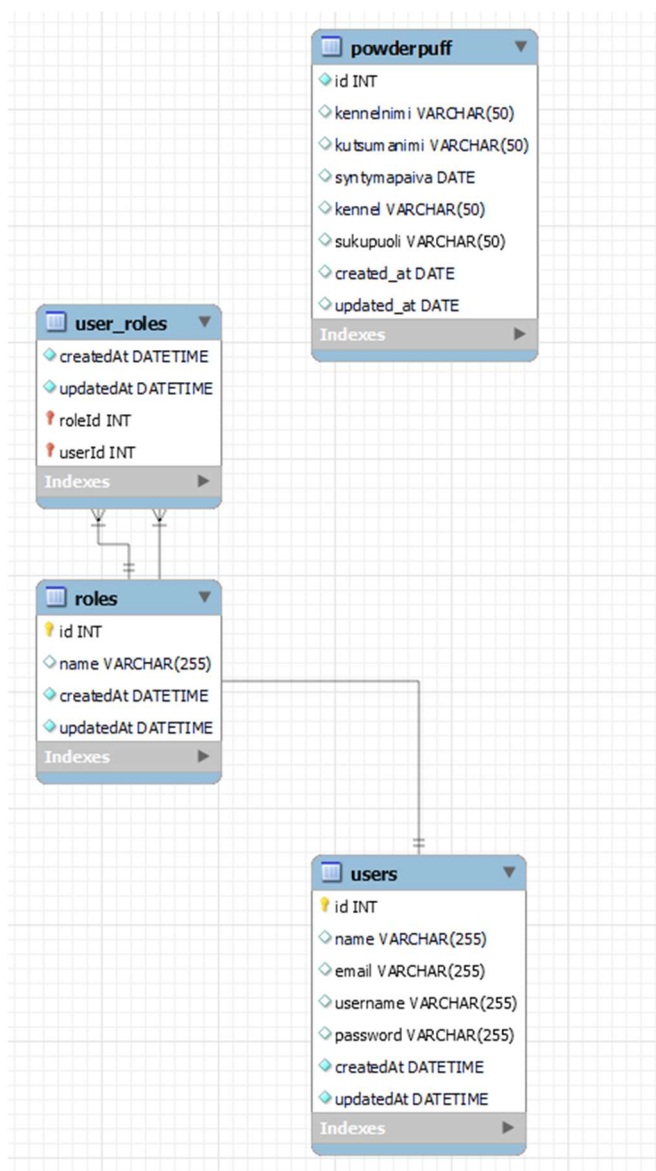
CRUD-toimintojen jälkeen lisätään back-end-sovellukseen käyttäjän rekisteröityminen, kirjautuminen, todentaminen toteutuen JWT-tokenilla ja valtuuttaminen käyttäen eri roolihierarkioita.

Polku	Sisältö
/	Etusivu tervehdys
/signin	Käyttäjän kirjautuminen
/signup	Rekisteröi käyttäjä
/test/admin ja /admin	Testaa admin tason käyttöoikeus
/test/pm	Testaa pm tason käyttöoikeus
/test/user	Testaa user tason käyttöoikeus
/api/test/all	Näytä kaikki rekisteröityneet käyttäjät
/koirat	Kaikki koirat resurssi
/koirat/:id	Yksittäinen koira tietyn id perusteella

Taulukko 2. Back-end polut.

7.1 Tietokanta

Ennen varsinaisen ohjelmoinnin aloittamista luodaan chinesecrested-tietokanta HeidiSQL-työpöytäversiossa, jossa sitä on helppo käsitellä. Tietokanta on paikallinen (eng. localhost) MYSQL-tietokanta ja luodaan siihen neljä eri taululla (kuva 6).



Kuva 6. ER-diagrammi näkymä MySQL WorkBench sovelluksessa.
Tietokannan chinesecrested neljä taulua.

Luodaan ensin powderpuff-taulu, joka sisältää koirien tietoja (kuva 7). Kenttiä on taulussa seuraavat: id (kokonaisluku), kennelnimi (merkkijono), kutsumanimi (merkkijono), syntymäpäivä (päivämäärä), kennel (merkkijono), sukupuoli (merkkijono), created_at (päivämäärä) ja updated_at (päivämäärä).

Sovelluskokeilun rajapinnan ensimmäisessä vaiheessa REST-rajapinta käyttää vain tätä taulua powderpuff ja jättää muut taulut huomioimatta.

chineseccrested.powderpuff: 8 riviä yhteensä (suunnilleen)

id	kennelnimi	kutsumanimi	syntymapaiva	kennel	sukupuoli	created_at	updated_at
32	Kennel Puff	Nelly	2023-09-04	Puff	naaras	2023-08-17	2023-08-17
38	Kennel Huisku	Murre	2022-02-20	Huisku	uros	2023-08-18	2023-08-18

Kuva 7. Taulu powderpuff.

Seuraavassa vaiheessa tarvitsemme käyttäjiin ja käyttäjien roolien tietojen sisältäviä tauluja, joita hyödynnetään valtuuttamis- ja todentamisominaisuuksien lisäämisessä sovellukseen. Valtuuttamisen ja todentamiseen liittyviä tauluja luodaan kolme. Ensimmäinen taulu users (käyttäjät) sisältää käyttäjän tiedot: id (kokonaisluku), name (merkkijono), email (merkkijono), password (merkkijono /encrypt), createdAt (päivämäärä) ja updatedAt (päivämäärä) (kuva 8).

id	name	email	username	password	createdAt	updatedAt
1	nina	nina@gma...	admin1	\$2a\$08\$AqikCUCRgVuIth3b2L...	2023-08-11 09:31:59	2023-08-11 09:31:59

Kuva 8. Taulu users.

Taulu roles (roolit) sisältää kolme eri valittavaa RBAC-roolia, jotta järjestelmän käyttäjälle voidaan määrittää valtuutettu roolitaso: id (kokonaisluku), name (merkkijono), createdAt (päivämäärä) ja updatedAt (päivämäärä) (kuva 9).

chineseccrested.roles: 3 riviä yhteensä (suunnilleen)

id	name	createdAt	updatedAt
1	USER	2023-08-11 09:24:11	2023-08-11 09:24:11
2	ADMIN	2023-08-11 09:24:11	2023-08-11 09:24:11
3	PM	2023-08-11 09:24:11	2023-08-11 09:24:11

Kuva 9. Taulu roles.

Taulu user_roles (käyttäjäroolit) sisältää tietoja: createdAt (päivämäärä), updatedAt (päivämäärä), roleId (kokonaisluku) ja userId (kokonaisluku). Taulu kertoo tiedon mikä rooli on milläkin käyttäjällä (kuva 10).

chinesecrested.user_role >> Seuraava ◆ Näytä kaikki ▼ Lajitellaan (2) ▼ Sarakkeet (4/4)

createdAt	updatedAt	roleId	userId
2023-09-15 07:48:39	2023-09-15 07:48:39	1	5
2023-09-15 07:26:50	2023-09-15 07:26:50	2	4
2023-09-15 07:20:19	2023-09-15 07:20:19	2	3
2023-08-11 09:31:59	2023-08-11 09:31:59	2	1

Kuva 10. Taulu user_roles.

Alla oleva taulukko selventää mikä on roolien hierarkia, USER on kaikkein matalimman käyttötason valtuudet sisältävä rooli ja ADMIN-rooli on taas kaikkein korkein (taulukko 3).

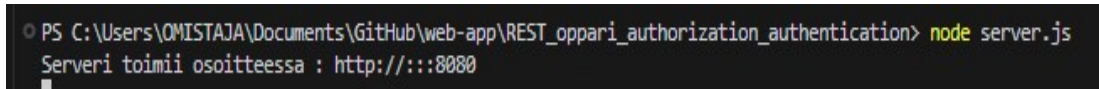
Oikeus katsoa sisältöä	USER	PM	ADMIN
USER	kyllä	kyllä	kyllä
PM	ei	kyllä	kyllä
ADMIN	ei	ei	kyllä

Taulukko 3. RBAC-roolien hierarkia järjestelmässä.

7.2 Ohjelmoinnin ensivaiheet

NodeJS back-end-sovellusta päästää rakentamaan Visual Studio Codessa aloittamalla kirjoittamalla tulevassa ohjelman kansio polussa konsolissa komento *npm init*, joka luo kansion juureen sovelluskansion ja päätiedoston (main) *server.js* (nimen voi valita luomisvaiheessa haluamakseen).

Päätiedosto on se tiedosto, joka käynnistyy, kun annamme sovellukselle käynnistymiskomennon *node server.js*. (kuva 3) Ohjelmoitsijoiden paljon käyttämä käynnistymiskomento on myös *nodemon server.js*, jolloin rajapintasovellukseen tehdyt muutokset näkyvät välittömästi sovelluksen ollessa käynnissä, eikä back-endia tarvitse sulkea ja käynnistää uudelleen.



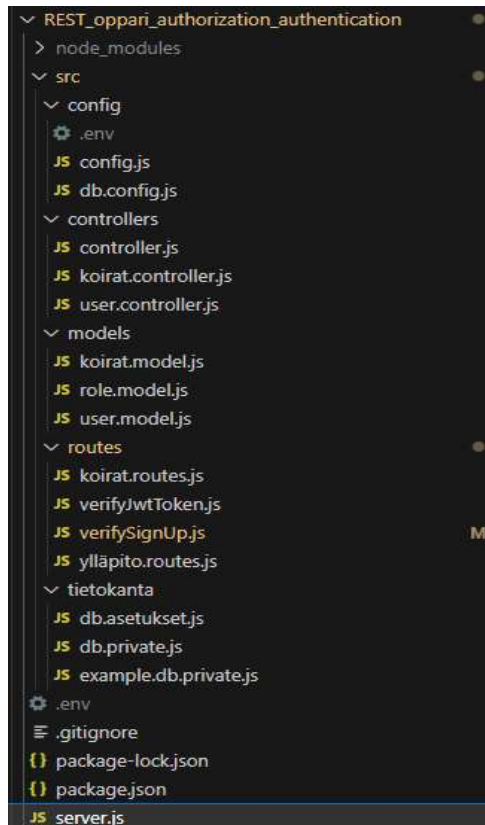
```
PS C:\Users\OMISTAJA\Documents\GitHub\web-app\REST_oppari_authorization_authentication> node server.js
Serveri toimii osoitteessa : http://:::8080
```

Kuva 11. Ohjelman käynnistyskäsky node.server.js

7.2.1 Sovelluksen rakenne

Yleinen tiedostorakenne REST-rajapinnoille on käytetty routes-, controllers- ja models-rakenne (kuva 12), jota käytetään nyt myös ehdottomasti ohjelman selkeyttämiseksi ja ylläpidon helpottamiseksi tätä rakenne, vaikka se tuottaakin lisää koodaus työtä. Routes-, controllers-, models-rakenteesta löytyy samankaltaisia variaatioita useita.

Ohjelmaa on myös helpompi testata, kun sen rakenne on mukavasti eritelty. Tämä koodi kirjoittaa lisää koodeja, mutta lopulta koodit ovat paljon helpommin ylläpidettäviä ja selkeitä.



Kuva 12. Kansiorakenteesta on nähtävissä src kansion alla oleva sovelluksen models-, controllers- ja routes-rakenne. Server.js, package.json tiedostot ja node_modules kansio ovat ohjelman juuressa.

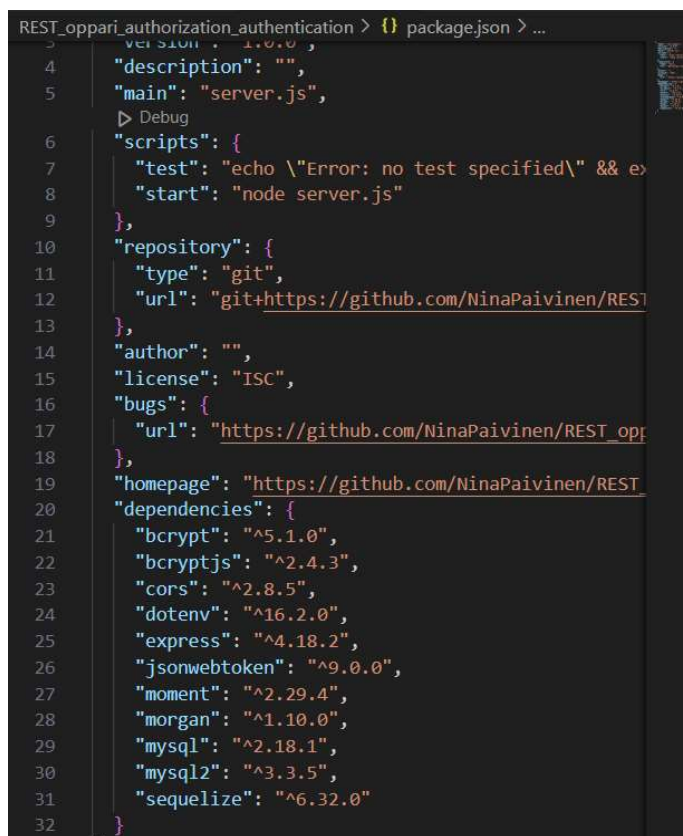
Kansioiden sisältö lyhyesti, sovelluksessa käytetään englanninkielisiä nimiä:

- ohjaimet-kansio (eng. controllers): ohjaimet käsittelevät kaiken logiikan pyyntöparametrien vahvistamisen takana; kyselyt ja vastausten lähettämiset oikeilla koodeilla.
- mallit-kansio (eng. models): sisältää tiedot funktioista eli toiminnoista; tietokantaan lisäyksen koodit ja MYSQL-kyselyt. Mallin skeeman määritelmät löytyvät myös täältä.
- reitit kansio (eng. routes): sisältää reititystiedot (eng. route maps) ja metodit resursseihin (huom. routes-kansiossa olevat tiedostot verifySignUp.js ja verifyJwtToken.js ovat monesti myös omassa middleware-kansiossa.)
- tietokanta-kansio (eng. database): sisältää tietokantaan yhdistymisen mahdollistavat ohjelmointitiedot.
- kokoonpano-kansio (eng. config/configuration): asetuksia ohjelmoinnin määrittämiseen

Ensimmäinen vaihe oli luoda rajapinta, joka yhdistyy tietokantaan ja voi suorittaa tarvittavat CRUD-toiminnot, tässä vaiheessa ilman valtuuttamista ja todentamista.

7.2.2 Riippuvuudet

Riippuvuuksia (eng. dependencies) käytetään paljon erilaisia ohjelman kanssa niin front-end kuin back-end puolellakin, jotka asentuvat node_modules kansioon ja nuo asennetut riippuvuudet näkyvät package.json tiedostossa (kuva 13), joka löytyy ohjelman juuresta.



```

REST_oppari_authorization_authentication > {} package.json > ...
3  version: "1.0.0",
4  "description": "",
5  "main": "server.js",
6  "scripts": {
7    "test": "echo \\\"Error: no test specified\\\" && ex",
8    "start": "node server.js"
9  },
10 "repository": {
11   "type": "git",
12   "url": "git+https://github.com/NinaPaivinen/REST_opp
13 },
14 "author": "",
15 "license": "ISC",
16 "bugs": {
17   "url": "https://github.com/NinaPaivinen/REST_opp
18 },
19 "homepage": "https://github.com/NinaPaivinen/REST_opp
20 "dependencies": {
21   "bcrypt": "^5.1.0",
22   "bcryptjs": "^2.4.3",
23   "cors": "^2.8.5",
24   "dotenv": "^16.2.0",
25   "express": "^4.18.2",
26   "jsonwebtoken": "^9.0.0",
27   "moment": "^2.29.4",
28   "morgan": "^1.10.0",
29   "mysql": "^2.18.1",
30   "mysql2": "^3.3.5",
31   "sequelize": "^6.32.0"
32 }

```

Kuva 13. Package.json tiedostossa näkyy riippuvuudet, jotka asennettu back-end sovellukselle (rivit 20-31). Server.js-tiedosto joka on määritetty pääkäynnistymistiedostoksi on havaittavissa rivillä viisi.

Näille jokaiselle riippuvuudelle on oma asennuskomentonsa, esimerkkinä tässä expressin asennus alkaa kirjoittamalla komentoriville komento *npm i express*.

Riippuvuus on asennettu `node_modules` kansion ja lisätty `package.json` tiedostoon saadaan se seuraavaksi ohjelmassa käyttöön komennolla `const express = require('express');`. Asennuksen jälkeen päästään aloittamaan itse riippuvuutta käyttö ohjelmassa komennolla `const app = express();`.

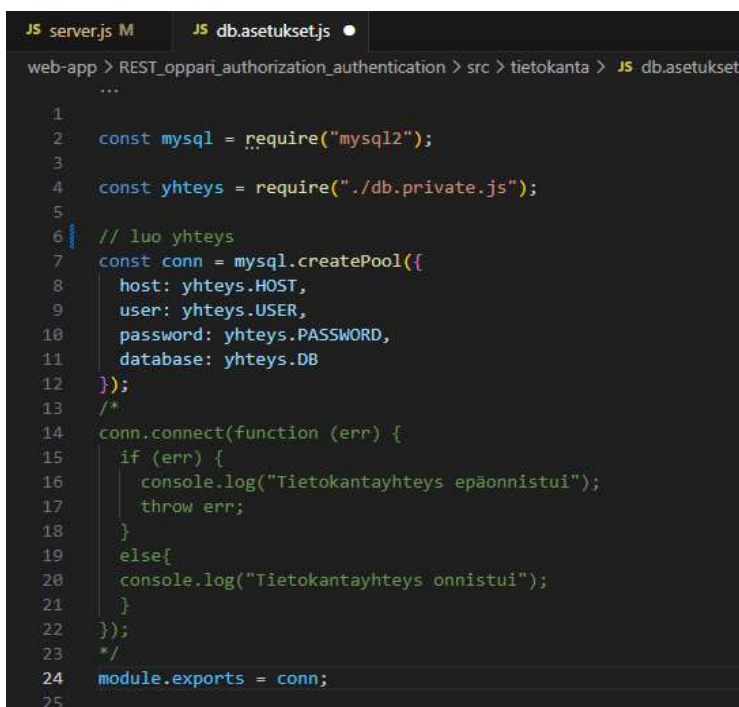
7.2.3 Tietokanta-kansio

Tiedosto `db.asetukset.js`

Luodaan seuraavaksi rajapinnassa yhteys äsken luotuun `chinesecrested`-tietokantaan. Käytin `MYSQL2`:sta perinteisen `MYSQL` sijasta sen kehittyneempien ja parempien ominaisuuksien takia. Käytetään yhteydenmuodostamis toimintoon `MYSQLn` funktiota `createPool` (kuva 14), joka tarvitsee tietokantayhteyden muodostamiseen seuraavat tiedot:

- host: 127.0.0.1 eli localhost
- user: käyttäjätunnus
- password: käyttäjän salasana
- database: tietokannan nimi
- dialect: tietokanta muoto, mysql

Toinen yleinen vaihtoehto tietokantayhteyden muodostamiseen on `createConnection`. Pool on mekanismi, joka ylläpitää tietokantayhteyden välimuistia, jotta yhteyttä voidaan käyttää uudelleen sen vapauttamisen jälkeen. Joten yhteydet tallennetaan, eikä niitä tarvitse sulkea jatkuvasti. `CreateConnection`ia käytettäessä on vain voimassa yksi tietokantayhteys ja se kestää niin pitkään päällä, kunnes se suljetaan.



```

1
2: const mysql = require("mysql2");
3
4: const yhteys = require("../db.private.js");
5
6 // luo yhteys
7 const conn = mysql.createPool({
8   host: yhteys.HOST,
9   user: yhteys.USER,
10  password: yhteys.PASSWORD,
11  database: yhteys.DB
12 });
13 /*
14 conn.connect(function (err) {
15   if (err) {
16     console.log("Tietokantayhteys epäonnistui");
17     throw err;
18   }
19   else{
20     console.log("Tietokantayhteys onnistui");
21   }
22 });
23 */
24 module.exports = conn;
25

```

Kuva 14. CreatePool funktio mahdollistaa yhteyden muodostamisen.

Tiedosto db.private.js

Äsken luotu d.b.asetukset.js tiedosto sisältää sovelluksen tietokantaan liittymiseen tarvittavat koodit ilman arkoja tietoja. Nämä arkaluontoiset tiedot löytyvät eri tiedostosta db.private.js, jota d.b.asetukset.js käyttää omassa tiedossaan *require function* avulla eli se toimii omana erillisenä moduulina. Moduulit helpottavat ohjelman koodin selkeyttämistä ja mahdollistavat ohjelman osien käyttämistä uudelleen eri kohdissa ohjelmaa.

d.b.private.js tiedosto sisältää tietokantaan yhdistimen mahdollistavat henkilökohtaiset tiedot. Yllä kuvassa esimerkki tiedosto example.db.private.js. Githubia käytettäessä tulee jättää seuraamatta oikea db.private.js tiedosto, jotta ei paljasteta henkilökohtaisia tietoja muille (kuva 15). Moduuli on itsenäinen toimintoyksikkö tarvitsee aina *module.exports* koodi lisäyksen tiedostoon, jotta sitä voidaan edelleen käyttää.


```

1  You, last month | 1 author (You)
2  You, last month * commit message ...
3  // private
4  module.exports = {
5    HOST: "127.0.0.1",
6    USER: "root",
7    PASSWORD: "SALASANA",
8    DB: "TIETOKANTA",
9    dialect: "mysql",
10   pool: {
11     max: 5,
12     min: 0,
13     acquire: 30000,
14     idle: 10000
15   }
16 };

```

Kuva 15. D.b.drivates.js esimerkki tiedosto.

8 Sovelluskokeilu: CRUD-toiminnot

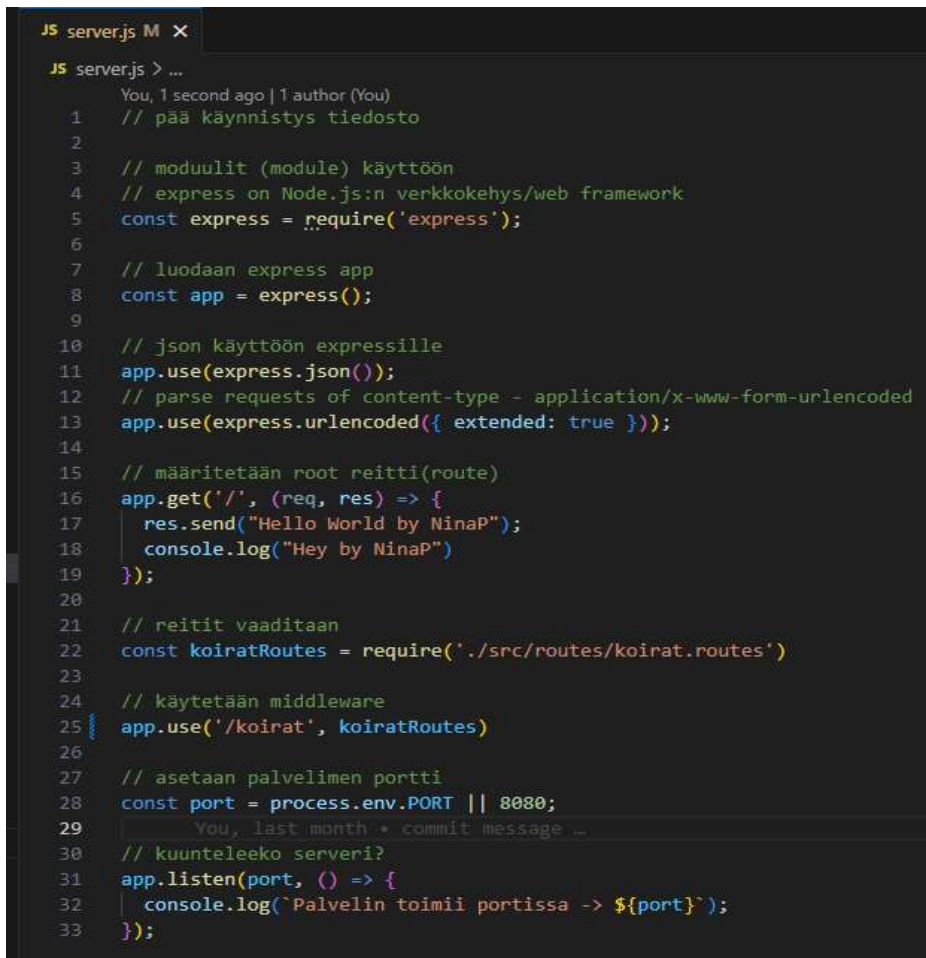
Lähdetään luomaan sovellukseen seuraavaksi mahdollisuus käsitellä eri tavoin koirat resurssia Visual Studio Codessa. Koirien tiedot ovat tietokannasta powderpuff-taulusta ja back-end-rajapintaan lisätään nyt koodia, jotta päästään manipuloimaan tuota taulua eli hyödynnetään REST-tekniikkaa ja sen HTTP-metodeja (taulukko 4).

Toiminto	Metodi
Näytä kaikki koirat	GET
Luo uusi koira	POST[id]
Näytä koira by id	GET[id]
Päivitä koira by id	PUT[id]
Poista koira by id	DELETE[id]

Taulukko 4. Koirat resurssi ja sen metodit.

8.1.1 Server.js päätiedosto

Tärkeää että päätiedosto pidetään mahdollisemman tyhjänä ja selkeänä, joka sisältää vain oleelliset koodit (kuva 16). Tiedosto toimii niin sanottuna middleware eli keskikerroksena rajapinnassa.



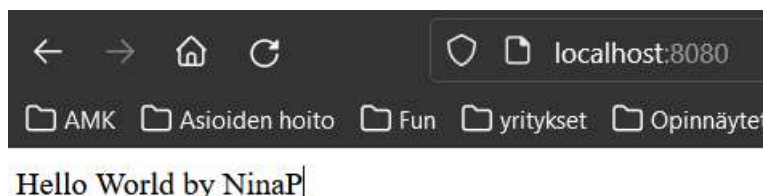
```
JS server.js M X
JS server.js > ...
You, 1 second ago | 1 author (You)
// pää käynnistys tiedosto
2
// moduulit (module) käyttöön
// express on Node.js:n verkkokehys/web framework
const express = require('express');
6
// luodaan express app
const app = express();
9
// json käyttöön expressille
app.use(express.json());
12 // parse requests of content-type - application/x-www-form-urlencoded
app.use(express.urlencoded({ extended: true }));
14
// määritetään root reitti(route)
app.get('/', (req, res) => {
16   res.send("Hello World by NinaP");
17   console.log("Hey by NinaP");
18 });
19
20
21 // reitit vaaditaan
const koiratRoutes = require('./src/routes/koirat.routes')
23
24 // käytetään middleware
app.use('/koirat', koiratRoutes)
26
27 // asetaan palvelimen portti
const port = process.env.PORT || 8080;
29 You, last month • commit message ...
// kuunteleeko serveri?
app.listen(port, () => {
31   console.log(`Palvelin toimii portissa -> ${port}`);
32 });
33
```

Kuva 16. Päätiedosto on selkeä ja tiivis.

Päätiedostoon lisätään koodia, jossa otetaan käyttöön moduuli Express-verkkokehys koodilla `const express = require('express');`. Luomisen jälkeen voidaan tehdä Express-sovelluksen esiintymä käyttäen `const app = express();`.

App.use()-funktioita käytetään seuraavaksi väliohjelmiston (eng. middleware) määrittämiseen Express-sovellukselle. Koodissa cors()-komentoa käytetään mahdollistamaan CORS-moduuli (eng. Cross-Origin Resource Sharing), joka välttämätön Expressin kanssa verkkopalvelujen WWW-palvelujen luomiseen. Lisäksi express.json()- ja express.urlencoded()-koodeja käytetään saapuvien pyyntöjen runkojen JSON- ja URL-muodon mahdollistamiseen.

Rajapinnalle asetetaan käytettäväksi koodissa oletusportti 8080. Päättiedostoon lisätään myös oletus tervehdys, joka tulee näkyviin, kun rajapinta käynnistetään ja siirrytään ohjelman juuren <http://localhost:8080/> (kuva 17).



Kuva 17. Ohjelman käynnistystervehdys ohjelman juuressa.

8.1.2 Routes-kansio

Tiedosto koirat.routes.js

Sovelluksen routes-kansiossa sijaitseva koirat.routes.js tiedosto sisältää tärkeät tiedot URI-päätepisteistä mitä rajapinta käyttää hallinnoidessaan koirat taulua ja mitä funktioita on määritelty eri URI-päätepisteille. Tiedosto on riippuvaiseksi määritelty koirat.controller.js tiedoston kanssa ja käyttääkin sen funktioita, esimerkiksi koiratController.findAll, joka hakee meille tietokannasta esille kaikki koirat taulun koirat.

Tässä vaiheessa ei huomioida väliohjelmisto (eng. middleware) toimintoja todennusta ja valtuutusta varten JWTn (JSON Web Tokens) avulla, jotka myös näkyvät.

Koodi määrittelee useita HTTP-metodeja vuorovaikutukseen koirat-resurssin kanssa (kuva 18):

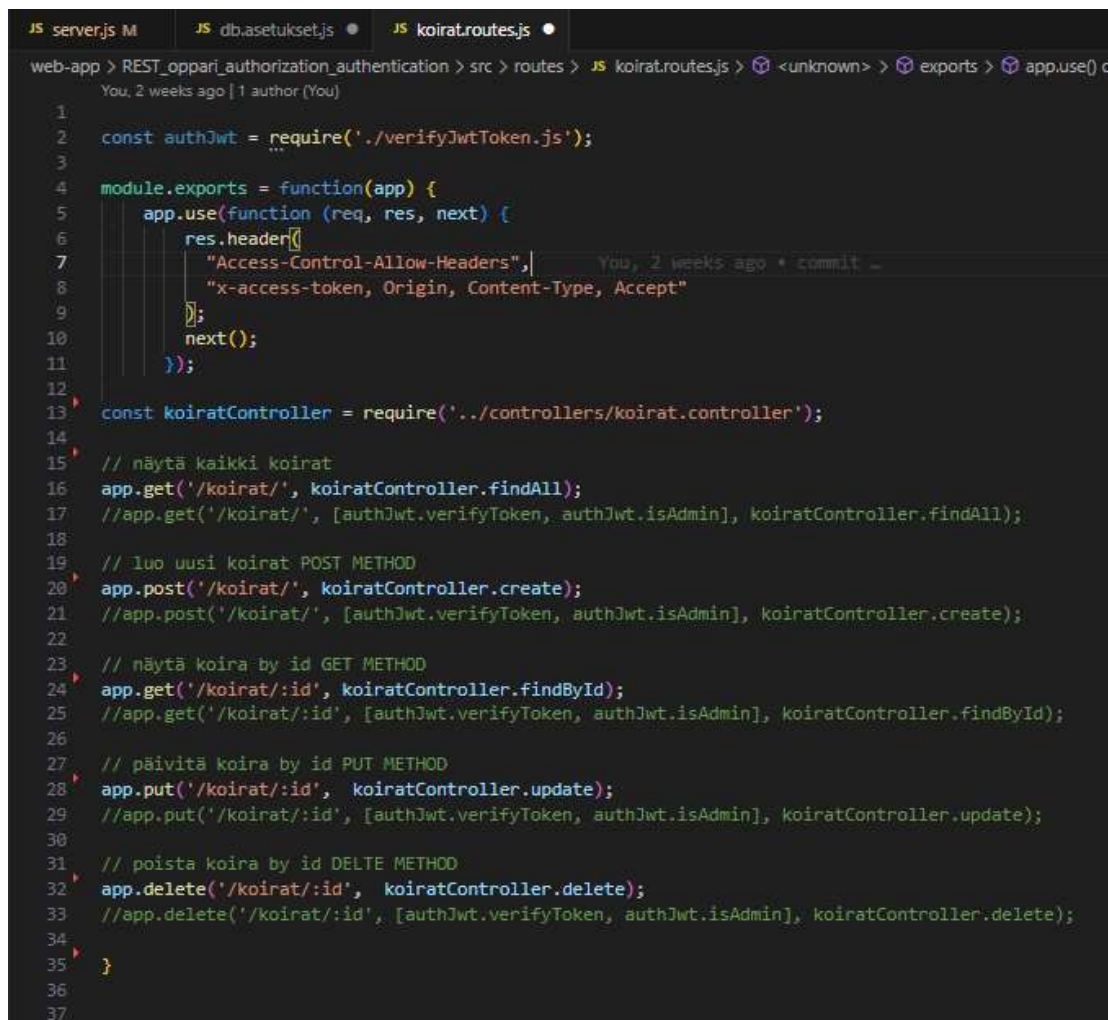
GET /koirat/: Tätä reittiä käytetään kaikkien koirien hakemiseen.

POST /koirat/: Tätä reittiä käytetään uuden koiran luomiseen.

GET /koirat/:id: Tätä reittiä käytetään tietyn koiran hakemiseen sen tunnuksen perusteella.

PUT /koirat/:id: Tätä reittiä käytetään tietyn koiran tunnuksen päivittämiseen.

DELETE /koirat/:id: Tätä reittiä käytetään tietyn koiran poistamiseen sen tunnuksella.



```

1
2 const authJwt = require('./verifyJwtToken.js');
3
4 module.exports = function(app) {
5   app.use(function (req, res, next) {
6     res.header({
7       "Access-Control-Allow-Headers": "x-access-token, Origin, Content-Type, Accept"
8     });
9     next();
10  });
11
12  const koiraController = require('../controllers/koira.controller');
13
14  // näytä kaikki koirat
15  app.get('/koirat/', koiraController.findAll);
16  //app.get('/koirat/', [authJwt.verifyToken, authJwt.isAdmin], koiraController.findAll);
17
18  // luo uusi koirat POST METHOD
19  app.post('/koirat/', koiraController.create);
20  //app.post('/koirat/', [authJwt.verifyToken, authJwt.isAdmin], koiraController.create);
21
22  // näytä koira by id GET METHOD
23  app.get('/koirat/:id', koiraController.findById);
24  //app.get('/koirat/:id', [authJwt.verifyToken, authJwt.isAdmin], koiraController.findById);
25
26  // päivitä koira by id PUT METHOD
27  app.put('/koirat/:id', koiraController.update);
28  //app.put('/koirat/:id', [authJwt.verifyToken, authJwt.isAdmin], koiraController.update);
29
30  // poista koira by id DELETE METHOD
31  app.delete('/koirat/:id', koiraController.delete);
32  //app.delete('/koirat/:id', [authJwt.verifyToken, authJwt.isAdmin], koiraController.delete);
33
34  }
35
36
37

```

Kuva 18. Koirat.routes.js tiedosto.

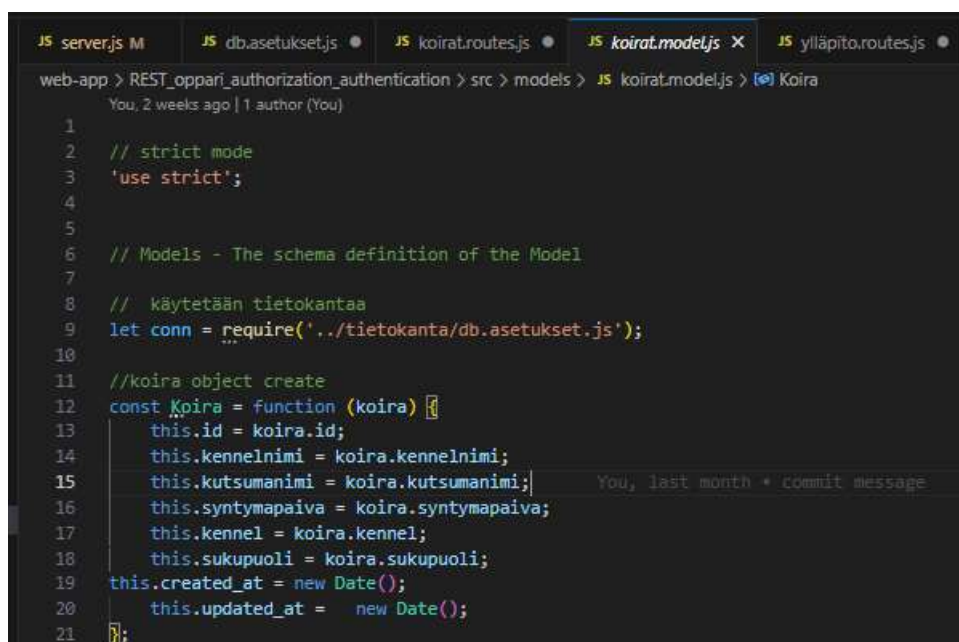
8.1.3 Models-kansio

Tiedosto koirat.models.js

Sovelluksen Models-kansiossa sijaitseva tiedosto koirat.models.js sisältää Koira-objectin ja sen funktioita, joiden avulla tietokantaa pystytään manipuloimaan. Funktiot sisältävät MYSQL-kyselyitä, joilla tietokannan tietoja pystytään käsittelemään. Koodiin on lisätty virhesanomien ilmoitus ja sovelluksessa käytetään paljon console.log toimintoa, ohjelmoinnin virheiden

etsimisen helpottamiseksi. Jos prosessin aikana tapahtuu virhe, palauttaa sovellus näkyville tapahtuman kirjatut lokitiedot tai muussa tapauksessa palauttaa onnistuneen tuloksen.

Koodi alkaa Koira-objektimallin määritelmällä (kuva 19), joka sisältää ominaisuuksia, kuten id, kennelnimi, kutsumimi, syntymäaika, kennel, sukupuoli, milloin luotu ja päivitetty mahdollisesti. Koira-konstruktorifunktiota käytetään koiramallin luomiseen annetuilla ominaisuuksilla.



```
1 // strict mode
2 'use strict';
3
4
5 // Models - The schema definition of the Model
6
7 // käytetään tietokantaa
8 let conn = require('../tietokanta/db.asetukset.js');
9
10 //koira object create
11 const Koira = function (koira) {
12   this.id = koira.id;
13   this.kennelnimi = koira.kennelnimi;
14   this.kutsumanimi = koira.kutsumanimi;
15   this.syntymapaiva = koira.syntymapaiva;
16   this.kennel = koira.kennel;
17   this.sukupuoli = koira.sukupuoli;
18   this.created_at = new Date();
19   this.updated_at = new Date();
20 };
```

Kuva 19. Koira objektin määrittäminen.

Koira-funktio on määritetty luomaan viestipyyntö eli uusi Koira-objekti (newDog) runkotietojen syötteen perusteella (kuva 20). Koira.create-toimintoa käytetään uuden koiran lisäämiseen tietokantaan. Tämän toiminnon sisällä suoritetaan SQL-kysely uuden koirayksilön tiedon lisäämiseksi tauluun powderpuff. Se ottaa parametreiksi uuden koiraobjektin ja tuloksen takaisinkutsun (eng. callback as

parameters), viestivastaus (eli koodissa result), edustaa vastausobjektia, jota käytetään halutun sisällön lähettämiseen takaisin sopivan tilakoodin kanssa.

```

//*****luo koira POST */
Koira.create = function (newDog, result) {
  // autocommit päälle
  // sql.query("SET autocommit =1");
  conn.query("INSERT INTO powderpuff set ?", newDog, function (err, res) {
    // Jos virhe haussa:
    if (err) {
      console.log("Virhe luoda uusi koira -model: ", err);
      result(err, null);
    }
    // Jos ei virhettä: You, last month * commit message
    else {
      console.log("Onnistui luoda uusi koira -model", res.insertId);
      result(null, res.insertId);
    }
  });
  // autocommit varmasti taas onhan päällä!!
  // sql.query("COMMIT");
};

```

Kuva 20. Koira.create toiminto.

Koira.getAll-menetelmä käyttää tietokantayhteysmoduulia conn SQL-kyselyn suorittamiseen kaikkien tietueiden hakemiseksi "powderpuff"-taulusta (kuva 21). Jos onnistuu, näyttää rajapinta onnistuneesta kaikkien koirien tiedot tai muuten antaa virhe sanoman.

```

//*****näytä kaikki GET*/
Koira.getAll = function (result) {
  conn.query("Select * from powderpuff", function (err, res) {
    if (err) {
      console.log("Virhe löytää kaikki koirat -model: ", err);
      result(null, err);
    }
    else {
      console.log('Kaikki koirat löytyi - model : ', res);
      result(null, res);
    }
  });
};

```

Kuva 21. Koirat.getAll toiminto.

Koirat.findById-toiminto on määritetty etsimään tietokannasta tiettyä koira sen id-tunnuksen perusteella (kuva 22). Se käyttää conn.query-menetelmää SQL SELECT-käskyn suorittamiseen tietojen hakemiseksi tietokannasta.


```

//*****findby id GET */
Koirra.findById = function (id, result) {
  conn.query("Select * from powderpuff where id = ?", id, function (err, res) {
    if (err) {
      console.log("Virhe löytää koira by id -model: ", err);
      result(err, null);
    }
    else {
      result(null, res);
      console.log("Löytyi koira by id - model: ", id);
    }
  });
};

```

Kuva 22. Koirat.findById toiminto.

Koira.update funktio toiminto on tietyn koiran tietojen muuttamista varten määritelty staattinen menetelmä, jolla on käytössään parametrit ID, koira ja result (kuva 23). Metodin sisällä se käyttää conn.query-funktiota SQL UPDATE -käselyn suorittamiseen tietokannassa. SQL-kysely päivittää powderpuff-taulun kentät koira-objektissa annetuilla uusilla arvoilla annetun ID:n perusteella.

```

//*****päivitä koira update PUT */
Koirra.update = function (id, koira, result) {
  conn.query("UPDATE powderpuff SET kennelnimi=?,kutsumanimi=?,syntymaiva=?,kennel=?,sukupuoli=?,updated_at=? WHERE id = ?",
    [koira.kennelnimi, koira.kutsumanimi, koira.syntymaiva, koira.kennel, koira.sukupuoli, koira.updated_at, id], function (err, res) {
    if (err) {
      console.log("Virhe päivittää koira by id -model: ", err);
      result(null, err);
    }
    else {
      result(null, res);
      console.log("Koiran päivitys onnistui by id - model: ");
    }
  });
};

```

Kuva 23. Koira.update toiminto.

Koira.delete on poista toiminto, jota käytetään tietyn koiran poistamiseen tietokannasta sen ID-tunnuksen perusteella (kuva 24). Se käyttää kyselyä vastaavan merkinnän poistamiseksi tietokannan powderpuff-taulusta.

```

//*****poista DELETE*/
Koirra.delete = function (id, result) {
  conn.query("DELETE FROM powderpuff WHERE id = ?", [id], function (err, res) {
    if (err) {
      console.log("Virhe poistaa koira by id - model: ", err);
      result(null, err);
    }
    else {
      result(null, res);
      console.log("Koiran poistettu onnistui by id - model: ");
    }
  });
};
module.exports = Koirra;

```

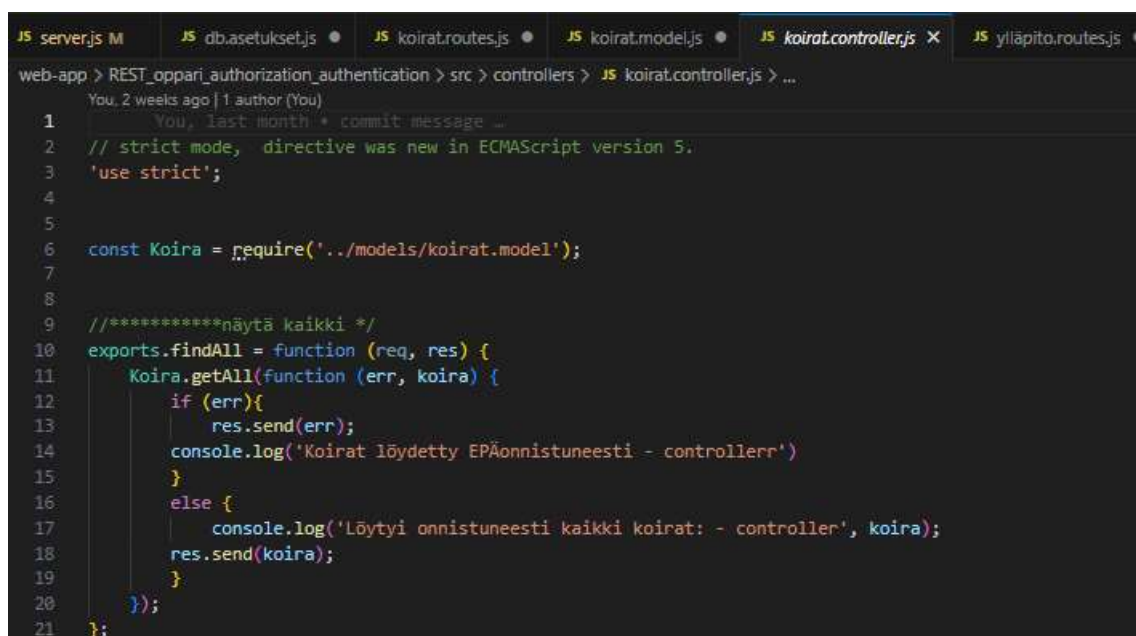
Kuva 24. Koira.delete toiminto.

8.1.4 Controllers-kansio

Tiedosto koirat.controllers.js

Sovelluksen Controllers-kansiossa sijaitseva koirat.controllers.js moduuli tiedosto sisältää logiikan Koira-mallin CRUD-metodien käsittelyyn. Ensimmäisenä koodissa on määritetty "user strict", joka on ECMAScript-versiossa 5 käyttöön otettu ominaisuus, jonka avulla kehittäjät voivat valita rajoitetun JavaScript-version.

Seuraavaksi koodiin tuodaan käyttöön koirat.model.js tiedoston moduuli. Exports.findAll-toiminto on ohjain-funktio (eng. controller-function) kaikkien koiratietueiden hakemiseen liittyvien pyyntöjen käsittelemistä varten, jotta saadaan kaikkien koirien tiedot näkyviin. Kaikkien koirien näyttämistä varten tietokannasta on taas Koira.getAll-funktio, joka hakee kaikki tietueet tietokannan powderpuff-taulusta (kuva 25). Lyhyesti ensin koodissa on ohjainlogiikka, jonka jälkeen on tietokannan käsittelytoiminnot.



```
JS server.js M JS db.asetukset.js JS koirat.routes.js JS koirat.model.js JS koirat.controller.js X JS ylläpito.routes.js
web-app > REST_oppari_authentication_authentication > src > controllers > JS koirat.controller.js > ...
You, 2 weeks ago | 1 author (You)
You, last month • commit message ...
1 // strict mode, directive was new in ECMAScript version 5.
2 'use strict';
3
4
5
6 const Koira = require('../models/koirat.model');
7
8
9 //*****näytä kaikki */
10 exports.findAll = function (req, res) {
11   Koira.getAll(function (err, koira) {
12     if (err){
13       res.send(err);
14       console.log('Koirat löydetty EPÄonnistuneesti - controllerr')
15     }
16     else {
17       console.log('Löytyi onnistuneesti kaikki koirat: - controller', koira);
18       res.send(koira);
19     }
20   });
21 };
```

Kuva 25. Koirat.controller.js tiedosto ja ohjaintoiminto kaikkien koirien tietojen näyttämiseen.

Uuden koiran lisäämistä varten on create-funktio, jolla on req (request eli pyyntö) ja res (response eli vastaus) parametrit (kuva 26). Create.funktion sisällä luodaan uusi Koira-mallin esiintymä käyttämällä pyyntörungon (req body) tietoja. Koodiin on lisätty virheen tunnistus tyhjää pyyntörunko varten, jolloin antaa sovellus virhesanoman, kun taas jos pyynnönrunko ei ole tyhjä, Koira.create-metodia kutsutaan tallentamaan uusi koira tietokantaan.

```

22 //*****luo koira */
23 exports.create = function (req, res) {
24   const new_dog = new Koira(req.body);
25   // Jos käyttäjä syöttää tyhjiä kenttiä
26   if (req.body.constructor === Object && Object.keys(req.body).length === 0) {
27     res.status(400).send({ error: true, message: 'Kaikki kentät vaaditaan' });
28   } else {
29     Koira.create(new_dog, function (err, koira) {
30       if (err){
31         res.send(err);
32         console.log('Koira lisätty EPÄonnistuneesti - controller')
33       }
34       else {
35         res.json({ error: false, message: "Koira lisätty onnistuneesti - controller", data: koira });
36       }
37       console.log("Koira lisätty onnistuneesti- controller");
38     });
39   }
40 }
41 };

```

Kuva 26. Create-ohjaintoiminto uuden koiran lisäämistä varten.

Koiraa etsittäessä tietokannasta on findById-ohjainfunktio, joka käyttää Koira-mallin findById-toimintoa tietyn koiran hakemiseen tietokannasta tietyn ID:n perusteella (kuva 27). ID-tunnus välitetään parametrina pyyntöobjektissa (req.params.id), jos koira löytyy määritetyllä ID:llä, koira-muuttujaksi asetetaan löytynyt kyseinen koira ja res-objektilla lähetetään löytynyt koira JSON-muodossa vastauksena takaisin.

```

//***** find by id */
exports.findById = function (req, res) {
  Koira.findById(req.params.id, function (err, koira) {
    if (err){
      res.send(err);
      console.log('Koirat löydetty EPÄonnistuneesti by id - controllerr')
    }
    else{
      res.json(koira);
      console.log("Koira löytyi onnistuneesti by id - controller", koira);
    }
  });
};

```

Kuva 27. FindById-ohjaintoiminto tietyn koiran etsimistä varten ID:n perusteella.

Exports.update on ohjainfunktio, joka toteuttaa tietyn ID-tunnuksen perusteella olevan koiran tietojen päivittämisen (kuva 28). Ohjainfunktio kutsuu Koira.update()-metodia, jos pyyntö on vain kelvollinen. Metodissa on määritetty kyseisen päivitettävän Koiran-objektin id-tunnus parametriksi req.params.id, kaksi muuta metodia jotka ovat update-metodissa ovat uusi Koira-objekti päivitettyillä kentillä pyynnönrunгон tietoihin perustuen ja takaisinsoittotoiminto funktio.

Funktio luo sitten uuden Koira-objektin pyynnönrunгон perusteella ja välittää sen toisena parametrina Koira.update()-metodille. Koira-objekti luodaan uudella Koira(req.body)-syntaksilla, joka asettaa objektin ominaisuudet pyynnönrungonssa oleviin arvoihin/kenttiin.

Onnistuneessa päivityksessä takaisinsoittotoiminto lähettää onnistuneen vastauksen takaisin käyttämällä res.json()-funktia.

```

55 //*****päivitä */
56 exports.update = function (req, res) {
57   if (req.body.constructor === Object && Object.keys(req.body).length === 0) {
58     res.status(400).send({ error: true, message: 'Kaikki kentät vaaditaan' });
59   } else {
60     Koira.update(req.params.id, new Koira(req.body), function (err, koira) {
61       if (err){
62         res.send(err);
63         console.log('Koirat päivitetty EPÄonnistuneesti by id - controllerr')
64       }
65       else {
66         res.json({ error: false, message: 'Koira päivitetty onnistuneesti - controller'})
67         console.log("ID: ", req.body.id)
68         // req.body palauttaa pyynnön kutsusta osan: body
69         console.log("Koira päivitetty onnistuneesti by id - controller", req.body);
70       }
71     });
72   }
73 };

```

Kuva 28. Update-ohjain toiminto koiran tietojen päivittämistä varten.

Delete-ohjainfunktio on tietyn koira tietueen poistamiseksi tietokannasta ID perusteella (kuva 29). Ohjainfunktio kutsuu metodia Koira.delete() req.params.id parametrilla. Toiminto poistaa tietueen taulusta onnistuneesti Koira-objektin ID-tunnuksen perusteella, jos se vain löytyy tietokannasta.

```

74 //***** poista */
75 exports.delete = function (req, res) {
76   Koira.delete(req.params.id, function (err, koira) {
77     if (err) {
78       res.send(err);
79
80       console.log('Koirat poistettu EPÄonnistuneesti by id - controllerr')
81     }
82     else { // You, last month * commit message
83       res.json({ error: false, message: 'Koira poistettu onnistuneesti - controller' });
84       console.log("Koira poistettu onnistuneesti by id - controller. ID:", req.params.id);
85     }
86   });
87 };

```

Kuva 29. Delete-ohjainfunktio koiran poistamista varten.

9 Testaus: CRUD-toiminnot

Ohjelmoinnin ensimmäinen koodin tuotanto vaihe on suoritettu ja aika testata luotuja toimintoja:

- luo uusi koira
- päivitetään koira IDn perusteella
- poistetaan koira IDn perusteella
- katsotaan kaikki koirat
- katsotaan yksi tietty koira IDn perusteella

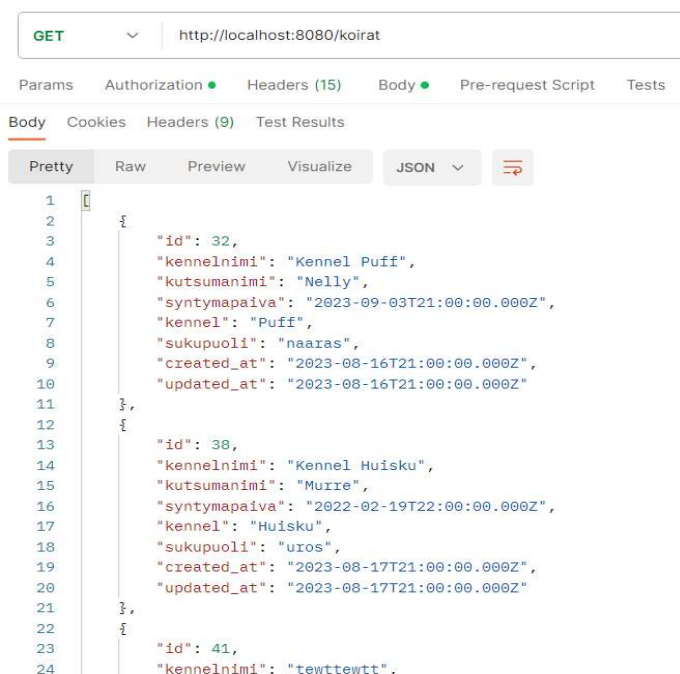
Postman on testaus käyttöön erinomaisesti soveltuva työpöytäohjelma. Lisäksi käytetään helppokäyttöistä ja nopeaa REST Client Visual Studio Coden lisäosaa testaukseen, jota käyttöä varten tehdään ohjelman juureen tiedosto `client_syntax.http`.

Testauksen tarkoituksena on syöttää oikeat parametrit, tarkoituksella väärät ja puutteelliset, jonka jälkeen seurata sovelluksen reagointia eri tilanteissa.

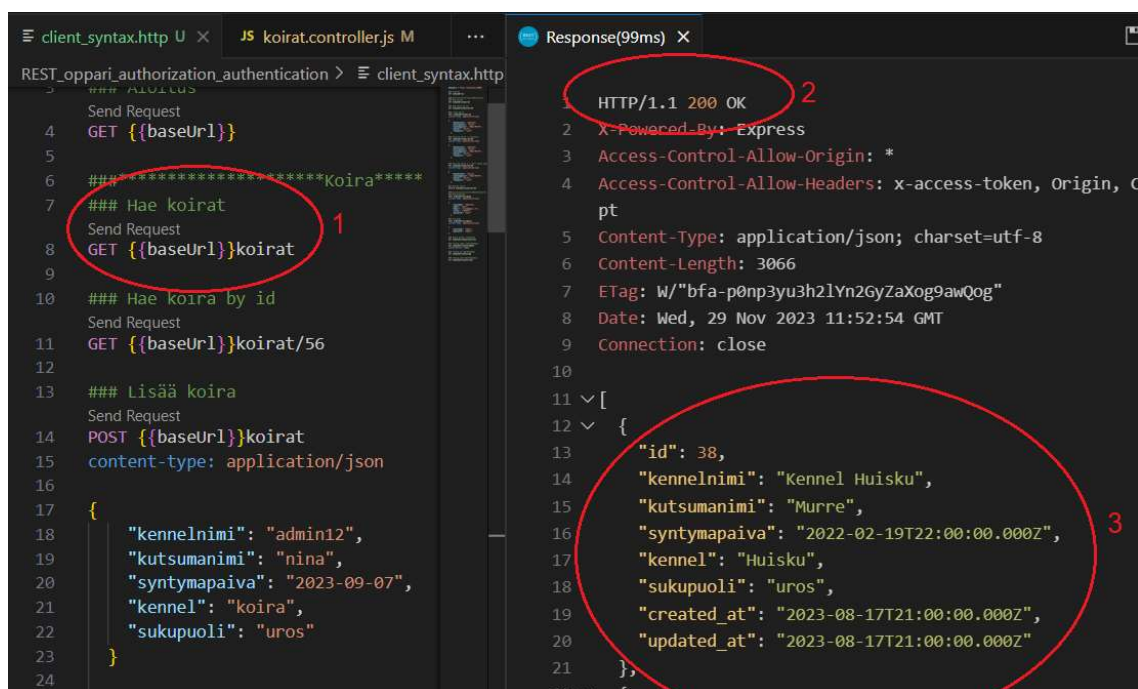
9.1.1 Testi: GET-pyyntö

Tehdään testi HTTP GET-pyyntöä rajapinnalle URI-päätepisteeseen <http://localhost:8080/koirat/> (kuva 30). Kaikkien koirien tietojen tulisi palautua ja

testi menee läpi onnistuneesti Postman sovelluksessa, koirien tiedot ja tilakoodi 200 OK palautuu. REST-Clientissa testi onnistuu myös samoilla tuloksilla ja vastausajaksi antaa 99ms (kuva 31).



Kuva 30. Postman GET-pyyntö /koirat polkuun.



Kuva 31. REST-Client testaamista varten oleva tiedosto client_syntax.http ja testi käsky GET {{baseUrl}}koirat/ (punainen ympyröity kohta 1), palauttaa

statuskoodin 200 OK (kohta 2) ja kaikkien koirien tiedot pyynnön vastauksena (kohta 3).

Visual Studio Coden puolella näemme konsoliin tulostuneet syötetiedot, josta myös käy ilmi tulostuneet koirat kahden eri tiedoston kautta, eli koirat.model.js ja koirat.controller.js, jotka siis molemmat toimivat (kuva 31 ja kuva 32).

```
Kaikki koirat löytyi - model : [
  {
    id: 32,
    kennelnimi: 'Kennel Puff',
    kutsumanimi: 'Nelly',
    syntymaiva: 2023-09-03T21:00:00.000Z,
    kennel: 'Puff',
    sukupuoli: 'naaras',
    created_at: 2023-08-16T21:00:00.000Z,
    updated_at: 2023-08-16T21:00:00.000Z
  },
  {
```

Kuva 31. Koirat.model.js tiedoston syöte konsoliin.

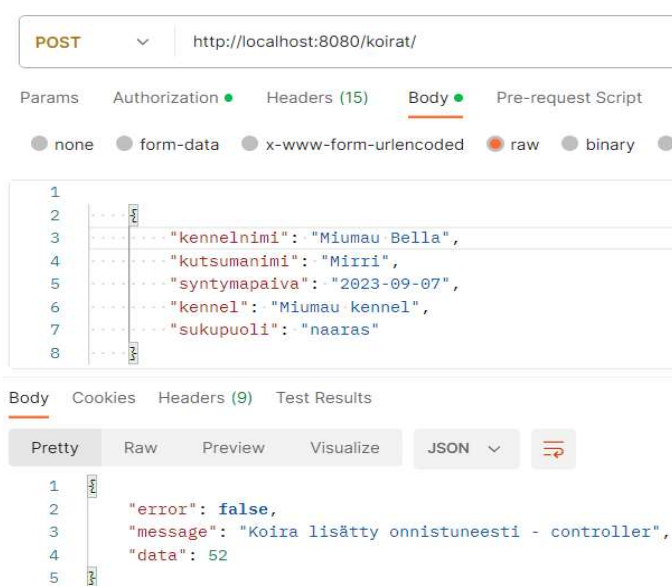
```
]
Löytyi onnistuneesti kaikki koirat: - controller [
  {
    id: 32,
    kennelnimi: 'Kennel Puff',
    kutsumanimi: 'Nelly',
    syntymaiva: 2023-09-03T21:00:00.000Z,
    kennel: 'Puff',
    sukupuoli: 'naaras',
    created_at: 2023-08-16T21:00:00.000Z,
    updated_at: 2023-08-16T21:00:00.000Z
  },
  {
    id: 38,
```

Kuva 32. Koirat.controller.js tiedoston syöte konsoliin.

Console.log tiedoston runsas käyttö koodin seassa auttaa näin konkreettisesti havainnoituna selventämään mahdollisia vika tilanteita helpommin, esimerkiksi tilanteessa jossa on epäselvää missä moduulissa ei haluttuja tuloksia synny voidaan console.log toiminnolla tutkia tilannetta. En jatkossa käsittele tätä demonstraatiota enempää console.login käyttöä.

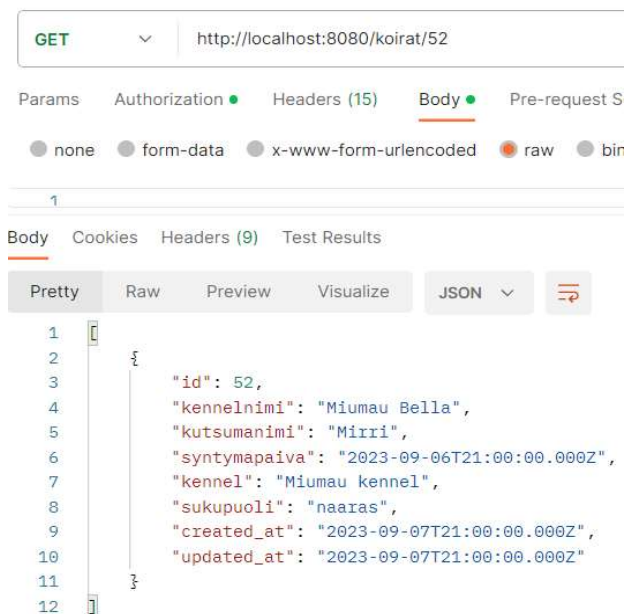
9.1.2 Testi POST

Tehdään testi POST eli lisää uuden koiran tiedot pyyntö palvelimelle päätepisteeseen: <http://localhost:8080/koirat/> Tulisi viestipyynnön luoda uusi koira koirat-tauluun (kuva 33). Halutut tiedot syötetään viestipyynnön runkosassa ja lähetetään rajapinnalle käsiteltäväksi. Päivämäärän kirjoitusmuoto on tärkeä tässä vaiheessa huomioida, että se tulee oikeassa muodossa, muut arvot ovatkin merkkijonoja ja koiran IDtä joka on kokonaisluku (muistutuksena integer) ei tarvitse syöttää, koska se on AUTO_INCREMENT eli muodostuu automaattisesti. Viestivastaus ilmoittaa pyynnön käsitellyksi hyväksytysti ja palautti statuskoodin 200 OK.



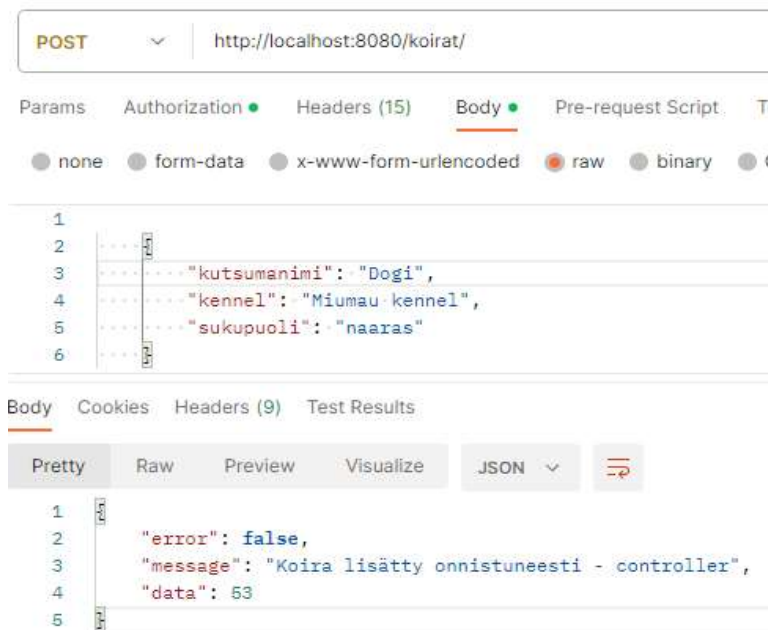
Kuva 33. POST eli pyyntö lisätä uusi koira tietokantaan, joka onnistuu.

Tarkistetaan vielä GET pyynnöllä äskeisen koiran lisäys, jonka IDksi tuli 52 ja viestipyyntö näyttää halutun viestivastauksen eli koiran kyseisellä IDllä, jolla on samat arvot kuin koiralle joka luotiin (kuva 34). Kohta created_at eli arvoltaan päivämäärä muodostuu automaattisesti, kun luodaan uusi koira eli osoittaa koiran tietueen luomispäivämäärän. GET-viestipyyntö palauttaa statuskoodin 200 OK ja äsken luodun koiran oikeilla tiedoilla näkyviin. Jos käyttäjä yrittää syöttää IDn joka on jo olemassa, tallentuu duplikaatti-IDllä olevat koirat tietokantaan, tässä kohtaa koodia tulisi vielä jatko jalostaa, ettei se ole mahdollista.



Kuva 34. Tarkistettu GET pyynnöllä äskeisen koiran lisäys, jonka ID oli 52. Pyyntö onnistuu ja vastaus on oletettu.

Jos käyttäjä yrittää syöttää koiraa, josta puuttuu joitakin kenttiä kokonaan, koira tietue tallentuu kuitenkin tietokantaan (kuva 35). Koiran kentät joita käyttäjä ei ole syöttänyt ollenkaan tallentuvat (NULL)-arvoina eli tyhjinä (kuva 36), koska se on sallittua ja statuskoodi 200 OK palautuu. Full-Stack-sovellusta rakentaessa front-endin puolella voidaan tämäkin ratkaista niin, että määritetään saako käyttäjä lomakkeessa syöttää tyhjiä kenttiä vai ei. Ongelma on myös back-endinkin puolella jalostamalla koodia vielä lisää.



Kuva 35. POST-pyyntö, jossa viestipyynnön rungossa ei ole kaikkia arvoja annettu tallentuu tietue kuitenkin

53 (NULL)	Dogi	(NULL)	Miumau kennel	naaras	2023-09-08	2023-09-08
-----------	------	--------	---------------	--------	------------	------------

Kuva 36. NULL-arvot näkyvät tietokannassa tietueen tyhjinä kenttinä-

Kokeillaan syöttää tuplatietoja samoilla koiran arvoilla, tämäkin tallentuu tietokantaan (kuva 37). Voitasi koodia jalostamalla poistaa mahdollisuus esimerkiksi duplikaatti-ID syötteiden antamiseen.

38	Kennel Huisku	Murre	2022-02-20	Huisku	uros	2023-08-18	2023-
38	Kennel Huisku	Murre	2022-02-19	Huisku	uros	2023-09-08	2023-

Kuva 37. Jos käyttäjä yrittää syöttää duplikaatti ID onnistuu sekin.

SQL ei salli ollenkaan virheellisiä arvoja siellä kuulumattomassa paikassa, esimerkkinä virheellinen data-arvo estää tietojen tallentumisen kokonaan tietokantaan (kuva 38). Yritetään syöttää päivämäärän arvo satunnaisena kokonaislukuina ja tämä on virheellinen päivämäärä muoto, eikä täten tallennuttu tietokantaan, vaikka statuskoodi onkin 200 OK.

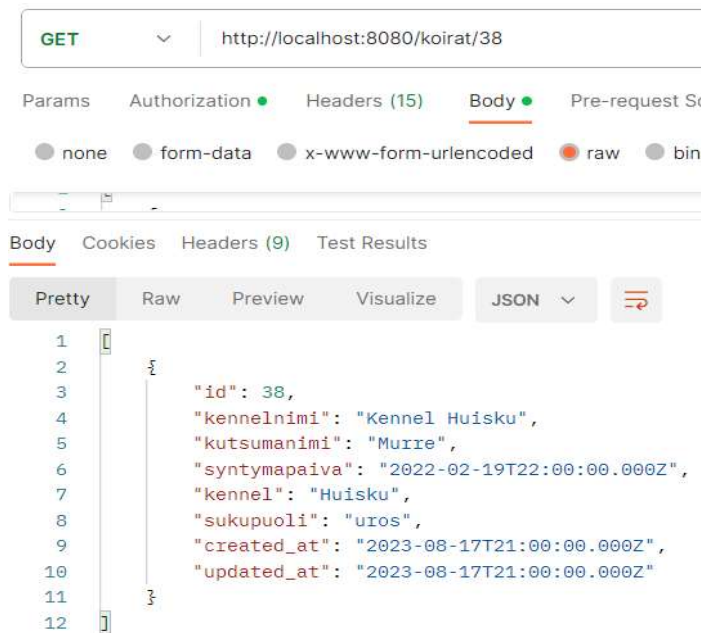


Kuva 38. Virheellinen päivämäärä tieto koira-tietueessa estää tallentumisen tietokantaan ja SQL antaa oman virhepalautteen asiasta.

9.1.3 Testi GET:ID

Tehdään testi GET:ID eli näytä tiedot pyyntö palvelimelle päätepisteeseen: <http://localhost:8080/koirat/38> (kuva 39). Koiran tiedot IDllä 38 tulisi vain tulla näkyviin ja testi onnistuukin koska kyseisellä IDllä on tietokannassa koira ja statuskoodi 200 OK palautuu.

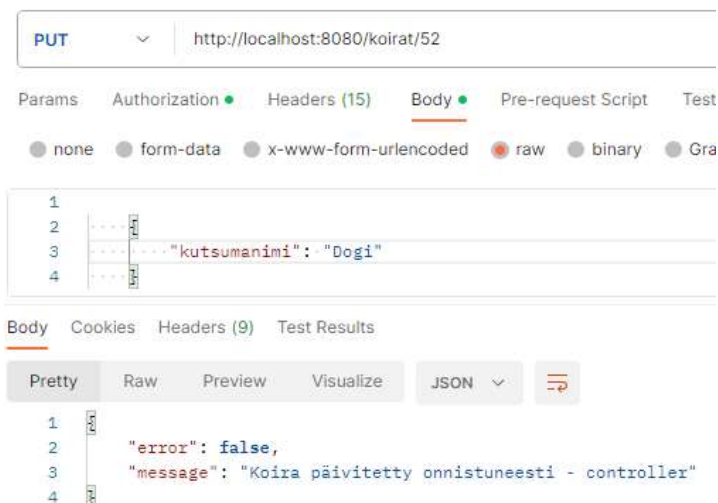
Koodista havaitaan kehittämisen paikka, koska haettaessa koira IDllä jota ei ole olemassa koodi palauttaa konsoliin console.login runsaan käytön seurauksena näkyviin tyhjän taulukon koirat.controller.js tiedostosta [] ja statuskoodin 200 OK, vaikka tietokannassa ei ole koira kyseisellä IDllä.



Kuva 39. GET-pyyntö tietyllä ID:llä olevan koiran näyttämiseksi tietokannasta ja kyseisellä ID:llä löytyykin koira.

9.1.4 Testi PUT:ID

Tehdään PUT eli päivitä tiedot viestipyyntö palvelimelle päätepisteeseen: <http://localhost:8080/koirat/52> (kuva 40). Tulisi äsken luodun koiran, jonka ID oli 52 tiedot päivittyä, joita kohtia halutaan muuttaa. Päivitetään kutsumanimi: "Dogi" kentäksi ja lähetetään se viestipyyntö runko-osassa. Viestipyyntö menee läpi ja palauttaa statuskoodin 200 OK. Ilmoitus onnistumisesta tulee, mutta HeidiSQL paljastaa, että muut kentät ovat muuttuneet NULL-kentiksi (kuva 41). Tämä vaatisi koodauksessa jatkojalostusta.

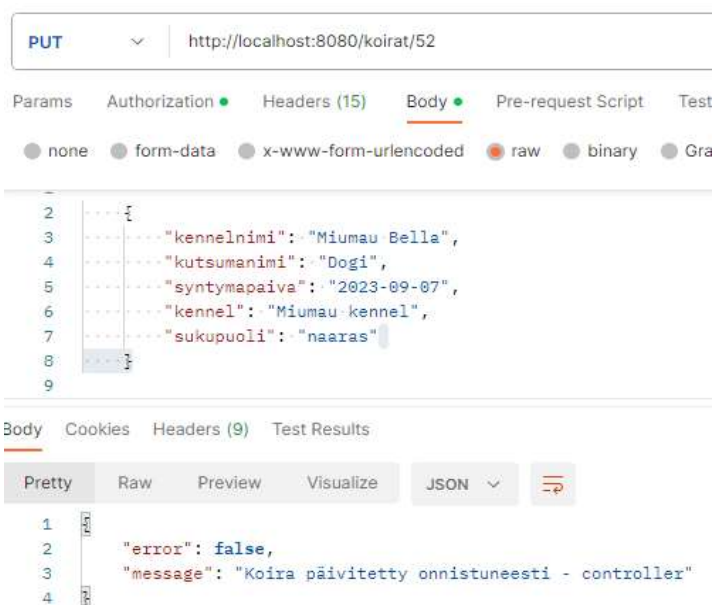


Kuva 40. Päivitetään pelkkä kutsumanimi kenttä.



Kuva 41. PUT-viestipyyntö muutti muut kentät NULL-arvoiksi, paitsi kentän johon vaihdettiin arvo.

Lähetetään seuraavaksi viestipyyntöön rungossa kaikki koiran tiedot, myös nekin joita ei muokata ja seurataan ohjelman vastetta. Viestipyyntö hyväksytään statuskoodilla 200 OK ja tiedot ovat nyt päivittyneet onnistuneesti tietokantaan (kuva 42 ja kuva 43).



Kuva 42. Onnistunut koiran tietojen päivitys.

52	Miumau Bella	Dogi	2023-09-07	Miumau kennel	naaras	2023-09-08	2023-09-08
----	--------------	------	------------	---------------	--------	------------	------------

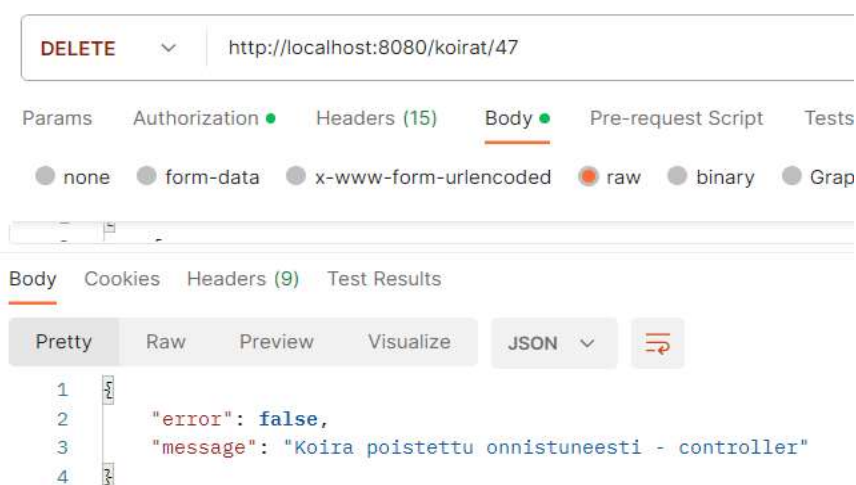
Kuva 43 HeidiSQL:stä katsotaan ja tarkastetaan vielä onnistunut päivitys

Updated_at päivämäärä arvo kenttä päivittyy myös onnistuneesti, kun tehdään PUT viestipyyntö onnistuneesti koiralle tietyllä IDllä. Kyseinen päivämäärä kertoo, onko koiran tietoja päivitetty milloin mahdollisesti.

9.1.5 Testi DELETE:ID

Tehdään testi DELETE eli poista tiedot viestipyyntö URI-päätepisteeseen: <http://localhost:8080/koirat/47>. Tulisi koiran ID numerolla 47 poistua ja testi onnistuukin ja palauttaa statuskoodin 200 OK (kuva 44). HeidiSQL:stä tarkistetaan myös ja katsottuna koira kyseisellä IDllä on näyttää olevan poistunut taulusta.

Koodissa havaitaan jatko jalostuksen paikka, kun yritetään antaa viestipyyntö DELETE koiraan IDllä, jota ei ole olemassa, palautuu statuskoodilla 200 OK, vaikka mitään poistettavaa ei ole kyseisessä resurssissa.



Kuva 44. DELETE-viestipyyntö koirasta IDllä 47.

10 Sovelluskokeilu: todentamisen ja valtuuttamisen

10.1.1 Sovelluksen juuri

Tiedosto server.js

Pääkäynnistystiedostoon eli server.js lisätään valtuuttamiseen ja todentamiseen liittyviä koodeja, ensimmäisenä tuodaan ylläpitoreititys-moduuli koodilla `require("./src/routes/ylläpito.routes.js")(app)` päätiedoston käyttöön.

Reititysmoduuli sisältää oleelliset ylläpitoon liittyvät URI-päätepiisteet ja niiden käyttöön soveltuvat toiminnot.

Koodi `const db = require('./src/config/db.config.js');` tuo moduulin, joka määrittää Sequelize-tietokantayhteyden ja määrittää tietokantamallit käyttäjille ja rooleille. Lisätään `const Role = db.role;` koodi, jossa role-objektia käytetään db-objektissa ja näin ollen mahdollistetaan role-objektin käyttää äsken tuotua Sequelize-tietokantayhteyttä.

Sequelize on Object-Relational Mapping (ORM) -kirjasto Node.js:lle, joka tarjoaa helpon pääsyn useisiin tietokantoihin, kuten MySQL, PostgreSQL, SQLite ja muihin, se mahdollistaa rajapintojen vuorovaikutuksen tietokantojen kanssa ja CRUD-toimintojen suorittamisen helpommin. Sequelize yksinkertaistaa tietokantavuorovaikutusta sallimalla kehittäjien työskennellä objektien ja -funktioiden kanssa sen sijaan, että he kirjoittaisivat SQL-kyselyitä.

Sovellus käyttää RBAC eli roolipohjaista käyttöoikeuksien hallintaa, joka luo rooleja Role-objektin avulla. Funktiota `initial()` kutsutaan luomaan kolme roolia: "USER", "ADMIN" ja "PM".

```

55 require("../src/routes/ylläpito.routes.js")(app);
56
57 const db = require('../src/config/db.config.js');
58
59 const Role = db.role;
60
61 function initial(){
62   Role.create({
63     id: 1,
64     name: "USER"
65   });
66
67   Role.create({
68     id: 2,
69     name: "ADMIN"
70   });
71
72   Role.create({
73     id: 3,
74     name: "PM"
75   });
76 }

```

Kuva 45. Päättiedostoon lisätty koodi RBAC-hallintaa varten.

10.1.2 Routes-kansio

Tiedosto ylläpito.routes.js

Routes-kansiossa oleva ylläpito.routes.js reititystiedosto kertoo meille samalla tavalla kuin koirat.routes.js rajapinnan toiminnan kannalta tärkeät URI-päätepisteet ja niiden metodit ja funktiot joita käytetään.

Tiedosto on riippuvainen controller.js ja user.controller.js ohjaintiedostojen moduuleista, jotka käsittelevät HTTP-metodien käytön logiikan. Myös middleware-tiedosto verifySignUp.js, joka käsittelee käyttäjän valtuuttamisen ja todentamisen on yhteydessä tähän tiedostoon.

Koodi määrittelee useita HTTP-metodeja vuorovaikutukseen ylläpitoon liittyvien resurssin kanssa:

POST /signup/: Tätä reittiä käytetään käyttäjän rekisteröimiseen

POST /signin/: Tätä reittiä käytetään käyttäjän kirjautumiseen

GET /test/user: Tätä reittiä käytetään USER-roolitason testaamista varten

GET /test/pm: Tätä reittiä käytetään PM-roolitason testaamista varten

GET /test/admin: Tätä reittiä käytetään ADMIN-roolitason testaamista varten

GET /api/test/all: Tätä reittiä käytetään kaikkien rekisteröityneiden käyttäjien näyttämiseen

```

18
19  /******REKISTERÖIDY*/
20
21  app.post('/signup', [verifySignUp.checkDuplicateUserNameOrEmail, verifySignUp.checkRolesExisted], controller.signup);
22
23  /* POSTMAN - POST - http://localhost:8080/signup
24  {
25    "username": "admin",
26    "name": "nina",
27    "email": "fddgagaa",
28    "roles": ["ADMIN"],
29    "password": "admin"
30  }
31
32  /******KIRJAUDU
33  */
34  app.post('/signin', controller.signin);
35
36  /* POSTMAN - POST - http://localhost:8080/signin
37  {
38    "username": "admin",
39    "password": "admin"
40  }
41  */
42  app.get("/admin", [authJwt.verifyToken, authJwt.isAdmin], controller.adminBoard
43  );
44  // testaa USER käyttöoikeutta      You, 2 weeks ago • commit
45  app.get('/test/user', [authJwt.verifyToken], controller.userContent);
46
47  // testaa PM käyttöoikeutta
48  app.get('/test/pm', [authJwt.verifyToken, authJwt.isPmOrAdmin], controller.managementBoard);
49
50  // testaa ADMIN käyttöoikeutta
51  app.get('/test/admin', [authJwt.verifyToken, authJwt.isAdmin], controller.adminBoard);
52
53  // ***** KÄYTTÄJÄT
54  // näytä kaikki käyttäjät
55  app.get("/api/test/all", usercontroller.getAll);
56

```

Kuva 46. Ylläpito.routes.js URI-päätepisteitä ja toimintoja.

Tiedosto verifySignUp.js

Routes-kansioon lisätty verifySignUp.js tiedosto tarkistaa uuteen rekisteröitymiseen liittyvien tietojen oikeellisuuden. Se sisältää funktion, joka käsittelee kirjautumisyrityksessä onko kyseinen käyttäjänimi tai sähköposti jo olemassa (kuva 47) ja toinen funktio, jonka toistorakenne koodi hoitaa roolin olemassa olon tarkistuksen käsittelyn, kun muuttuja käy läpi halutut arvot for-silmukassa (kuva 48).

```

JS etusivu.js M JS verifySignUp.js M X
REST_oppari_authorization_authentication > src > routes > JS verifySignUp.js > ...
You, 1 second ago | 1 author (You)
1 const db = require('../config/db.config.js');
2 const config = require('../config/config.js');
3 const ROLES = config.ROLES;
4 const User = db.user;
5 const Role = db.role;
6
7
8 // Kun ollaan rekisteröimässä käyttäjää, tarkistetaan onko duplicaatti tietoja
9 checkDuplicateUserNameOrEmail = (req, res, next) => {
10   // -> onko Username(käyttäjänimi) käytössä
11   User.findOne({
12     where: {
13       username: req.body.username
14     }
15   }).then(user => {
16     if(user){
17       res.status(400).send("Virhe! Username jo käytössä");
18       return;
19     }
20
21     // -> onko email käytössä
22     User.findOne({
23       where: {
24         email: req.body.email
25       }
26     }).then(user => {
27       if(user){
28         res.status(400).send("Virhe! Email jo käytössä");
29         return;
30       }
31
32       next();
33     });
34   });
35 }

```

Kuva 47. VerifySignUp.js tiedoston funktion checkDuplicateUserNameOrEmail tarkistaa mahdolliset duplicaatti käyttäjätunnukset tai sähköpostit.

```

// onko ADMIN, PM, USER rooli KÄYTTÄJÄLLÄ olemassa, vai joku muu virhe syöte
checkRolesExisted = (req, res, next) => {
  for(let i=0; i<req.body.roles.length; i++){
    if(!ROLES.includes(req.body.roles[i].toUpperCase())){
      res.status(400).send("Virhe! Roolia ei ole olemassa = " + req.body.roles[i]);
      return;
    }
  }
  next();
}

You, last month • commit message ...
const signUpVerify = {};
signUpVerify.checkDuplicateUserNameOrEmail = checkDuplicateUserNameOrEmail;
signUpVerify.checkRolesExisted = checkRolesExisted;
module.exports = signUpVerify;

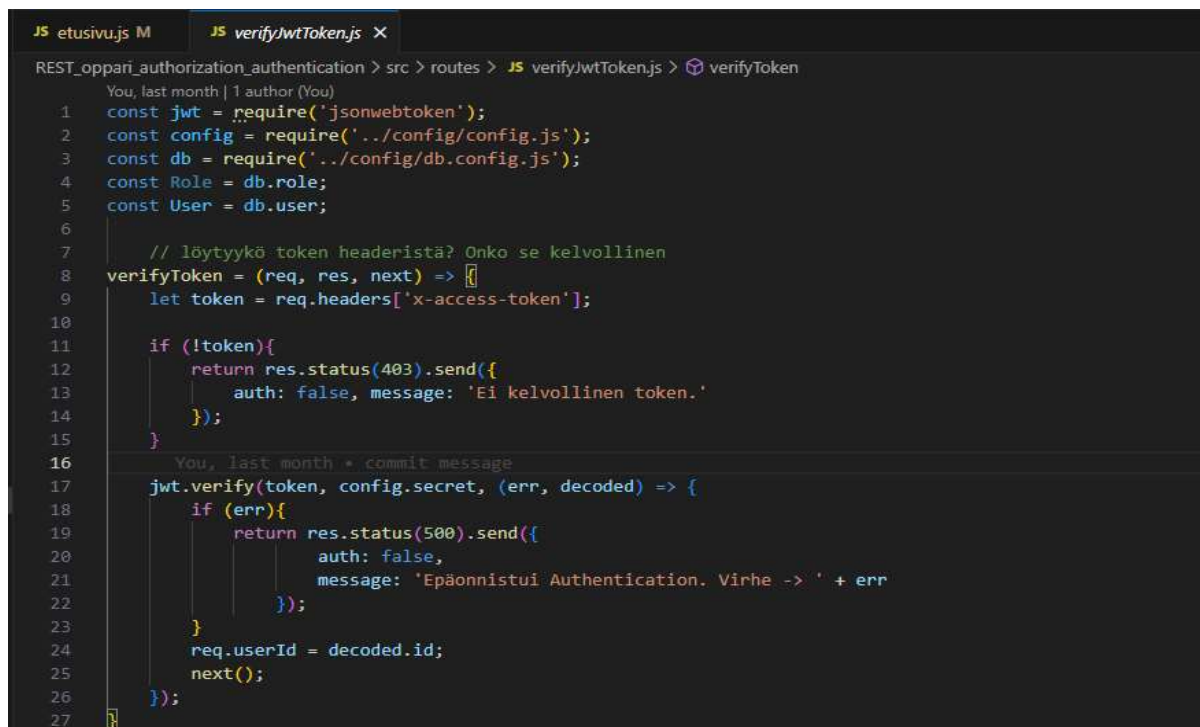
```

Kuva 48. VerifySignUp.js tiedoston checkRolesExisted funktion tarkistaa onko olemassa edes roolia, jolla käyttäjä yrittää rekisteröityä.

Tiedosto verifyJwtToken.js

Routes-kansiossa sijaitseva verifyJwtToken.js tiedosto on JWT-tokenin ja roolitason tarkastusta varten. Tiedoston Javascript koodi sisältää verifyToken-

funktion, joka tulee olemaan hyvin oleellinen varmentamisessa, koska se kertoo onko JWT-token, joka vaaditaan pääsyssä resursseihin, huomioiden vielä tietysti lisäksi vaadittava roolitaso (kuva 49).



```

JS etusivu.js M JS verifyJwtToken.js X
REST_oppari_authorization_authentication > src > routes > JS verifyJwtToken.js > verifyToken
You, last month | 1 author (You)
1  const jwt = require('jsonwebtoken');
2  const config = require('../config/config.js');
3  const db = require('../config/db.config.js');
4  const Role = db.role;
5  const User = db.user;
6
7  // löytyykö token headeristä? Onko se kelvollinen
8  verifyToken = (req, res, next) => {
9    let token = req.headers['x-access-token'];
10
11    if (!token){
12      return res.status(403).send({
13        auth: false, message: 'Ei kelvollinen token.'
14      });
15    }
16    You, last month * commit message
17    jwt.verify(token, config.secret, (err, decoded) => {
18      if (err){
19        return res.status(500).send({
20          auth: false,
21          message: 'Epäonnistui Authentication. Virhe -> ' + err
22        });
23      }
24      req.userId = decoded.id;
25      next();
26    });
27  }

```

Kuva 49. VerifyJwtToken.js ja tiedoston ensimmäinen funktio verifyToken.

Roolitus määrittää vielä erikseen onko missäkin URI-päätepisteessä minkä tason vaadittava pääsyoikeus, jokainen resurssi ei ole saatavilla alimman tason käyttöoikeuksilla, mutta on taas korkeimmilla roolivaltuuksilla.

Seuraavaksi tiedostossa on isAdmin-funktio (kuva 50), jota voidaan ylläpitoreititys tiedostossa URI-päätepisteessä käyttää funktiona, jolloin sovellus vaatii meiltä ADMIN-tason käyttöoikeutta halutessa käyttäjän päästä tuohon kyseiseen resurssiin käsiksi. Funktio tarkistaa käyttäjällä onko kelvollinen ja voimassa oleva JWT-token ja minkä tason rooli kyseisellä kirjautuneella käyttäjällä on.

```

// testaa ADMIN käyttöoikeus löytyykö
isAdmin = (req, res, next) => {
  let token = req.headers['x-access-token'];

  // etsitään käyttäjä by id
  User.findById(req.userId)
    .then(user => {
      // tarkistetaan käyttäjän rooli
      user.getRoles().then(roles => {
        for(let i=0; i<roles.length; i++){
          console.log(roles[i].name);
          // onko admin vai ei?
          if(roles[i].name.toUpperCase() === "ADMIN"){
            next();
            return;
          }
        }
        // jos ei ole ADMIN, kerrotaan että ADMIN rooli vaaditaan
        res.status(403).send("Vaaditaan ADMIN rooli!");
        return;
      })
    })
}

```

Kuva 50. VerifyJwtToken.js ja isAdmin funktio.

Tiedostoon on lisätty myös funktio tarkistamaan, onko käyttäjällä ADMIN- tai PM-tason rooli (kuva 51). PMlla on järjestelmässä monesti vähän heikommät oikeudet kuin ADMIN-tasolla, jolla taas on kaikki mahdolliset valtuudet järjestelmän käyttöön.

```

51 // testaa ADMIN/PM käyttöoikeus löytyykö, JOMPI KUMPI vaaditaan
52 isPmOrAdmin = (req, res, next) => {
53   let token = req.headers['x-access-token'];
54
55   User.findById(req.userId)
56     .then(user => {
57       user.getRoles().then(roles => {
58         for(let i=0; i<roles.length; i++){
59           if(roles[i].name.toUpperCase() === "PM"){
60             next();
61             return;
62           }
63
64           if(roles[i].name.toUpperCase() === "ADMIN"){
65             next();
66             return;
67           }
68         }
69
70         res.status(403).send("Require PM or Admin Roles!");
71       })
72     })
73 }
74
75 const authJwt = {};
76 authJwt.verifyToken = verifyToken;
77 authJwt.isAdmin = isAdmin;
78 authJwt.isPmOrAdmin = isPmOrAdmin;
79
80 module.exports = authJwt;

```

Kuva 51. VerifyJwtToken.js ja isPmOrAdmin funktio.

10.1.3 Models-kansio

Tiedosto role.models.js

Tiedoston koodi määrittää moduulin, joka vie Role-nimisen funktion, joka on malli Roles-tietokantataulukolle (kuva 52). Roolimallilla on kaksi attribuuttia: ID (kokonaisluku, primary key) ja name (merkkijono). The sequelize.define-metodia käytetään määrittämään uusi Rooli-malli.



```
JS etusivu.js M JS role.model.js X
REST_oppari_authorization_authentication > src > models > JS role.model.js > ...
You, last month | 1 author (You)
1 module.exports = (sequelize, Sequelize) => {
2   const Role = sequelize.define("roles", {
3     // määritetään rooli
4     id: {
5       type: Sequelize.INTEGER,
6       primaryKey: true,
7     },
8     name: {
9       type: Sequelize.STRING,
10    },
11  });
12
13  return Role;
14 };
15
```

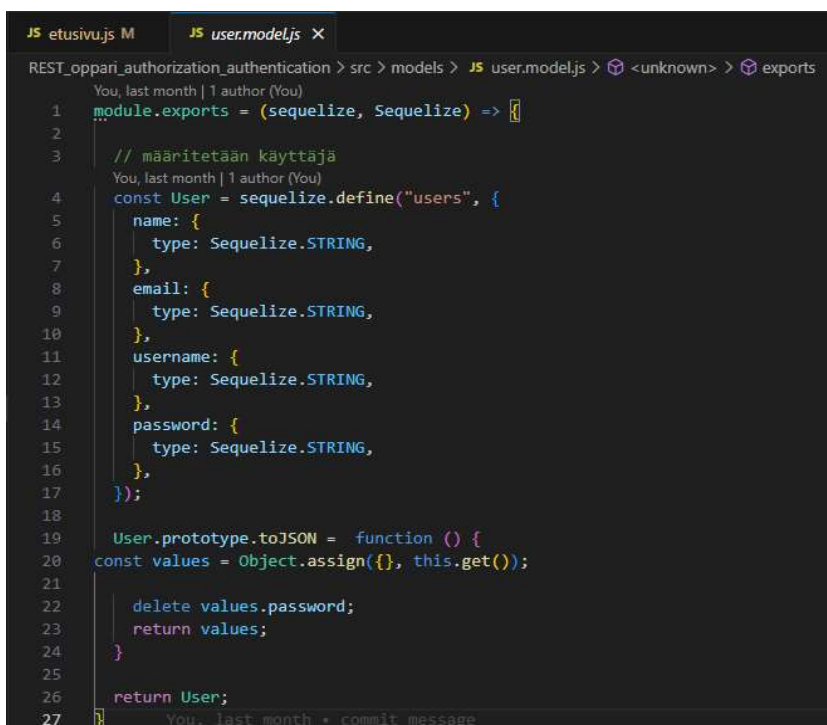
Kuva 52. Role.model.js

Tiedosto user.model.js

Yllä olevassa tiedostossa määritettiin Role-malli ja nyt seuraavaksi tullaan käyttäjä-mallin luomista varten tekemään oma tiedostonsa (kuva 53).

Sequelize.define-menetelmää käytetään uuden Role-malli tekemiseen ja se sisältää merkkijono attribuutit: name, email, username ja password. Jokainen attribuutti edustaa objektia tietokantataulukon sarakkeesta.

User.prototype.toJSON-menetelmä määritetään sitten mukauttamaan käyttäjämallin JSON-esitystä, huomioiden että salasana kuuluu arkaluontoisiin tietoihin, jotka poistetaan itsestään esityksestä ennen JSON-tietojen näyttämistä.



```

1  module.exports = (sequelize, Sequelize) => {
2
3    // määritetään käyttäjä
4    const User = sequelize.define("users", {
5      name: {
6        type: Sequelize.STRING,
7      },
8      email: {
9        type: Sequelize.STRING,
10     },
11     username: {
12       type: Sequelize.STRING,
13     },
14     password: {
15       type: Sequelize.STRING,
16     },
17   });
18
19   User.prototype.toJSON = function () {
20     const values = Object.assign({}, this.get());
21
22     delete values.password;
23     return values;
24   }
25
26   return User;
27 }

```

Kuva 53. User.model.js tiedosto.

10.1.4 Controllers-kansio

Tiedosto user.controller.js

Tiedosto user.controller.js, joka hakee kaikki käyttäjät tietokannasta ja niihin liittyvät roolit eli hoitaa logiikan mitä tarvitaan näiden toimintojen suorittamiseen. Se käyttää on asynkroninen funktiota getAll, joka käyttää taas await-toimintoa tietojen käsittelyyn (kuva 54). Se hakee ensin kaikki käyttäjät tietokannasta käyttämällä User.findAll() funktiota. Sitten se hakee kunkin käyttäjän roolit ja rakentaa uuden objektin, joka sisältää käyttäjän tiedot rooleineen.



```

JS etusivu.js M JS user.controller.js X
REST_oppari_authorization_authentication > src > controllers > JS user.controller.js > getAll > getA
You, 4 weeks ago | 1 author (You)
1 const db = require('../config/db.config.js');
2 const config = require("../config/config");
3 const User = db.user;
4
5 exports.getAll = async (req, res) => {
6   await User.findAll()
7     .then(async (users) => {
8     var allUsers = [];
9     for (let i = 0; i < users.length; i++) {
10      var user = users[i];
11      await User.findOne({
12        where: {
13          id: user.id,
14        },
15      }).then(async (user) => {
16        var authorities = [];
17        user.getRoles().then((roles) => {
18          for (let y = 0; y < roles.length; y++) {
19            authorities.push("ROLE_" + roles[y].name.toUpperCase());
20          }
21        });
22        let userR = {
23          id: user.id,
24          username: user.username,
25          name: user.name,
26          email: user.email,
27          roles: authorities,
28          createdAt: user.createdAt,
29          updatedAt: user.updatedAt,
30        };
31        allUsers.push(userR);
32      });
33    }
34    res.status(200).send(allUsers);
35  })
36  .catch((err) => {
37    res.status(500).send({ message: err.message });
38  });
39 }

```

Kuva 54. User.controller.js tiedosto.

Tiedosto controller.js

Controller.js tiedoston tärkeisiin toimintoihin kuuluu käyttäjän rekisteröitymisen ja kirjautumisen logiikan hoitaminen (kuva 55). Rajapinnalle lähetetään viestipyynnön runko-osassa rekisteröitymisen attribuuttien tiedot, jolla käyttäjä halutaan rekisteröidä: name, username, email, password ja rooli. Tietojen ollessa kelvolliset rekisteröityminen hyväksytään ja sen jälkeen tiedot lisätään tietokantaa, paitsi käytetään bcryptia käyttäjän salasanan hajauttamiseen ennen sen tallentamista tietokantaan.

```

JS etusivu.js M JS controller.js X
REST_oppari_authorization_authentication > src > controllers > JS controller.js > ...
You, 4 weeks ago | 1 author (You)
1 const db = require('../config/db.config.js');
2 const config = require('../config/config.js');
3 const User = db.user;
4 const Role = db.role;
5
6 const Op = db.Sequelize.Op;
7
8 const jwt = require('jsonwebtoken');
9 const bcrypt = require('bcryptjs');
10
11 // REKISTERÖIDY
12 exports.signup = (req, res) => {
13
14   console.log("Rekisteröityminen käynnissä.");
15
16   // req.body tiedot käyttäjästä
17   User.create({
18     name: req.body.name,
19     username: req.body.username,
20     email: req.body.email,
21     // salasana kryptataan tietokantaan
22     password: bcrypt.hashSync(req.body.password, 8)
23   }).then(user => {
24     Role.findAll({
25       where: {
26         name: {
27           [Op.or]: req.body.roles
28         }
29       }
30     }).then(roles => {
31       user.setRoles(roles).then(() => {
32         let message = "Käyttäjä nimeltä: " + req.body.name + " on rekisteröity onnistuneesti!"
33         res.send(message);
34       });
35     }).catch(err => {
36       res.status(500).send("Virhe -> " + err);
37     });
38   }).catch(err => {
39     // ...
40   });
41 }

```

active issue Sign in to Bitbucket 0 0 0 0 You, last month Ln 8, Col 37 Tab Size: 4 UTF-8 CRLF

Kuva 55. controlles.js ja singup eli rekisteröityminen sovelluksen käyttäjäksi.

Rekisteröitymisen jälkeen tiedostoon lisätään mahdollisuus kirjautua itse rajapintasovellukseen (kuva 56). Onnistunut kirjautuminen palauttaa JWT-Tokenin viestivastauksessa takaisin käyttäjälle. Koodiin on lisätty virheilmoituksia, mahdollisen väärän salasanan tai käyttäjätunnuksen varalta, jolloin selain ei palauta JWT-tokenia, eikä järjestelmää pääse käyttämään.


```

JS etusivu.js M JS controller.js X
REST_oppari_authorization_authentication > src > controllers > JS controller.js > signup > signup > then0 c
81 exports.signin = (req, res) => {
82   User.findOne({
83     where: {
84       username: req.body.username,
85     },
86   })
87   .then((user) => {
88     if (!user) {
89       return res.status(404).send({ message: "Wrong username or password!" });
90     }
91
92     var passwordIsValid = bcrypt.compareSync(
93       req.body.password,
94       user.password
95     );
96
97     if (!passwordIsValid) {
98       return res.status(401).send({
99         accessToken: null,
100         message: "Wrong username or password!",
101       });
102     }
103
104     var token = jwt.sign({ id: user.id }, config.secret, {
105       expiresIn: 86400,
106     });
107
108     var authorities = [];
109     user.getRoles().then((roles) => {
110       for (let i = 0; i < roles.length; i++) {
111         authorities.push("ROLE_" + roles[i].name.toUpperCase());
112       }
113       res.status(200).send({
114         id: user.id,
115         username: user.username,
116         roles: authorities,
117         accessToken: token,
118       });
119     });
120   });

```

Kuva 56. Controller.js signin eli sisäänkirjautuminen.

Tiedoston lopussa on lisätty koodia, joka mahdollistaa roolitasojen testausta niin että rajapinta palauttaa selkeän viestivastauksen näkyville (kuva 57).

```

126
127 // voit testata toimiiko USER käyttötason oikeus
128 exports.userContent = (req, res) => {
129   res.status(200).send(">>> User sisältö!");
130 }
131
132 // voit testata toimiiko ADMIN käyttötason oikeus
133 exports.adminBoard = (req, res) => {
134   res.status(200).send(">>> Admin sisältö!");
135 }
136
137 // voit testata toimiiko PM käyttötason oikeus
138 exports.managementBoard = (req, res) => {
139   res.status(200).send(">>> PM sisältö!");
140 }

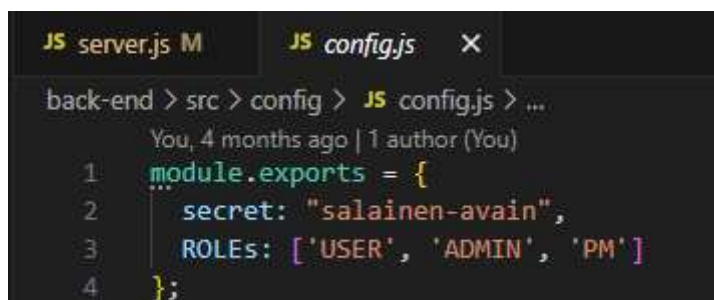
```

Kuva 57. Roolitasojen testaukseen liittyvät koodi lisäykset.

10.1.5 Config-kansio

Tiedosto config.js

Pieni tiedosto joka sisältää moduulin, josta löytyy rajapinnan salainen avain. Secret: "salainen-avain": tässä kohtaa koodia määritetään salaisen avaimen arvoksi "salainen-avain" (kuva 58). Avain-objektia käytetään sovelluksessa JWT-tokenin luomisen varmistamiseen.



```
back-end > src > config > JS config.js > ...  
You, 4 months ago | 1 author (You)  
1 module.exports = {  
2   secret: "salainen-avain",  
3   ROLES: ['USER', 'ADMIN', 'PM']  
4 };
```

Kuva 58. Salainen avain.

Tiedosto db.config.js

Db.config.js-tiedosto määrittää Sequelize-tietokantayhteyden ja määrittää tietokantamallit käyttäjille ja rooleille. Se luo myös useista-moneen-suhteen käyttäjien ja roolien välille.


```

REST_oppari_authorization_authentication > src > config > JS db.config.js > ...
You, 3 months ago | 1 author (You)
1  const database = require('./.env')
2
3  const Sequelize = require("sequelize");
4
5  const sequelize = new Sequelize(database.DB, database.USER, database.PASSWORD, {
6    host: database.HOST,
7    dialect: database.dialect,
8    operatorAliases: false,
9
10   pool: {
11     max: database.max,
12     min: database.pool.min,
13     acquire: database.pool.acquire,
14     idle: database.pool.idle
15   }
16 });
17
18 const db = {};
19
20 db.Sequelize = Sequelize;
21 db.sequelize = sequelize;
22
23 db.user = require("../models/user.model.js")(sequelize, Sequelize);
24 db.role = require("../models/role.model.js")(sequelize, Sequelize);
25
26 db.role.belongsToMany(db.user, { through: 'user_roles', foreignKey: 'roleId', otherKey: 'userId' });
27 db.user.belongsToMany(db.role, { through: 'user_roles', foreignKey: 'userId', otherKey: 'roleId' });
28
29 db.ROLES = ["user", "admin", "moderator"];
30
31 module.exports = db;
You, 4 months ago • commit message

```

Kuva 59. Db.config.js tiedoston sisältö.

11 Testaus: valtuuttaminen ja todentaminen

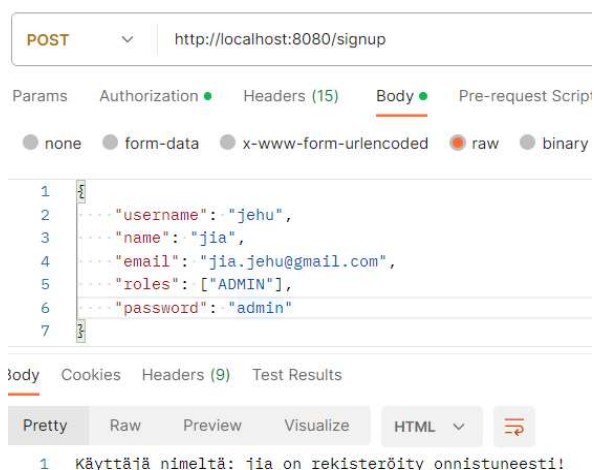
Luodaan rajapinnan todentamisen ja valtuuttamiseen testaamiseen seuraavat testit:

- rekisteröi uusi käyttäjä
- kirjaudu järjestelmään ja palauta käyttäjälle JWT-token
- ADMIN-roolitaso
- USER-roolitaso
- PM-roolitaso

11.1.1 Testi: Rekisteröi uusi käyttäjä

Tehdään POST-kutsu osoitteeseen <https://localhost:8080/signup>, viestipyyntöön rungon mukana lähetetään oikein olevat uuden käyttäjän rekisteröitymiseen liittyvät attribuutit: username, name, email, roles ja password (kuva 60).

Tapahtumasta palautuu statuskoodi 200 OK.



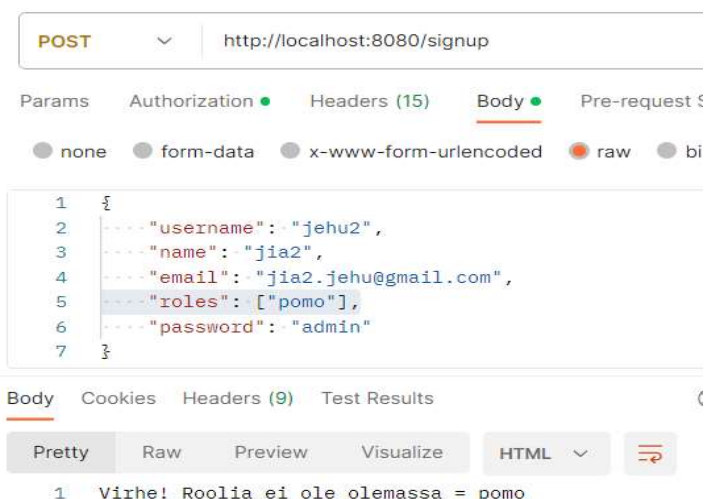
Kuva 60. Onnistunut POST-viestipyyntö.

Postman ilmoittaa tapahtuman onnistuneen, mutta tarkistetaan vielä HeidiSQL:stä, joka näyttää uuden käyttäjän tietueen tulleen tietokantaan onnistuneesti (kuva 61). Salasana muuttuu myös onnistuneesti tietokantaan lisäysvaiheessa kryptatuksi.

id	name	email	username	password	createdAt	updatedAt
3	jia	jia.jehu@gmail.com	jehu	\$2a\$08\$KAXYxuzGUg/B.czz1...	2023-09-15 07:20:19	2023-09-15 07:20:19
1	nina	nina@gmail.fi	admin1	\$2a\$08\$AqikCUCRgVuIth3b2...	2023-08-11 09:31:59	2023-08-11 09:31:59

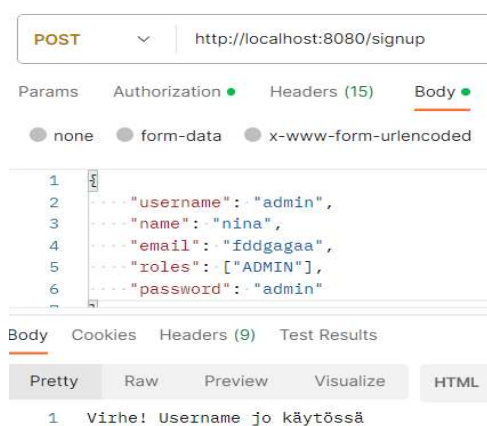
Kuva 61. HeidiSQL näkyy uusi lisätty käyttäjä.

Testataan virheellisen rooli tiedon syöttämistä seuraavaksi, yritetään lisätä käyttäjä roolilla "pomo", jota ei ole olemassa. `verifySignUp.js` tiedostossa `checkRolesExisted` funktio siis toimii halutusti ja ilmoittaa ettei kyseistä roolia ole olemassa ja näin ollen uuden käyttäjän lisäys epäonnistuu, mikä olikin haluttu tulos testissä (kuva 62). Statuskoodi joka palautuu on 400 Bad Request, mikä on oikein hyvä rajapinnan vastaus tässä tilanteessa.



Kuva 62. Käyttäjä syöttää virheellisen roolitasoa rekisteröitymisvaiheessa ja tapahtumasta palautuu virheilmoitus.

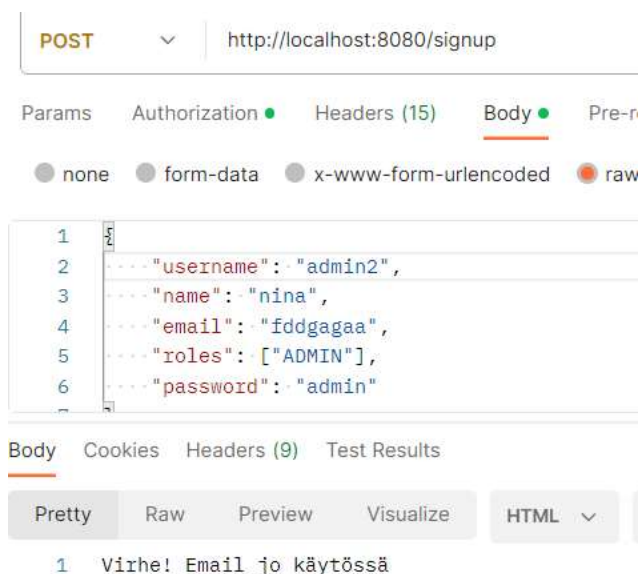
Yritetään seuraavaksi rekisteröidä käyttäjä, jonka username on jo olemassa (kuva 63). Tiedoston verifySignUp.js funktio checkDuplicateUserNameOrEmail antaa oikein virheilmoituksen ja ei rekisteröi uutta käyttäjää. Testitulos on siis mitä haluttiinkin ohjelman vasteen tilanteessa olevan, statuskoodi joka palautui oli 400 Bad Request eli huono viestipyyntö ja tämäkin ilmoitus on aivan oikein.



Kuva 63. Rajapinta ei hyväksy duplikaatti username-tietoja tietokantaan rekisteröitäväksi ja rekisteröityminen epäonnistuu.

Muutetaan username sellaiseksi, jota ei löydy järjestelmästä, mutta pidetään email-kenttä samana. Rajapinta antaa myös nyt virheilmoituksen ja rekisteröinti epäonnistuu, koska tietokannasta löytyy jo kyseisellä sähköpostilla oleva

käyttäjä (kuva 64). Tämä viestipyyntö palauttaa myös statuskoodin 400 Bad Request eli kelvoton viestipyyntö.

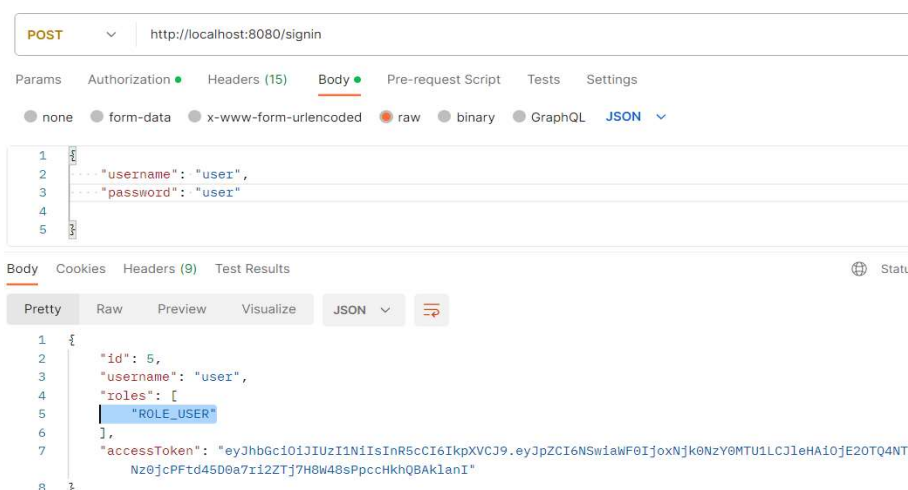


Kuva 64. Duplikaatti sähköpostin rekisteröiminen ei onnistu rajapintaan.

Jos yritetään rekisteröidä uusi käyttäjä, jonka joku kenttä jää tyhjäksi tulee vastaus Could't response ja rajapinta vaatii uudelleen käynnistämistä. Statuskoodia ei edes palaudu vaan suora Error ilmoitus. Havaitaan testissä, että on korjattava virhe rajapinnan koodissa.

11.1.2 Testi: Roolin luonti

Luodaan käyttäjä, jolla on USER-roolin käyttöoikeustaso ja tarkastellaan HeidiSQL:ssä tarkemmin, onko käyttäjän rooli oikein asetettu. Postman ilmoittaa onnistumisesta ja antaa statuskoodin 200 OK (kuva 65).



Kuva 65. Käyttäjä luotu roolilla: USER.

Postman näyttää myös käyttäjän lisäyksen onnistuneen ja käyttäjä sai Id-tunnuksen: 5. user_roles taulu kertoo mikä on käyttäjän Id-tunnus ja mikä sen käyttäjän roolin Id-tunnus taas on (kuva 66). Äskettäin luotu käyttäjä, joka oli omalla henkilökohtaisella id-tunnuksella 5. näyttää taulun mukaan omistavan rooli id-tunnuksen 1.

createdAt	updatedAt	roleId	userId
2023-09-15 07:48:39	2023-09-15 07:48:39	1	5

Kuva 66. User_roles kertoo mikä rooli on milläkin käyttäjällä.

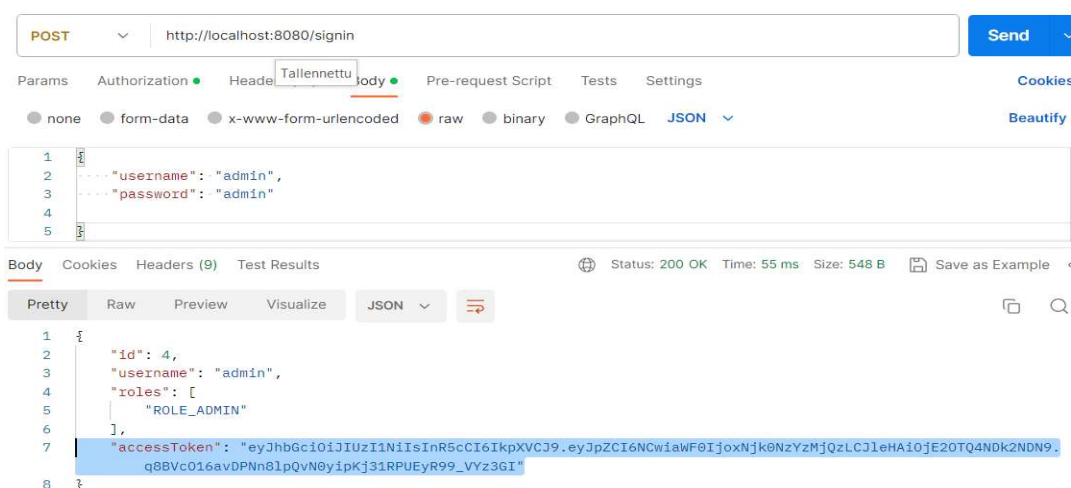
Seuraavaksi tarkastellaan roles-taulua, kertoo meille mikä id-tunnus, on milläkin roolilla (kuva 67). user_roles-taulusta selvisi, että käyttäjällä jota tarkastellaan oli id-tunnus roolina 1, joka roles-taulu, kertoman mukaan onkin USER-roolin. Testi tarkastelu näyttää rekisteröitymisvaiheessa roolin asettuvan oikein tietokantaan.

id	name	createdAt	updatedAt
1	USER	2023-08-11 09:24:11	2023-08-11 09:24:11
2	ADMIN	2023-08-11 09:24:11	2023-08-11 09:24:11
3	PM	2023-08-11 09:24:11	2023-08-11 09:24:11

Kuva 67. Roles-taulu näyttää eri roolit, joita tietokannassa on.

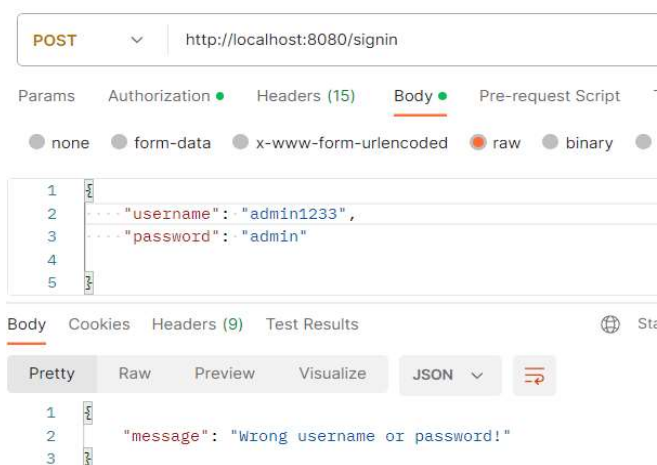
11.1.3 Testi: sisään kirjautuminen

Kirjannuttaan järjestelmään salasanalla ja käyttäjätunnukselle, jotka on luotu aiemmin onnistuneessa testissä 7.6.1: rekisteröi uusi käyttäjä. Kirjautuminen tehdään POST-viestipyynnöllä osoitteeseen `http://localhost:8080/signin` sisältäen viestirungossa username ja password-kentät. Kirjautumisyrityksestä palautuu statuskoodi 200 OK ja tapahtumasta palautuu viestivastauksena accessToken, joka on JWT-token, jota tulemme tarvitsemaan resursseihin käsiksi pääsyyn kyseisellä käyttäjällä (kuva 68).



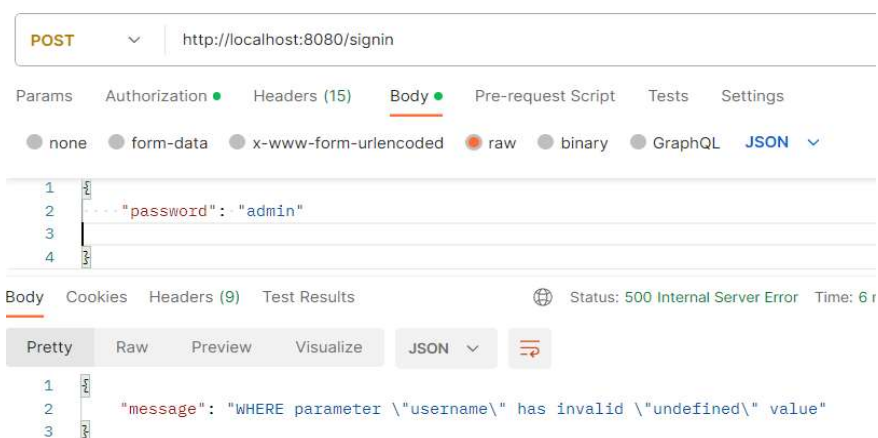
Kuva 68. Viestipyynnön vastaus sisältää accessTokenin, kun kirjautuminen on tehty onnistuneesti.

Kirjautumaan ei pääse virheellisillä tiedoilla, yritetään syöttää väärä käyttäjätunnus ja salasana viestipyynnön rungon tietoina (kuva 69). Rajapinta palauttaa virhesanoman käyttäjätunnus tai salasana on väärä ja antaa statuskoodin 404 Not Found eli ei löydy.



Kuva 69. Virheellinen salasana tai käyttäjätunnus ei salli kirjaamista.

Kirjautumisyrityksessä, jossa jätetään salasana kenttä tai käyttäjätunnus kokonaan antamatta viestipyyntöä ei oikeuta kirjautumaan (kuva 70). Puuttuva käyttäjätunnus tai salasana antaa statuskoodin 500 Internal Server Error, joka kertoo, että rajapinta kohtaa odottamattoman tilanteen, joka estää sitä täyttämästä pyyntöä. Sopivampi statuskoodi olisi kyllä tilanteessa esimerkiksi 401 Unauthorized.

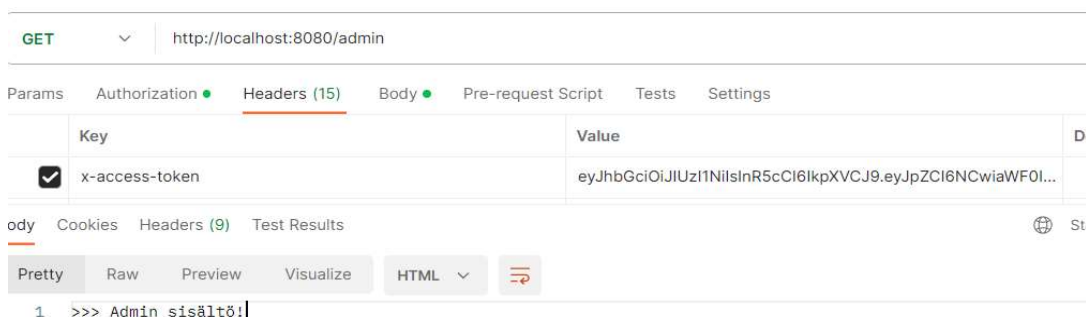


Kuva 70. Puuttuva salasana tai käyttäjätunnus ei oikeuta kirjautumaan.

11.1.4 Testi: ADMIN-käyttöoikeus

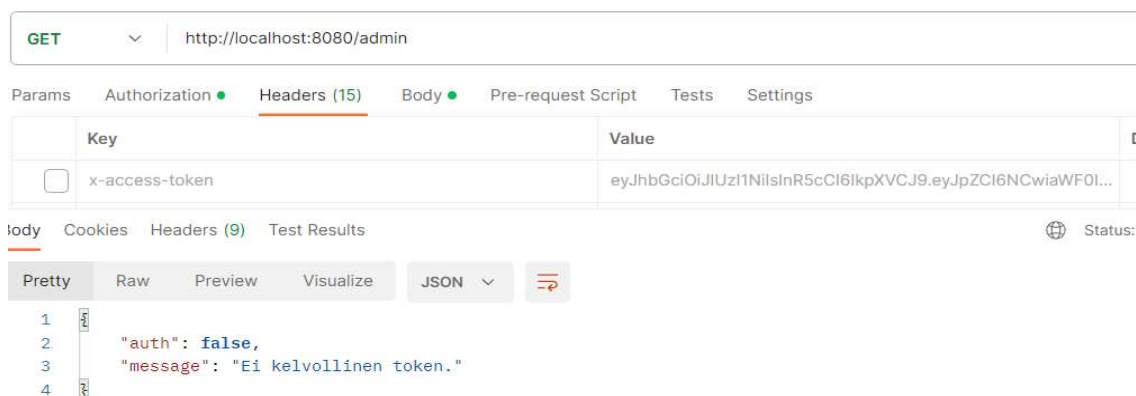
Testataan rajapinnan ADMIN-roolin vaativan `http://localhost:8080/admin` resurssin pitävyyttä yrittämällä päästä käsiksi tuohon URI-päätepisteeseen kirjautuneena käyttäjänä, jolla on ADMIN-rooli. Resurssiin lähetetään GET-

pyyntö, jonka viestipyynnön otsikossa on JWT-token, rungossa ei lähetetä mitään. ADMIN-sisältö tulee näkyviin ja statuskoodi palautuu 200 OK, testi onnistuu halutuvin tuloksin (kuva 71).



Kuva 71. Admin-käyttöoikeuden vaativa resurssi näkyy ADMIN-tason käyttäjälle.

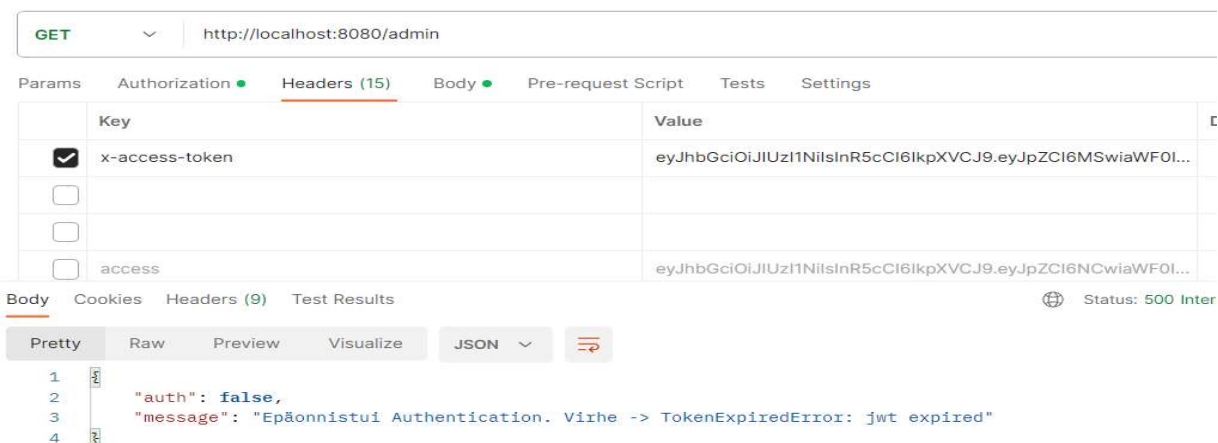
Yritetään saada ADMIN-tason sisältö esille, ilman että käytetään viestipyynnössä JWT-tokenia. Pysytään edelleen kirjautuneena ADMIN-tason käyttäjänä, mutta lähetetään sama viestipyyntö, mutta nyt checkbox ei ole valittuna viestipyynnön otsikon kohdassa x-access-token, jolloin JWT-tokenia ei lähetetä ja viestipyyntöön tulee vastaus virheilmoitus "ei kelvollinen Token" ja statuskoodi 403 Forbidden eli pääsyeväty, joten testi onnistuu (kuva 72).



Kuva 72. Viestipyyntö ilman tokenia ei oikeuta pääsyä resurssiin.

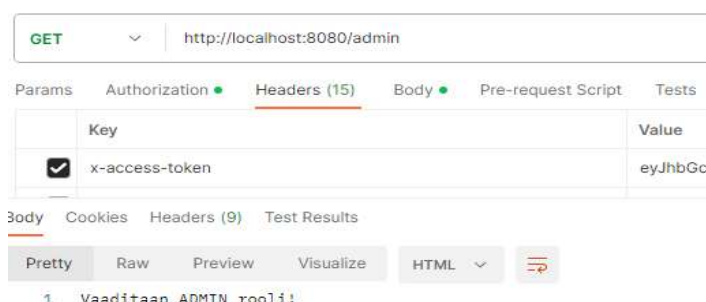
Yritetään seuraavaksi saada resurssi auki keksityllä JWT-tokenilla. Lähetetään viestipyynnön otsikossa x-access-token kohdassa aivan satunnaisena keksitty merkkijono ja rajapinta antaa viestivastauksena statuskoodin 500 Internal Server Error eli rajapinta kohtaa odottamattoman tilanteen, joka estää sitä

täyttämästä pyyntöä (kuva 73). Tilanteessa sopivampi statuskoodi olisi esimerkiksi 401 Unauthorized.



Kuva 73. Virheellinen token estää pääsyn resurssiin.

Testataan ADMIN-tason vaatimaan resurssiin pääsyä vielä lopuksi USER-tason käyttöoikeuksilla (kuva 74). Kirjaudutaan ulos järjestelmästä ja valitaan sisäänkirjautumiseen USER-tason käyttäjä. Otetaan onnistuneesta kirjautumisesta palautunut JWT-token talteen ja yritetään päästä käsiksi ADMIN-tason resurssiin. GET-viestipyyntö palauttaa vastaukseksi pääsyeväty ja vaaditaan ADMIN-tason rooli eli rajapinta toimii niin kuin pitääkin, antaa vielä statuskoodin 403 Forbidden eli pääsy evätty.

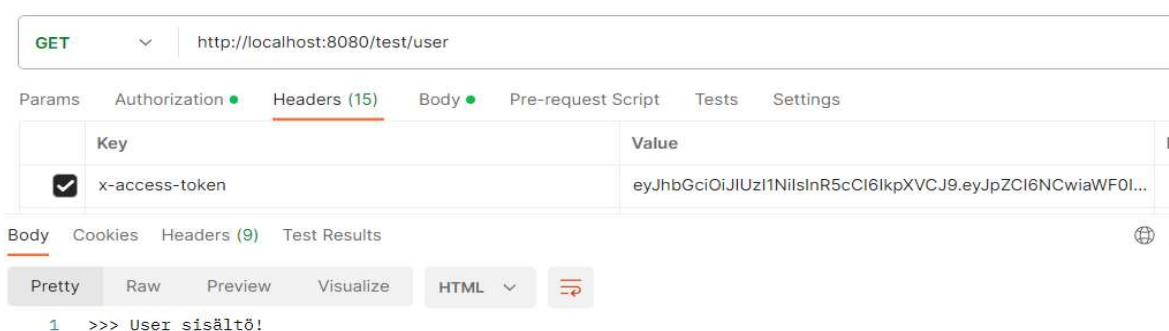


Kuva 74. USER-tason käyttöoikeuksilla ei päästä käsiksi ADMIN-tason vaatimaan resurssiin.

11.1.5 Testi: USER-sisältö

Rajapinnassa ollaan kirjautuneena USER-tason käyttäjänä ja tehdään GET-viestipyyntö, jonka viesti otsikossa oikea JWT-token osoitteeseen <http://localhost:8080/test/user>. Viestivastauksena palautuu statuskoodi 200 OK ja näkyviin tulee USER-käyttöoikeuden vaativa resurssin sisältö eli testi onnistuu halutusti.

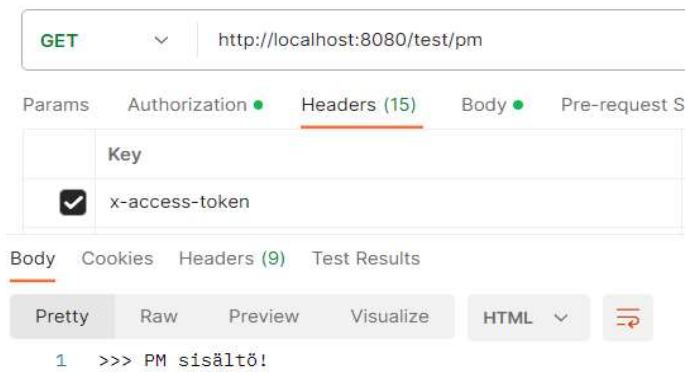
USER-käyttötaso on saatavilla kaikilla käyttöoikeus tasoilla, kunhan käyttäjä on kirjautunut järjestelmään ja antaa kelvollisen JWT-tokenin (kuva 75). Sama testi onnistuu myös samoin tuloksin, kun vaihdettiin käyttäjäksi ADMIN- ja PM-tason käyttäjä.



Kuva 75. USER-tason sisältö näkyy user-, pm- ja admin-käyttäjille, mutta ei rajapinnan käyttäjille, jotka eivät ole kirjautuneena ja anna viestipyyntönsä otsikossa JWT-tokenia.

11.1.6 Testi: PM-sisältö

Lähetetään kirjautuneena ADMIN-tason käyttäjänä GET-pyyntö, jonka viestiotsikossa lähetetään kelvollinen JWT-token URI-päätepiisteeseen <http://localhost:8080/test/pm>. Sisältö tulee näkyviin statuskoodin 200 OK kanssa niin kuin kuuluikin, koska ADMIN-roolitaso oli suurempi valtuuksiltaan kuin PM (kuva 76). Testi onnistuu myös samoin odotetuin tuloksin PM-tason käyttäjällä.



Kuva 76. Pm-roolitason sisältö tulee näkyviin admin ja pm-tason oikeuksilla.

USER-käyttötason oikeuksilla sama testi antaa myös halutun tuloksen eli palauttaa statuskoodin 403 Forbidden eli pääsy evätty ja antaa virheilmoituksen vaaditaan PM tai ADMIN-tason rooli (kuva 77).



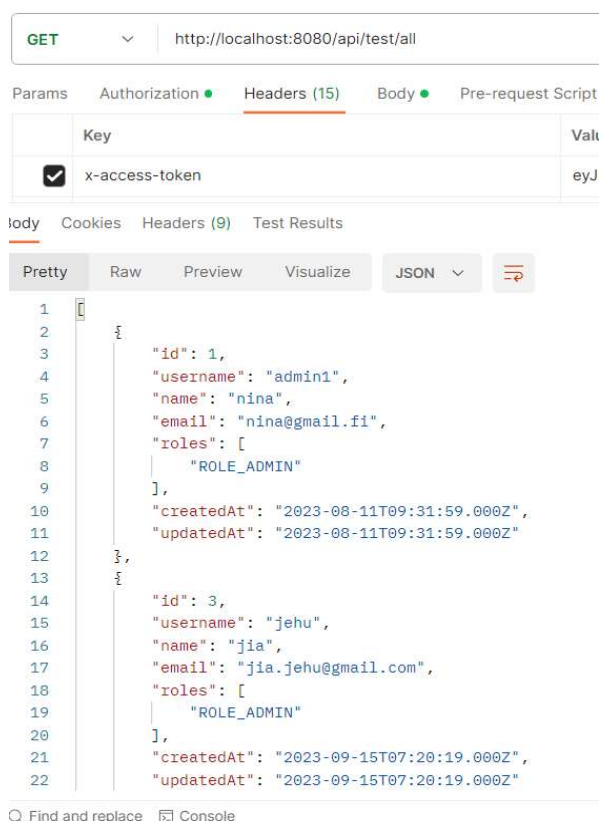
Kuva 77. User-käyttöoikeus ei riitä resurssiin.

11.1.7 Testi: Näytä kaikki käyttäjät

Tehdään GET-viestipyyntö osoitteeseen <http://localhost:8080/api/test/all>, joka palauttaa onnistuneesti näkyville kaikki käyttäjät tietokannasta ja antaa statuskoodin 200 OK (kuva 78). Salasanat eivät näy kuitenkaan käyttäjien

listassa, kuten ohjelman koodissa oli määriteltykin ja tämä toteutuu testissä myös onnistuneesti.

Tämä resurssi on koodissa määritelty niin, että ei vaadita kirjautumista, eikä minkään tason roolia.



Kuva 78. Haetaan näkyville kaikki järjestelmään rekisteröityneet käyttäjät.

12 Koirat-resurssin ja valtuuttamisen ja todentamisen yhdistäminen

Rajapinta-sovellukseen on saatu tähän mennessä toteutettua CRUD-toimintoinen koirat-resurssi ja käyttäjän valtuuttaminen ja todentaminen, seuraavaksi yhdistetään näiden toiminnot eli halutaan koirat-resurssi ei olisi enää vapaasti saatavilla, vaan se vaatisi käyttäjän valtuuttamisen ja todentamisen tietyllä roolitasolla.

Koirat.routes.js tiedostossa määriteltyyn GET /koirat/ resurssiin, joka tuo siis näkyville kaikki koirat, halutaan nyt lisätä ADMIN-roolituksen vaatimus. Tuodaan ensin tiedoston käyttöön moduuli koodilla `const authJwt = require('./verifyJwtToken.js');`, jonka jälkeen voidaan lisätä URI-päätepiisteeseen `isAdmin` ja `verifyToken` funktiot, jotka muuttavat resurssin ADMIN-roolitasoa vaativaksi (kuva 79).

```

4  module.exports = function(app) {
15  const koiratController = require('../controllers/koirat.controller');
16
17  // You, 3 months ago • commit
18  // näytä kaikki koirat
19  // app.get('/koirat/', koiratController.findAll);
20  app.get('/koirat/', [authJwt.verifyToken, authJwt.isAdmin], koiratController.findAll);
21

```

Kuva 79. Koirat.routes.js tiedoston muutokset ADMIN-käyttötason roolia varten.

Yritetään päästä kyseiseen resurssiin ilman kirjautumista seuraavaksi lähettämällä pelkkä GET-pyyntö ilman mitään viestin runko tai otsikkotietoja osoitteeseen <http://localhost:8080/koirat>. Vastaus on haluttu 403 Forbidden eli pääsy evätty (kuva 80).

```

1  @baseUrl = http://localhost:8080/
2
3  ### Aloitus
4  Send Request
5  GET {{baseUrl}}
6  ###*****koira****
7  ### Hae koirat
8  Send Request
9  GET {{baseUrl}}koirat
10
11 ### Hae koira by id
12 Send Request
13 GET {{baseUrl}}koirat/56
14
15 ### Lisää koira

```

```

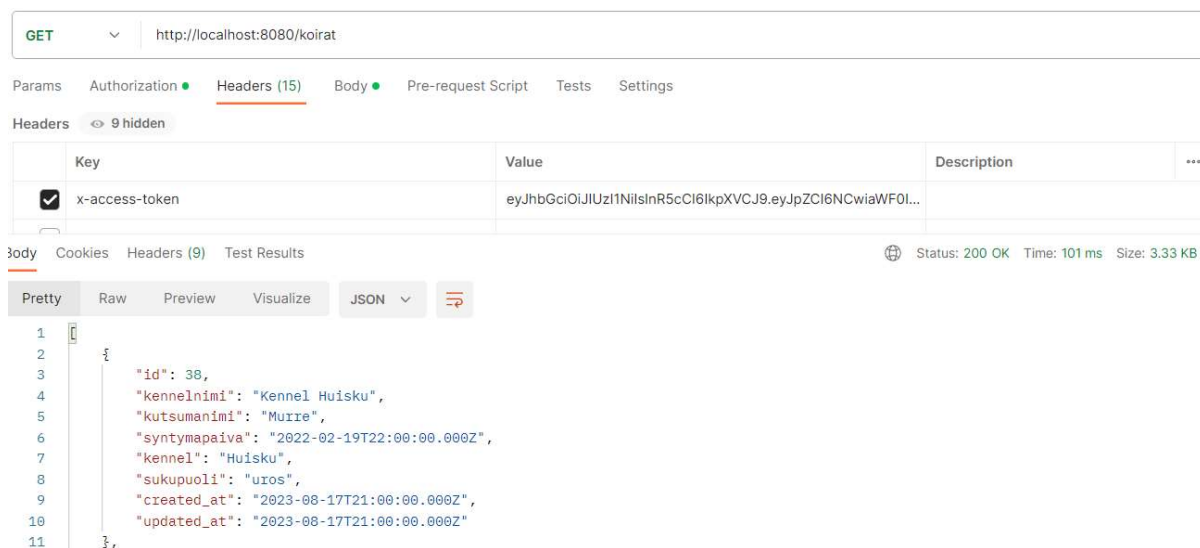
1  HTTP/1.1 403 Forbidden
2  X-Powered-By: Express
3  Access-Control-Allow-Origin: *
4  Access-Control-Allow-Headers: x-access-token, On
5  Content-Type: application/json; charset=utf-8
6  Content-Length: 48
7  ETag: W/"30-itB0lQcsfLXChI+koxwBYlVFh0"
8  Date: Fri, 01 Dec 2023 07:35:45 GMT
9  Connection: close
10
11 {
12   "auth": false,
13   "message": "Ei kelvollinen token."
14 }

```

Kuva 80. Viestipyynnön vastaus 403 Forbidden.

Seuraavaksi kirjaututaan käyttäjällä, jolla on ADMIN-tason oikeudet ja otetaan onnistuneen kirjautumisen jälkeen viestivastauksena palautuva JWT-token talteen. Tehdään uudestaan GET-pyyntö samaan <http://localhost:8080/koirat> URI-päätepiisteeseen, mutta nyt lähetetään kirjautuneena käyttäjänä JWT-token

viestipyynnön otsikossa myös rajapinnalle. Vastauksena palautuu kaikkien koirien tiedot eli ohjelman vaste oli juuri se mitä haluttiin (kuva 81).



Kuva 81. Rajapinta palauttaa resurssin näkyville onnistuneiden valtuutus ja todennustietojen jälkeen.

13 Pohdinta

Koodaamaan oppii vaan altistamalla itseänsä koodaukselle ja opinnäytetyössä pääsin syventymään juuri omaan suuntautumiseeni ja näin harjaannuttamaan osaamistani back-endin saralla. NODEJS on vain yksi monista back-endin ohjelmointikehyksistä, yhden tavan ymmärtäminen, näin edes auttaa jatkossa muitakin back-end ohjelmistokehityksiä oppimaan nopeammin. Koodin työstäminen ja läpikäyminen kirjallisesti auttoi kehittämään osaamista web-ohjelmoijana.

Hyvin järjestelmällisenä ihmisenä pidän selkeästi nimetyistä omiin kansioihin ja tiedostoihin jäsenennyistä koodista controllers-, models- ja routes-rakenteella toteutettuna. Varmasti aion jatkossakin noudattaa näitä opittuja tapoja. Ohjelma rakennettuna helposti tulkittavasti ja runsaasti kommentoituna helpottaa ohjelmoijan työtä ja sen huomaa ihan konkreettisesti sovelluksen varsinkin kasvaessa mitä suuremmaksi.

Opinnäytetyö vahvisti ajatusta, että olen oikean suuntautumisen valinnut koodarina ja tykkään työskennellä itsenäisesti erityisesti pitkäjänteisyyttä vaativissa projekteissa, jotka haastavat välillä ohjelmanratkaisukykyä.

Opinnäytetyöstä jouduttiin poistamaan paljon materiaalia, ettei sen koko kasvaisi laajuudeltaan liian suureksi, joten tulevaisuudessa varmasti paremmin osaisin hahmottaa minkä laajuiseksi tulisi määrittää vastaavia työprojekteja.

13.1 Sovelluksen kehitys mahdollisuudet

Kyseessä oli perus rajapintasovellus ja sitä olisi mahdollisuuksia laajentaa full-stack sovellukseksi yhteensopivan asiakaspuolen kanssa. Rajapintaa on myös laajennettavissa helposti monimuotoisemmaksi sen models-, routes- ja controller-rakenteen ansiosta, jossa on moduulit eroteltu selkeästi.

Sovelluksen koodia voisi vielä työstää paremmaksi, vaikka kokonaisuutena se toimii. Joissain tilanteessa rajapinta saisi antaa vielä selkeämmät vastaukset niin konsoliin kuin front-endille pyyntövastauksena. Sovelluksessa tulisi korjata mahdollisia bugeja ja back-endin antamat statuskoodit tulisi vielä joissakin kohdissa olla enemmän osuvampia.

NODEJSn kirjastot päivittyvät koko ajan ja sovellusta tulisi pitää huoltaa, jotta se pysyy ajan tasalla ja toimivana uudempien tulevien versioidenkin kanssa. Sovelluksen ajan tasalla pysyminen on turvallisuuden kannalta tärkeää ja tietoturvan saralla varmasti sovelluksessa löytyy vielä kehitettävää, kuten IDt voisivat olla kryptattuina tietokannassa.

13.2 REST-periaatteiden toteutuminen

Tarkastellaan Richardsonin kypsyyssmallin avulla, toteutuuko sovelluskokeilun rajapinnassa mikä taso, aloittaen matalimman tason tarkastelusta eli nollasta.

Nollatason REST-rajapinnalta vaadittaisi, että sovellus ei käytä mitään resursseja tai HTTP-verbejä, eli tämä ei toteudu. Tason yksi rajapinta käyttää useita resursseja, mutta vain yhtä HTTP-verbiä eli tämä taso toteutuu vain osittain, koska käytössä on useampia resursseja, mutta ei yksi vaan useampia HTTP-verbejä käytössä.

Tason kolme rajapinnalta vaaditaan useita resursseja ja käytössä olevia HTTP-verbejä eli sovellus täyttää kokonaan kolmostason vaatimukset. Korkeimman eli neljännen tason REST-rajapinta käyttää HATEOS-ominaisuuksia, näitä ei sovellukseen ole lisätty eli sovellus jää REST-tasolla kolme.

LÄHTEET

- Matveinen, M. 2023. Kurssi API-Rajapinnat. Luento Karelia-ammattikorkeakoulussa. 11.12.2023
- Http.dev. 2023. HTTP Status Codes. <https://http.dev/status> 1.9.2023
- JeffTK. 2018. History of HTTPS Usage. <https://www.jefftk.com/p/history-of-https-usage> 1.9.2023
- Aloi.io 2023. The History Of APIs. <https://www.aloi.io/api/history/> 1.9.2023
- Valjas. 2023. Mitä integraatio, rajapinta ja api tarkoittavat? <https://valjas.fi/opi/blogi/mita-integraatio-rajapinta-ja-api-tarkoittavat/> 3.9.2023
- RESTFUL-API. 2023. REST API Tutorial. <https://restfulapi.net/richardson-maturity-model/> 3.9.2023
- INTERVIEWBIT. 2021. Difference Between Frontend and Backend – Frontend Vs Backend. <https://www.interviewbit.com/blog/difference-between-frontend-and-backend/> 3.9.2023
- GeeksForGeeks. 2023. Frontend vs Backend. <https://www.geeksforgeeks.org/frontend-vs-backend/> 3.9.2023
- Guru99. 2023. Frontend Developer vs Backend Developer: Key Differences. <https://www.guru99.com/front-end-vs-back-end-developers.html> 10.9.2023
- TRay.io. 2023. How do apis work? <https://tray.io/blog/how-do-apis-work> <https://www.youtube.com/watch?v=XBu54nfzxAQ> 10.9.2023
- Kinsta 2023. GraphQL vs REST: Everything You Need To Know <https://kinsta.com/blog/graphql-vs-rest/> 10.9.2023
- Guru99. 2023. SOAP vs REST API: Difference Between Web Services. <https://www.guru99.com/comparison-between-web-services.html> 10.9.2023
- Altexsoft. 2023. Comparing API Architectural Styles: SOAP vs REST vs GraphQL vs RPC. <https://www.altexsoft.com/blog/soap-vs-rest-vs-graphql-vs-rpc/> 10.9.2023
- Rest API Tutorial. 2021. What is Richardson Maturity Model. <https://restfulapi.net/richardson-maturity-model/> 12.9.2023
- Ics. 2000. Architectural Styles and the Design of Network-based Software Architectures. <https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm> 12.9.2023
- GraphQL 2023. Learn <https://graphql.org/> 12.9.2023
- GeeksForGeeks. 2023. REST API Architectural Constraints. <https://www.geeksforgeeks.org/rest-api-architectural-constraints/> 12.9.2023
- GeeksForGeeks.. 2023. Difference between SOAP and WSDL. <https://www.geeksforgeeks.org/difference-between-soap-and-wsdl/> 15.9.2023
- Kingwaysoft. 2023. Pros and Cons of SOAP and REST API Integrations. https://www.kingwaysoft.com/resources/industry-analysis/soap-rest-apis?campaignid=18819483825&adgroupid=148798141848&network=g&device=c&qclid=Cj0KCQjwmtGjBhDhARIsAEqfDEeTL-sbVlVicTueaFmLleTnSK5QsnFa3l-RcR0tzbMzaDro3X3lhuYaAtAkEALw_wcB 15.9.2023
- GeeksForGeeks. 2023 JSON web token. <https://www.geeksforgeeks.org/json-web-token-jwt/> 15.9.2023

GeeksForGeeks. 2023. Explain http-authentication.

<https://www.geeksforgeeks.org/explain-http-authentication/> 24.9.2023

Writer. 2023 Bearer Authentication <https://dev.writer.com/docs/authentication> 24.9.2023

DevOpsSchool. 2023. What is Bearer token and How it works?

<https://www.devopsschool.com/blog/what-is-bearer-token-and-how-it-works/>

24.9.2023

DevOpsSchool. 2023, Oauth 2.0:n esittely.

<https://www.devopsschool.com/blog/introduction-of-oauth-2-0/> 24.9.2023

Swagger. 2023. Basic Authentication.

<https://swagger.io/docs/specification/authentication/basic-authentication/>

24.9.2023

Atostek. 2023. Autentikointi ja auktorisointi mikropalveluarkkitehtuurissa.

<https://atostek.com/autentikointi-ja-auktorisointi-mikropalveluarkkitehtuurissa/>

24.9.2023

Strongdm. 2023 The Definitive Guide to Authentication.

<https://www.strongdm.com/authentication> 4.10.2023

Auth0 by Okta. 2023. OpenID Connect Protocol-

<https://auth0.com/docs/authenticate/protocols/openid-connect-protocol>

4.10.2023

