Tampereen ammattikorkeakoulu

REST-toteutus Node.js-kehitys-alustalla

Jani Sillanpää

OPINNÄYTETYÖ
Maaliskuu 2022

Tieto- ja viestintätekniikka
Ohjelmistotekniikka

TIIVISTELMÄ
Tampereen ammattikorkeakoulu
Tieto- ja viestintätekniikka
Ohjelmistotekniikka

SILLANPÄÄ, JANI
REST-toteutus Node.js-kehitysalustalla

Opinnäytetyö 37 sivua, joista liitteitä 0 sivua
Maaliskuu 2022

Opinnäytetyössä toteutettiin Node.js-kehitysalustalla rakennettu API opiskelijan aikaisemmin toteutettuun web-sovellukseen. Node.js-kehitysalustalle on olemassa useita erilaisia sovelluskehyksiä ja kirjastoja, mikä mahdollistaa hyvin erilaisia ratkaisuja ohjelmointirajapinnan ohjelmoimiseen ja opinnäytetyössä tutustutaan yhteen toteutustapaan.

API ohjelmoidaan Express-sovelluskehyksellä, joka on ollut pitkään suosituin sovelluskehys Node.js-kehitysalustalle. Tietokannaksi luodaan PostgreSQL-relaatiotietokanta, johon rakennetaan TypeORM-kirjastolla ORM-abstrahointikerros. Koko sovellus on toteutettu TypeScript-ohjelmointikielellä.

Rajapinnasta pyritään tekemään mahdollisimman skaalautuva, ylläpidettävä ja pyritään jakamaan moduuleihin. Rajapinnassa käytetään controller-service-repository -arkkitehtuuria.

Asiasanat: API, Node.js, ORM, REST, tietokanta

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
ICT Engineering
Software Engineering

Sillanpää, Jani
REST implementation with Node.js

Bachelor's thesis 37 pages, appendices 0 pages
March 2022

The object of this thesis was to implement API with Node.js development platform to student's previously made web application. There are various frameworks and libraries for Node.js platform which allow API to be implemented in different ways.

In this thesis API is programmed with Express framework which has been the most popular framework for a long time. PostgreSQL relational database will be created as the database of API which will include ORM abstraction layer with TypeORM library. The entire application has been programmed with TypeScript.

The main goal is to make the interface as scaled as possible and easy to maintain as well as divided into modules. The interface is programmed with Controller-Service-Repository pattern.

Key words: API, database, Node.js, ORM, REST

SISÄLLYS

LYHENTEET JA TERMIT

Annotaatio	Metadatan muoto ohjelmoinnissa, millä voidaan joskus muokata suoritettavan ohjelman toiminnallisuutta
API	Application Programming Interface, ohjelmointirajapinta
Entity	Ohjelmointiluokka, jolla ORM-sovelluskehyksissä määritellään tietokantataulu
Entity Framework	ORM-sovelluskehys C#-ohjelmointikielelle
Hibernate	ORM-sovelluskehys Java-ohjelmointikielelle
http	Hypertext Transfer Protocol, hypertekstin siirtoprotolla
I/O	Input/Output, tiedonsiirto komponenttien välillä
JavaScript	Web-ohjelmoinnissa usein käytetty ohjelmointikieli
JSON	JavaScript Object Notation, merkintäkieli, jota käytetään muun muassa tiedonvälitykseen
JWT	JSON Web Token, avoimen standardin menetelmä käyttöoikeustietueiden hallinnoimiseen ohjelmistojen välillä
ORM	Object-relational mapping, ohjelmointitekniikka tietokantakyselyjen toteuttamiseen olio-ohjelmoinnilla kyselykielen sijaan
REST	Representational State Transfer, arkkitehtuurityyli
SQL	Structured Query Language, kyselykieli tietokantahakujen tekemiseen relaatiotietokannoissa
TypeScript	Ohjelmointikieli, laajennos JavaScriptiin, jota käytetään usein web-ohjelmoinnissa
YAML	YAML Ain’t Markup Language, merkintäkieli, jota käytetään usein konfiguraatiotiedostoissa ja tiedonvälityksessä
XML	Extensible Markup Language, merkintäkieli, jota käytetään muun muassa tiedonvälityksessä

1 JOHDANTO

Opinnäytetyö käsittelee REST API -rajapinnan toteuttamista Node.js-kehitysalustalle. Työssä tehtiin kaikki tietokannan kanssa tapahtuva interaktio ORM-tekniikalla. API toteutetaan käyttämällä Express-sovelluskehystä, joka on ollut pitkään suosituin sovelluskehys Node.js-kehitysalustalle.

REST API -rajapinta ja tietokanta toteutettiin opiskelijan aikaisemmin luomaan web-sovellukseen, joka on toteutettu React-kirjastolla. Web-sovellus on keskustelufoorumi, johon voi rekisteröidä käyttäjän. Käyttäjät lähettävät viestejä keskusteluaiheisiin eri keskustelualueilla.

Opinnäytetyössä käsitellään REST-arkkitehtuuria ja mitä se pitää sisällään. Siinä tutustutaan tietokantoihin ja yksinkertaisen autentikoinnin toteuttamiseen palvelimella, jonka jälkeen esitellään keskeisimmät käytetyt teknologiat, kirjastot, ohjelmointikielet. Työssä toteutetaan eräänlainen API-arkkitehtuuri, jossa API jaetaan kolmeen tasoon: controller, service ja repository.

Teknologioiden pohjustamisen jälkeen pureudutaan tarkemmin toteutettuun APIin, jossa tarkastellaan tarkemmin, kuinka Express-sovelluskehyksellä on saatu implementoitua edellä mainittu arkkitehtuuri.

Loppupohdinnassa tuodaan esille projektin aikana mieleen tulleita mietteitä Node.js-kehitysalustasta, sen ympärille rakentuneesta ekosysteemistä ja Express-sovelluskehyksestä ja millaisiin toimintaympäristöihin nämä voisivat soveltua.

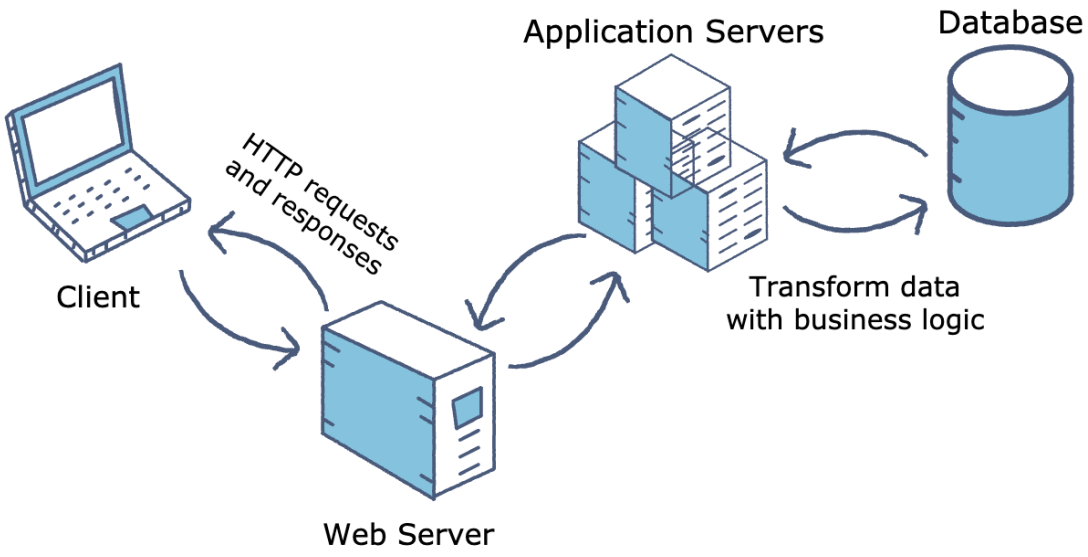
2 ARKKITEHTUURI

Ohjelmistot ovat usein hyvin laajoja, joiden kanssa voi työskennellä suuri määrä kehittäjiä ja ohjelmistot ovat liian monimutkaisia yhden ihmisen ymmärrettäväksi. Ohjelmistoissa voi olla useita kymmeniäkin vuosia pitkä elinkaari, jolloin kehittäjät ja ylläpitäjät vaihtuvat. Tämän vuoksi ohjelmiston dokumentointi ja arkkitehtuuri ovat tärkeitä osa-alueita ohjelmistokehittämisessä pitkällä aikavälillä.

Ohjelmistoarkkitehtuurin ohjelmiston yleisrakenne, jonka on tarkoitus palvella ohjelmistoa. Arkkitehtuuri koostuu joukosta valintoja ja päätöksiä, joilla on tarkoitus tehdä ohjelmistosta ymmärrettävä, ylläpidettävä, laajennettava ja skaalautuva. Usein arkkitehtuuri jaetaan erilaisiin kerroksiin ja moduuleihin, joiden välisiä riippuvuuksia hallinnoidaan erilaisin rajapinnoin. (Helsingin yliopisto 2009.)

2.1 REST-arkkitehtuurimalli

REST (Representational State Transfer) on ohjelmointirajapintoihin tarkoitettu arkkitehtuurimalli. Arkkitehtuurimallissa eritellään selain ja palvelin itsenäisiksi kokonaisuuksiksi, joita kehitetään erillään toisistaan. Palvelin ja selain eivät jaa tilaa tai kontekstia toisillensa.



KUVIO 1. Yksinkertaistettu kuvaus RESTistä (Larimer 2020).

Kuvio 1 havainnoi yksinkertaistettua REST-arkkitehtuuria. Selain pyytää http-kutsuilla esimerkiksi haluamaansa resurssia, jonka palvelin hakee tietokannasta ja palauttaa selaimelle. Palvelimella voi olla myös toimintalogiikkaa, jota voidaan suorittaa http-pyynnöllä. (REST API Tutorial n.d.a.) RESTille on olemassa muitakin ohjaavia periaatteita, joista opinnäytetyön kannalta tärkeitä ovat tilattomuus, välimuisti ja resurssit.

Tilattomuus

Tilattomuus palvelimen näkökulmasta tarkoittaa, että se ei säilö tai päättele selaimen tilaa. Jokaisessa selaimen lähettämässä kutsussa pitää olla kaikki tarvittava informaatio, mikä vaaditaan kutsun suorittamiseen. Tämä tarkoittaa myös mahdolliseen autentikointiin tai autorisointiin liittyvän tiedon säilömistä. (REST API Tutorial, n.d.b.)

Välimuisti

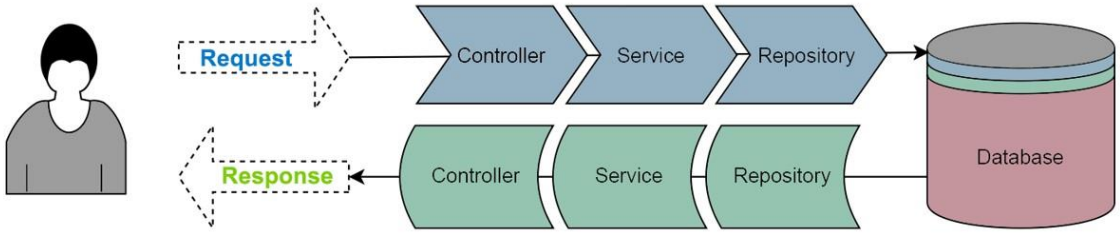
Välimuistin hyödyntämisellä pyritään keventämään kuormitusta palvelimella. Selaimen saadessaan vastaus tämä tallennetaan sen välimuistiin. Uuden kutsun tehdessä tarkistetaan selaimesta, että löytyykö vastaus jo välimuistista, ennen kuin kuormitetaan palvelinta. (Wilson n.d.)

Resurssit

Resurssit ovat selaimen ja palvelimen käsittelemiä tietoja, osoitteita ja käsitteitä. Resursseja voi olla esimerkiksi tietokantatieto, staattinen HTML-sivusto tai näitä yksilöivät osoitteet. Resurssi voi olla myös kokoelma tietoja.
Nykyään yleisimmät tiedostomuodot välittää resursseja palvelimelta selaimelle ovat JSON, YAML, XML, HTML. (Jansen 2012.)

2.2 APIn arkkitehtuuritasot

APIn jakaminen itsenäisiin tasoihin auttaa koodin jäsentelyssä, helpottaa ylläpidettävyyttä, refaktorointia ja jatkokehittämistä. Yksi yleinen tapa jäsentää arkkitehtuuria on jakaa se controller-service-repository -tasoihin; joskus samaan viitataan input-logic-data access -tasoilla. (Malav 2018.)



KUVIO 2. Kutsun ja vastauksen kulku arkkitehtuurissa (Manning 2022).

Eri tasot mahdollistavat muodostaa funktioita, joita on mahdollista testata helposti. Esimerkiksi kun controller-tasolle jää kaikki toiminnallisuus http-kutsuun liittyen, voidaan service- ja repository-tasoa testata helposti yksikkötesteillä (Cleary 2022.)

Controller-taso

Controller-taso on vastuussa vain kutsujen vastaanottamisesta ja lähettämisestä. Tällä tasolla ei ylläpidetä logiikkaa, mikä käsittelee resursseja. Controller-tasolta kutsutaan service-tasoa, jonka tehtävä on tuoda resursseja ja/tai tehdä kutsun vaatimaa logiikkaa. (Malav 2018.) Controller-taso toimii rajapintana palvelimen toimintalogiikalle, jota käytetään http-kutsuilla.

Service-taso

Service-tason vastuu on pitää kaikki kutsun vaatima logiikka itsessään. Service-tason komponentti ei välitä mikä tai mistä sitä kutsutaan ja sitä voi kutsua myös toinen service-tason komponentti. Service-tasossa ei suorita suoriteta hakuja tietokantaan, vaan tietokantojen kanssa tapahtuva interaktio kapseloidaan repository-tasolle. (Bechtol 2020.)

Repository-taso

Repository-tason sisään kapseloidaan kaikki tietokannan kanssa tapahtuva interaktio. Repository-tason komponentit käsittelevät service-tasolta saamiaan tietoja ja suorittavat tarvittavat haut tietokantaan. Repository-taso ikään kuin tarjoaa tarvittavat hakukyselymetodit service-tasolle. (Malav 2018.)

2.3 Tietokannat

Tietokannat sisältävät kaiken säilytettävän tiedon. Koska tietoa voi olla todella paljon, täytyy tietokantaa varten olla tehokas tietokannanhallintajärjestelmä, joka mahdollistaa tehokkaan tiedon hakemisen ja muokkaamisen (KeyCDN 2017). Tietokannanhallintajärjestelmät voidaan jakaa kahteen ryhmään: relaatiotietokantoihin (SQL) tai dokumenttitietokantoihin (NoSQL). Suurimmat erot näillä kahdella ovat tapa rakentaa tietokanta ja kyselykieli.

Relaatiotietokannoissa käytetään SQL-kyselykieltä, joka mahdollistaa tehdä monipuolisia ja monimutkaisia kyselyitä kompleksisesta tietokannasta. Relaatiotietokannan huonona puolena voidaan pitää sitä, että koko tietokanta täytyy rakentaa tietyllä tavalla, jotta SQL-kyselykielellä saa eheää dataa ja tietokannassa ei esiinny samaa tietoa useampaan kertaan.

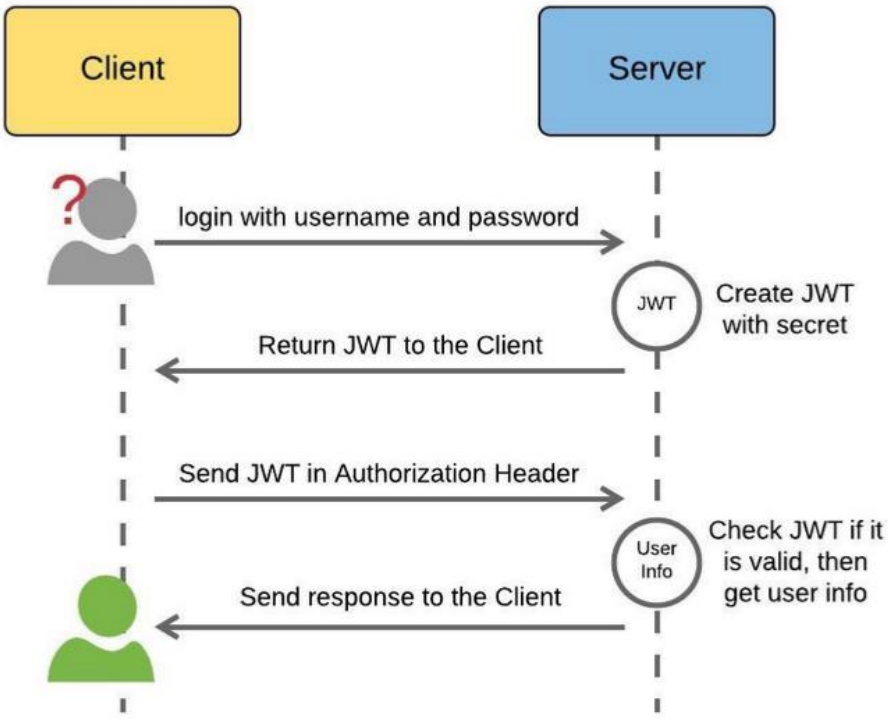
Dokumenttitietokannassa ei tarvitse noudattaa tiettyä struktuuria sitä rakentaessa, vaan tietokannan osat, dokumentit, voi itse määritellä ja niitä voi muokata myöhemmin. Dokumenttien välisten suhteiden luominen ja ylläpitäminen on itse päätettävissä.

Yleensä dokumenttitietokantaa rakentaessa pitääkin valita aikaisin haluttu tapa rakentaa tietokantaa, koska tietokantarakenteen muuttaminen jälkikäteen on vaikeata ja monimutkaista. Dokumenttikannoissa ei voi käyttää SQL-kyselykieltä, vaan kyselykieli riippuu käytettävästä tietokannan hallintajärjestelmästä. (Smallcombe 2021.)

2.4 Autentikointi

Autentikointi toteutetaan usein JWT:n (JSON Web Token) avulla, joka on määritelty avoimella standardilla RFC 7519. JWT on merkkijono, joka on tiiviste kolmen osan kokonaisuudesta: otsikko (header), sisältö (payload) ja salainen osa (secret). Tiiviste voidaan luoda usealla eri algoritmilla, esimerkiksi HMAC SHA-256 -algoritmilla. (JWT n.d.)

Koska REST on tilaton, täytyy käyttäjän tiedot lähettää selaimelle esimerkiksi kirjautuessa, jotta jokaisen kutsun yhteydessä käyttäjä voi liittää kutsuunsa tarvittavat tiedot autentikoimiseen. Yleisiä tapoja on tallentaa JWT selaimen paikalliseen muistiin tai evästeisiin. Paikalliseen muistiin tallentamisen heikkous on käyttäjän pääsy generoituun JWT:hen suorittamalla JavaScript-koodia, mikä mahdollistaa XSS-hyökkäykset. Evästeillä käyttäjän pääsy voidaan estää ohjelmallisesti käyttämällä vain ”httpOnly”- ja ”secure” -evästeitä. Evästeiden heikkoutena on pieni koko, niihin on mahdollista tallentaa vain 4 kb. (Wirantono 2020.)



KUVIO 3. JWT:n generointi ja käyttäminen (Reşit, H. 2019).

JWT liitetään http-kutsuun omaksi http-otsikoksi, jolloin palvelin purkaa tokenin ja tarkastaa käyttäjän tietojen vastaavan sitä, mitä resurssin käsittely vaatii.

3 KÄYTETYT TEKNOLOGIAT JA KIRJASTOT

Node.js:n ympärille on rakentunut suuri ekosysteemi, johon on paljon saatavilla erilaisia kirjastoja ja sovelluskehyksiä. Paketteja hallinnoidaan usein npm-pakkaustenhallintatyökalulla ja npm:ään voi julkaista kuka tahansa omia kirjastoja muiden käytettäviksi. Palvelinohjelmoinnin lisäksi Node.js ja npm ovat merkittävässä roolissa web-sivustojen ja mobiilisovelluksien ohjelmoinnissa.

3.1 Node.js-kehitysalusta

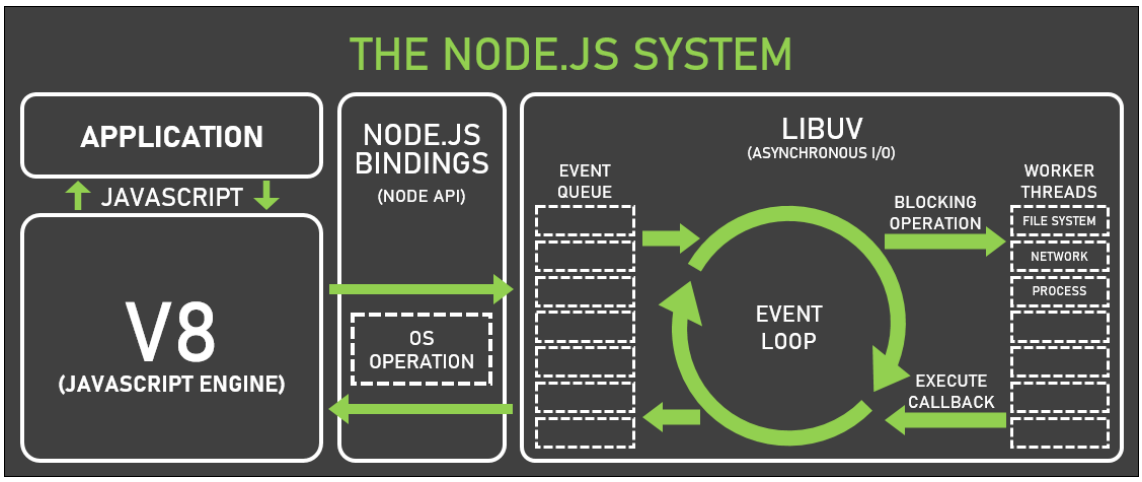
Node.js on vapaan lähdekoodin alustariippumaton kehitysalusta, joka mahdollistaa JavaScript-koodin suorittamisen selaimen ulkopuolella. Se suorittaa JavaScript-koodia samalla V8-moottorilla, jota käytetään Google Chrome -selaimessa. (OpenJS Foundation n.d.a.)

Node.js:ssä on monia etuja, jotka selittävät sen suosiota nykyään. Yksi etu on alustariippumattomuus, joka mahdollistaa kehittämisen monissa käyttöjärjestelmäympäristöissä. Toinen merkittävä etu on, että sitä voi ohjelmoida JavaScript-ohjelmointikielellä, joka on hyvin yleinen ohjelmointikieli web-ohjelmoinnissa. Tämä mahdollistaa myös ohjelmointikielten käyttämisen, mitkä käännetään JavaScriptiksi, kuten esimerkiksi TypeScript. (Mozilla 2022a.)

3.1.1 Toiminnallisuus

Suurin osa palvelinratkaisuista, kuten Spring, ASP.NET, käyttävät useita säikeitä toiminnassaan. Tällöin jokaiselle kutsulle luodaan uusi säie. Node.js eroaa huomattavasti tässä suhteessa käsitellessään jokaista kutsua samassa säikeessä.

On hieman harhaanjohtavaa sanoa Node.js:ää vain yksisäikeiseksi prosessiksi, koska todellisuudessa vain kutsujen käsittely toimii yhdessä säikeessä. Node.js:n standardikirjastot mahdollistavat I/O-toimintojen suorittamisen asynkronisesti omissa säikeissään (OpenJS Foundation n.d.a).



KUVIO 4. Event loop -malli Node.js:ssä (Tutoriallandexample, 2022).

Node.js:n keskiössä on yksisäikeinen ”event loop”, eli tapahtumasilmukka, joka vastaanottaa ja käsittelee kaikki tapahtumat. Uuden kutsun tullessa tapahtumasilmukka ottaa sen heti käsittelyyn ollessaan vapaa. Tapahtumasilmukka osoittaa kutsulle oman prosessin, mitä kutsu alkaa suorittaan itsenäisesti ja tapahtumasilmukka voi tällöin käsitellä muita kutsuja. Kun prosessi valmistuu, kutsuu se tapahtumasilmukkaa, joka käsittelee sen uudelleen heti ehdittyään. (OpenJS n.d.b)

3.1.2 npm-pakkaustenhallinta

npm on komentorivillä toimiva pakkaustenhallintatyökalu ja verkossa sijaitseva tietovarasto julkaistuille paketeille. Verkossa julkaistuja paketteja voi asentaa helposti npm:llä komennolla ”npm install (paketin nimi)” tai poistaa komennolla ”npm uninstall (paketin nimi)”. (OpenJS Foundation n.d.b.) Pakkausten hallintaan npm:lle vaihtoehtoinen työkalu on Facebookin kehittämä Yarn.

Node.js-projekteissa on package.json-niminen tiedosto, joka määrittelee projektiin asennetut paketit ja muita projektiin liittyviä tietoja, kuten esimerkiksi mahdolliset skriptit, nimi, kuvaus, versionumero ja vastaavia metatietoja. package.json päivittyy samalla kun npm:llä tai Yarnilla asennetaan, poistetaan tai päivitetään paketti.

3.2 TypeScript-ohjelmointikieli

TypeScript on Microsoftin kehittämä ja ylläpitämä ohjelmointikieli. Se on laajennus JavaScriptiin ja tuo mukanaan staattisen tyypityksen, luokkapohjaiset objektit ja olio-ohjelmoinnin piirteitä. Koska TypeScript käännetään JavaScriptiksi, on sillä mahdollista kirjoittaa uudemmalla ES6-syntaksilla ja kääntäessä valita minkä version JavaScript-koodia generoidaan. (GeeksforGeeks 2021.)

```
1  const multiplyNumberByTwo = (num: number): number => num * 2;
2
3  let randomNumber: number;
4
5  // Type 'string' is not assignable to type 'number'.
6  randomNumber = '3';
7
8  // Argument of type 'string' is not assignable to parameter of type 'number'
9  const sum = multiplyNumberByTwo(num: '2');
10
```

KUVA 1. Esimerkkikuva tyypitysvirheiden huomaamisesta ennen ajoa.

TypeScriptin suurimpana eroavaisuutena JavaScriptiin on tyypitys. Niin kuin kuvasta 1 huomataan, muuttujille määritellään tyypit, minkä jälkeen muuttujaa ei voi käsitellä erityyppisenä. Tämä mahdollistaa ohjelmointivirheiden huomaamisen ennen kuin ohjelmaa ajetaan.

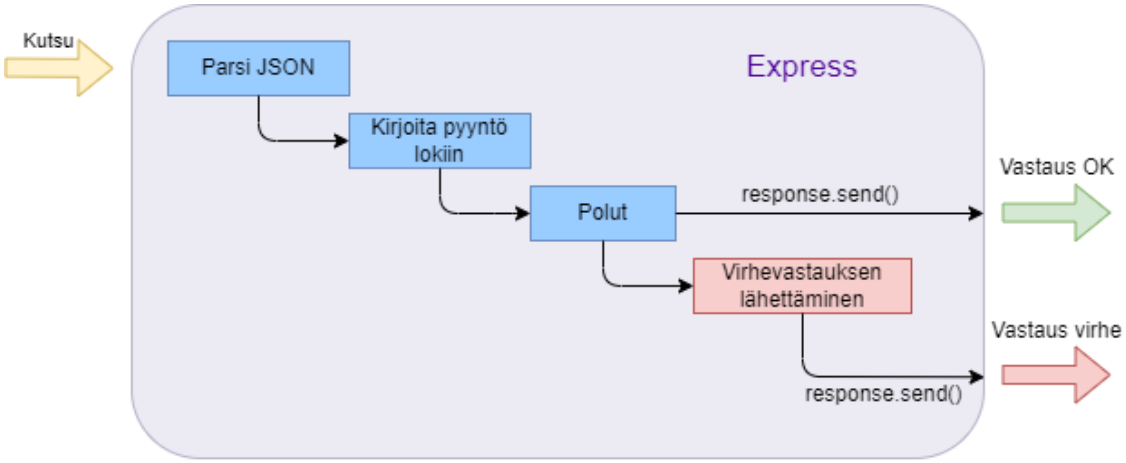
3.3 Express-sovelluskehys

Express on tällä hetkellä suosituin Node.js-sovelluskehys. Express itsessään on hyvin minimalistinen ja joustava, mikä mahdollistaa sovelluksen toteuttamisen usealla tavalla. Siihen on saatavilla lukuisia npm-paketteja, jolla sitä voi laajentaa. (Mozilla 2022a.)

3.3.1 Expressin tuoma toiminnallisuus

Express toimii samalla periaatteella kuin Node.js. Kutsu tulee callback-funktiona, joka kulkeutuu sovelluksen funktioiden läpi samassa järjestyksessä, mitä ne on

koodiin kirjoitettu. Ohjelman suoritus kutsun osalta päättyy, kun jokin funktio vastaa callback-funktion parametrina tulleella Response-objektilla. (Subramanian 2019.) Kuviossa 5 on kuvattu Express-sovelluksen esimerkkirakennetta.



KUVIO 5. Esimerkkirakenne Express-sovelluksesta.

Express ei pääpaketissaan tuo kaikkea mahdollista toiminnallisuutta, vaan siihen on olemassa lukuisia lisäpaketteja, joilla voi halutessaan lisätä toiminnallisuutta. Express luo palvelinkehittämisestä yksinkertaisempaa, jos vertaa Node.js palvelinohjelmointiin ilman erillistä sovelluskehystä tai kirjastoa.

Express yksinkertaistaa alkuperäisiä callback-funktion parametreina tulevaa Request- ja Response-objektia. Request-objekti pitää sisällään pyyntöön liittyviä tietoja valmiiksi parsittuna, kuten muun muassa polun (path), otsikot (headers), kyselyparametrit (params), kyselyrungon (body).

Myös Response-objektin metodit ovat yksinkertaisempia ja Express muun muassa muuttaa automaattisesti datan JSON-muotoon ennen lähettämistä. Kuvassa 2 lähetetään viesti ”Hello world!” -juuripolusta. HTML:ää lähettäessä ei tarvitse luoda koko HTML-dokumenttia, vaan riittää halutun HTML-elementin lähettäminen.

```
app.get( path: '/', (req: Request, res: Response) =>
  res.status( code: 200).send({ message: 'Hello world!' })
);
```

KUVA 2. Juuripolusta lähetetään onnistunut vastaus ”Hello world!”

Node.js:ssä itsessään ei ole työkaluja polkujen reitittämiseen, kun taas Express tuo Router-objektin, jolla on mahdollista muun muassa ketjuttaa polkuja yhteen. Express tuo myös paljon, mutta ei kaikkea, valmista toiminnallisuutta monelle osa-alueelle, esimerkiksi virhekäsittelyyn ja turvallisuuteen.

3.3.2 Polut

Palvelimella on poikkeuksetta useita polkuja, joihin lähetetään kutsuja. Näihin Expressillä vastataan asianmukaisella funktiolla (get, post, put, delete), jolle annetaan polku parametrina. (Mozilla 2022b.) Esimerkiksi kuvassa 2 vastataan GET-pyyntöön juuripolussa. Kuvitteellisesti haluttu poistettavan resurssin polku voisi olla api/user/:userId, jolloin kutsuttaisiin delete-funktiolla kuvan 3 tavalla.

```
app.delete( path: '/api/user/:userId', (req: Request, res: Response) => {
  const { userId } = req.params;
  console.log( message: 'userId ', userId);
  res.status( code: 202).send({ msg: 'Deleted successfully' });
});
```

KUVA 3. Esimerkki delete-kutsusta.

Polkuparametriin päästään käsiksi helposti lukemalla suoraan Request-objektista, jonka Express parsii automaattisesti. Polkuja voi olla suurikin määrä, jolloin niitä on hyvä ryhmitellä. Tähän Express tarjoaa Router-objektin, jolla voi ketjuttaa polkuja yhteen.

```
import boardRoutes from './boardRoutes';

const routes = Router();
routes.use( path: '/api/board', boardRoutes);
```

```
import { Router } from 'express';
const router = Router();

router.delete( path: '/:id', BoardController.deleteBoard);
router.get( path: '/', BoardController.getAll);
router.post( path: '/', BoardController.postBoard);

export default router;
```

KUVA 4. Esimerkki polkujen ketjuttamisesta.

Router-objektiin määritellään kutsutyyppi, polku ja suoritettavat funktiot. Kuvassa 4 alemman kuvan funktioiden polut on ketjutettu ylemmän kuvan polkuun, eli alemman kuvan polkujen eteen liitetään ’/api/board/’. Esimerkiksi alemman kuvan board-resurssin poistaminen tapahtuu polussa ’/api/board/:id’, jossa :id vastaa polkuparametria.

3.3.3 Middleware-funktiot

Middleware-funktioksi kutsutaan funktioita, joilla on pääsy Request- ja Response-objekteihin. Middleware-funktiolla on Response- ja Request-parametrien lisäksi myös next-funktio. Jokainen middleware-funktio täytyy päättyä joko kutsuun vastaamiseen Response-objektilla tai kutsua seuraavaa middleware-funktiota next-funktiolla. Jos näin ei tee, kutsu jää roikkumaan käsittelemättömänä APIin. (Express 2017.) Kutsun tullessa Express suorittaa middleware-funktiota samassa järjestyksessä, kuin ne on kirjoitettu.

```
const logRequest = (req: Request, res: Response, next: NextFunction) => {
  console.log( message: 'request', req);
  next();
};

const app = express();

app.use(logRequest);
app.get( path: '/', (req: Request, res: Response) =>
  res.status( code: 200).send({ msg: 'Hello world!' })
);
```

KUVA 5. Esimerkki middleware-funktion lisääminen, joilla kirjoitetaan lokiin kutsu.

Middleware-funktioita voidaan käyttää esimerkiksi kirjoittamaan kutsu lokiin ennen kuin kutsua vastaavan polun funktio suoritetaan tai lähettää virheviesti, jos ei löytynyt kutsun mukaista polkua. Kuvassa 5 kutsun tullessa APIin kirjoitetaan lokiin Request-objektin sisältö, ennen kuin se kulkeutuu juuripolkua vastaavalle käsittelijälle.

3.3.4 Virhekäsittely

Palvelinohjelmointi sisältää runsaasti asynkronista koodia, kun esimerkiksi kutsutaan tietokantaa tai muita palveluita. Asynkronisessa koodissa on aina mahdollisuus erilaisille virhetilanteille. Ohjelman ajonaikana tullut käsittelemätön virhe lopettaa palvelimen suorittamisen. Expressissä ei ole oletuksena virhekäsittelyä asynkronisessa koodissa viimeisimmässä versiossa (4.17.3), joten virhekäsittely täytyy luoda itse. Express tuo kumminkin valmiita rakennuspalikoita oman virhekäsittelyn toteuttamiseen.

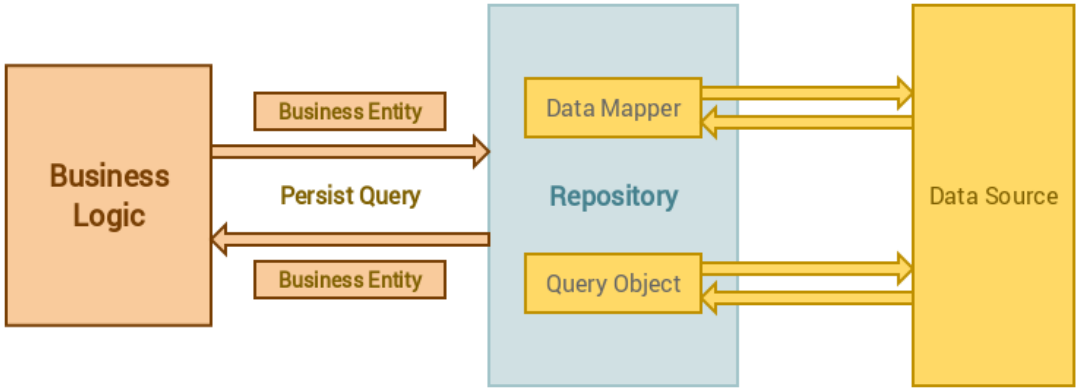
Virhekäsittely-middleware voidaan toteuttaa next-funktiolla samoin tavoin kuin muukin middleware-funktio. Kun next-funktiota kutsutaan error-parametrin kanssa, ohjelman suoritus jatkuu seuraavasta middleware-funktioista, joka tavallisten request-, response- ja next-parametrien sijaan ottaa vastaan error-, request-, response- ja next-parametrin. (Khanna 2021.)

3.4 PostgreSQL-tietokanta

PostgreSQL on avoimen lähdekoodin relaatiotietokannan hallintajärjestelmä, joka toimii käytetyimmillä käyttöjärjestelmillä, kuten Windowsissa, Linuxilla ja muilla UNIX-pohjaisilla järjestelmillä. PostgreSQL:n on suosittu luotettavuuden ja suorituskykynsä vuoksi. Tavallisten SQL-ominaisuuksien lisäksi PostgreSQL:ssa on automaattisesti päivittyviä näkymiä ja se tarjoaa paljon datan validointiin työkaluja. (Hammink & Poso 2020.)

3.5 ORM-ohjelmointitekniikka

ORM (Object-relational mapping) on abstrahointikerros, joka mahdollistaa tietokantakyselyt käyttämällä pelkästään olio-ohjelmointia SQL-kyselyjen sijaan. Tämän toiminnallisuuden tekemiseksi on olemassa monenlaisia sovelluskehyksiä ja kirjastoja, mutta usein näitä yhdistää käsitteet ”entity” ja ”repository”.



KUVIO 6. ORM-kerroksen osat (Bosnadev, 2015).

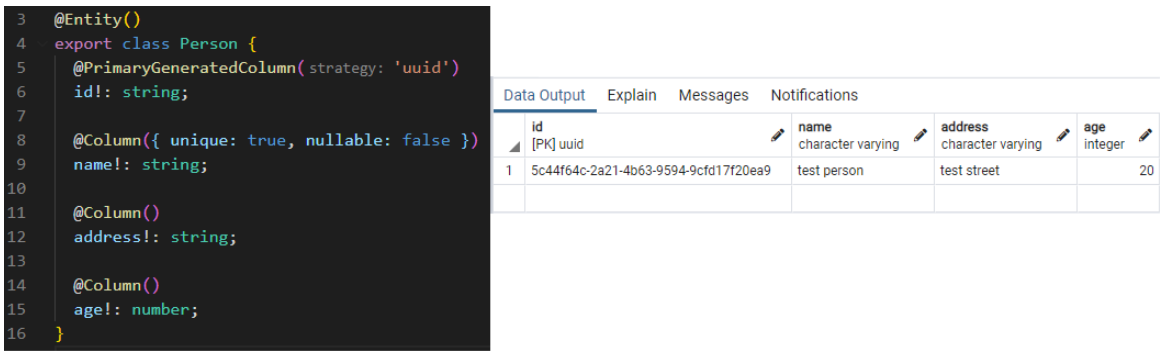
Entity on olioluokka, jonka perusteella määritellään tietokantataulu. Repository on objekti, jonka kautta entity-luokkien avulla tehdään ohjelmallisesti kyselyjä tietokantaan (DEV 2022).

3.5.1 TypeORM-kirjasto

TypeORM on ORM-kirjasto, jota voidaan käyttää muun muassa Node.js-ympäristössä. Se tukee JavaScriptiä ja TypeScriptiä. TypeORM on ottanut paljon vaikutteita muun muassa Hibernatesta ja Entity Frameworkista, mikä näkyy samankaltaisena toimintalogiikkana ja syntaksina (TypeORM n.d).

3.5.2 Entity-luokka

Entity on luokka, jolla määritellään tietokantaan luotava taulu. TypeORMissa luokka määritellään @Entity-annotaatiolla ja jokaista tietokantataulua vastaava sarake määritellään @Column-annotaatiolla. Annotaatioiden sulkujen sisälle voi antaa halutessaan objektin, jonka avulla voi antaa erilaisia tarkentavia parametrejä, esimerkiksi voiko arvo olla null tai alustaa jollain tietyllä arvolla.



```
@Entity()
export class Person {
  @PrimaryGeneratedColumn( strategy: 'uuid')
  id!: string;

  @Column({ unique: true, nullable: false })
  name!: string;

  @Column()
  address!: string;

  @Column()
  age!: number;
}
```

Data Output | Explain | Messages | Notifications

	id [PK] uuid	name character varying	address character varying	age integer
1	5c44f64c-2a21-4b63-9594-9cfd17f20ea9	test person	test street	20

KUVA 6. Havainnointikuva Entity-luokan muuttumisesta tietokantatauluksi.

Jokaiselle taululle täytyy myös määritellä yksikäsitteinen pääavain. Tämän voi tehdä usealla tavalla, mutta yksi yksinkertainen tapa on määritellä luokalle oma attribuutti ja merkitä annotaatiolla @PrimanyGeneratedColumn, jolloin TypeORM huolehtii automaattisesti pääavaimen luonnin. (TypeORM n.d.)

3.5.3 Repository-objekti

Repository on TypeORMin tarjoama objekti, jolla voi tehdä kyselyjä entity-luokkien avulla. Repository tarjoaa paljon erilaisia metodeja, jolla voi tehdä kyselyitä ja valmiita metodeja voi täydentää ja tarkentaa erilaisin parametrein, kuten kuvassa 7 on tehty.

```
const allOldMikes = await personRepository.find({
  where: { firstName: 'Mike', age: MoreThan( value: 70) },
  order: {
    age: 'DESC'
  }
});
```

KUVA 7. Repository-kysely, tietyn nimiset ja ikäiset henkilöt vanhimmasta nuorimpaan.

Jos repository-luokan metodeilla ei saa tuotettua haluamaan kyselyä, voi käyttää TypeORMin tarjoamaa QueryBuilderia tai kirjoittaa jopa suoraan SQL-kyselyitä query-metodilla.

```
@EntityRepository(Topic)
export class TopicRepository extends Repository<Topic> {
  async findTopicWithUserByTopicId(topicId: string): Promise<Topic> {
    const response = await this.createQueryBuilder( alias: 'topic')
      .where( where: 'topic.id = :id', { id: topicId })
      .leftJoinAndSelect( property: 'topic.user', alias: 'user')
      .getOne();

    if (!response)
      throw new ResponseError( message: 'Topic not found', errorType: 'ENTITY_NOT_FOUND');
    else return response;
  }
}
```

KUVA 8. Repository-luokan laajentaminen omalla metodilla.

TypeORMin repository-luokkaa on mahdollista periyttää ja laajentaa haluamallaan toiminnallisuudella samalla kapseloiden toiminnallisuuden repository-tasolle. Kuvassa 8 esimerkiksi on haettu keskusteluaihetta tämän tunnisteen perusteella ja palautetaan se ja siihen liittyvän käyttäjän tiedot.

3.5.4 Relaatioiden muodostaminen

Tietokantatauluilla on usein keskenään relaatioita erilaisin suhtein. Esimerkiksi keskustelualueelle voi kuulua useita viestejä tai jokaisella viestillä on tietty kirjoittaja. Tai käyttäjällä voisi olla yksi tietty asetusprofiili. Relaatiotietokannoissa tietokantataulujen välisiä suhteita ylläpidetään vierasavaimilla ja liitostauluilla.

Koska käytetään ORM-tekniikkaa, jolloin käsitellään tietokantatauluja olioina, ei tarvitse ohjelmallisesti määritellä vierasavaimia. Jotta tietokantojen väliset relaatiot tallentuvat oikein tietokantaan, eli TypeORM osaa generoida vierasavaimet ja mahdolliset liitostaulut oikein, määritellään tietokantataulujen relaatiot olioviitteillä ja annotaatioilla.

TypeORMissa entityiden väliset relaatiot määritellään annotaatioilla, joita ovat @OneToMany, @ManyToOne, @ManyToMany ja @OneToOne

Kuvassa 9 käyttäjä-entityllä on luotu posts-attribuutti, joka on käyttäjän viestit ja viesti-entitylle on user-attribuutti, joka on viestin tietty käyttäjä. Relaation omista-

valle puolelle (käyttäjä-entity) kirjoitetaan @OneToMany-annotaatio, jolla määritellään mihin viesti-entityn attribuuttiin olioviite kytketään. Vastaavasti viesti-entitylle määritellään relaation alistuva puoli, eli mihin käyttäjä-entityn attribuuttiin linkitetään viesti-entityn user-attribuutti.

```
/** Käyttäjän entity */
@OneToMany(() => Post, (post) => post.user)
posts?: Post[];
```

```
/** Viestin entity */
@ManyToOne(() => User, (user) => user.posts, { onDelete: 'CASCADE' })
user!: User;
```

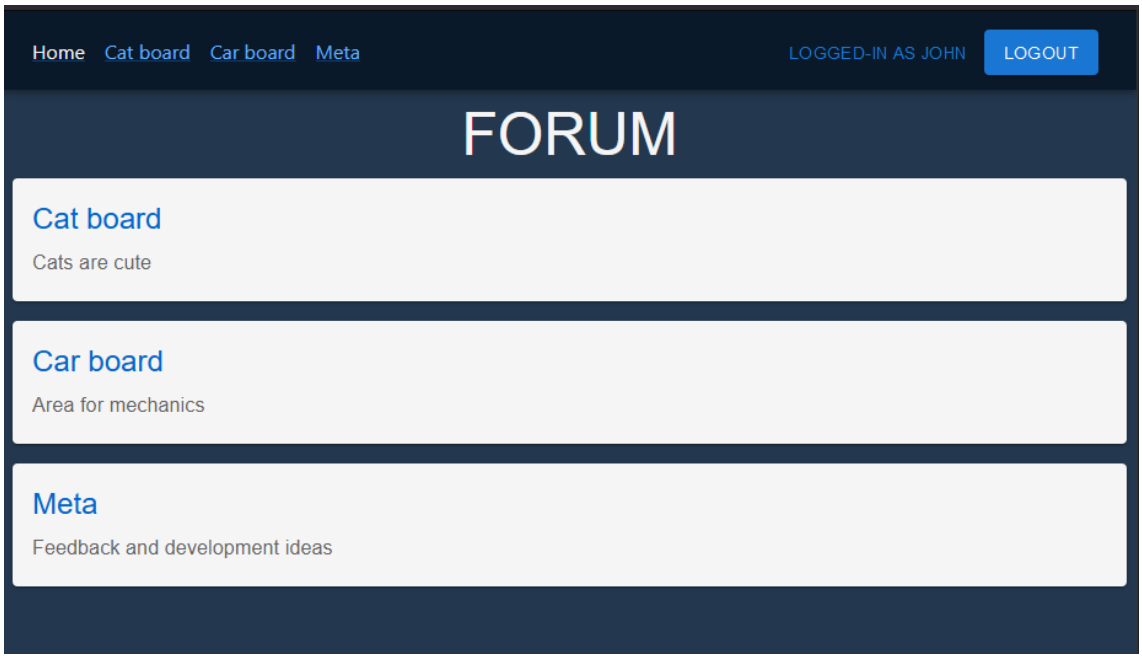
KUVA 9. Relaation tekeminen entityillä

Tämän linkittämisen jälkeen TypeORM osaa luoda oikein mahdolliset vierasavaimet. Monen suhde moneen- ja yhden suhde yhteen -relaatioissa TypeORM osaa luoda automaattisesti myös mahdolliset liitostaulut.

Relaatioannotaatioiden yhteydessä voidaan antaa erilaisia tarkentavia parametreja relaation liittyen. Kuvan ”onDelete: ’CASCADE’” käyttäjän poistaessa poistaa myös kaikki siihen liittyvät viestit tietokannasta. Toinen yleinen parametri on eager, eli haetaanko entityn yhteydessä myös kaikki siihen liittyvät entityt.

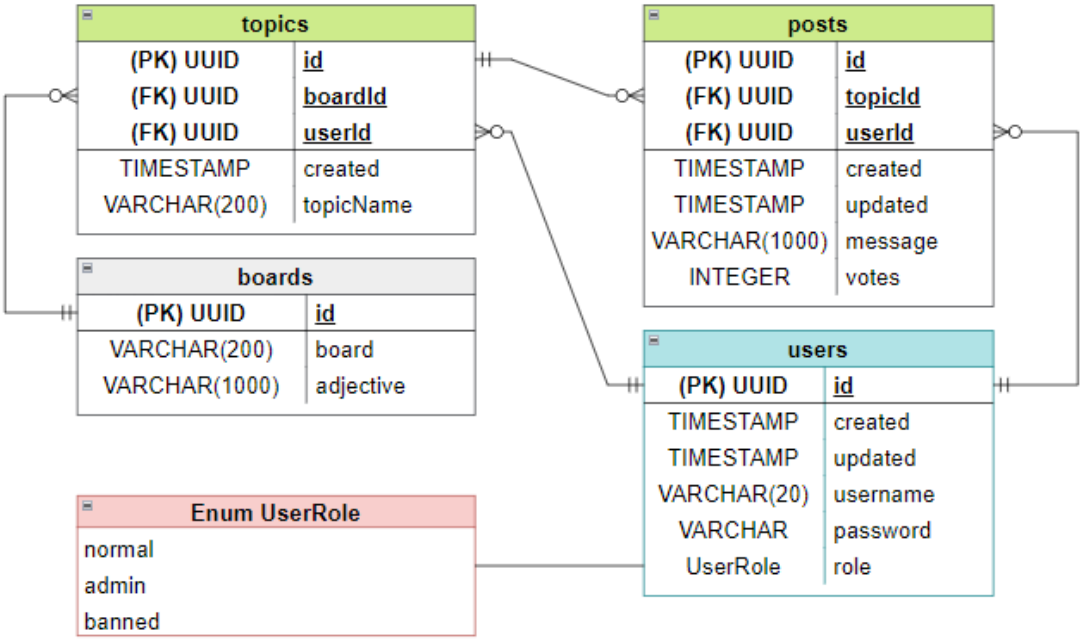
4 TOTEUTUS

Opinnäytetyössä tehty API käsittelee web-applikaatioon liittyvää dataa ja on tehty opiskelijan aikaisemmin toteuttamalle web-applikaatiolle, joka on keskustelufoorumi, missä käyttäjät voivat lähettää viestejä eri keskusteluaiheisiin. Web-sivustolla on yksinkertainen toteutus: sivustolla voi luoda käyttäjän ja tämän jälkeen aloittaa ja jatkaa keskusteluja eri keskustelualueilla. Kuvassa 10 on kuvakaappaus web-applikaation etusivusta.



KUVA 10. Kuvakaappaus keskustelufoorumin etusivusta.

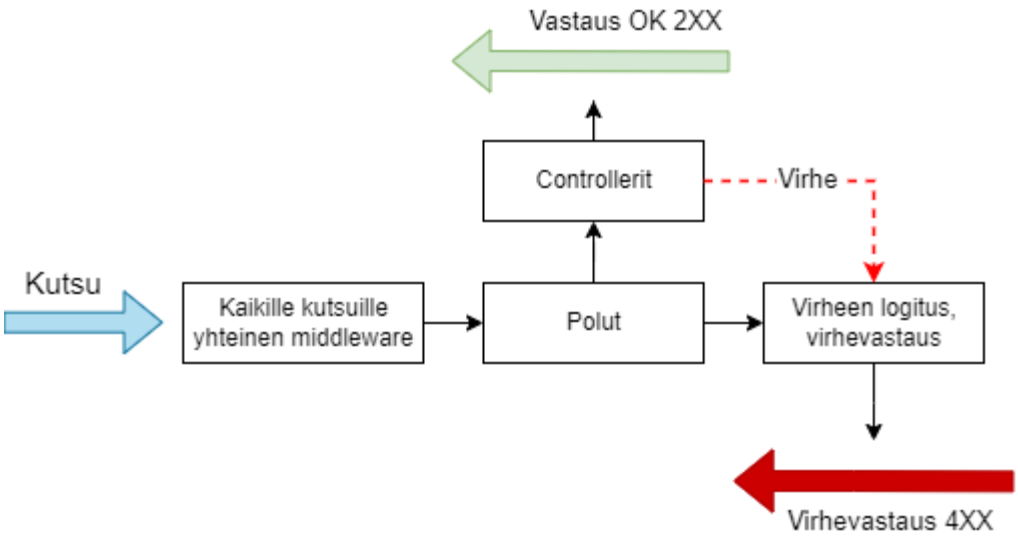
Keskeisimmät käsitteet ovat keskustelualue (board), keskusteluaihe (topic), viesti (post) ja käyttäjä (user). Käsitteiden välisiä suhteita on kuvattu kuviossa 7. Applikaatiossa on keskustelualueita, joihin kuuluu keskusteluaiheita. Keskusteluaiheisiin kuuluu viestejä. Viestit ja keskusteluaiheet kuuluvat tietylle käyttäjälle. Käyttäjällä voi olla erilaisia rooleja: tavallinen käyttäjä, käyttäjällä voi olla admin-oikeudet tai käyttäjällä on porttikielto.



KUVIO 7. Applikaation tietokannan kuvaus.

4.1 Yleiskuvaus APIn toiminnasta

Expressissä kutsu kulkeutuu pipeline-mekanismin tavoin APIn läpi, kunnes siihen vastataan Response-objektilla. Ohjelmaan tuodaan middleware-funktioita, joilla käsitellään Request- ja Response-objekteja kirjoitusjärjestyksessä. Ohjelman rakenne on yksinkertaistettuna kuvion 8 mukainen.



KUVIO 8. Sovelluksen rakenne.

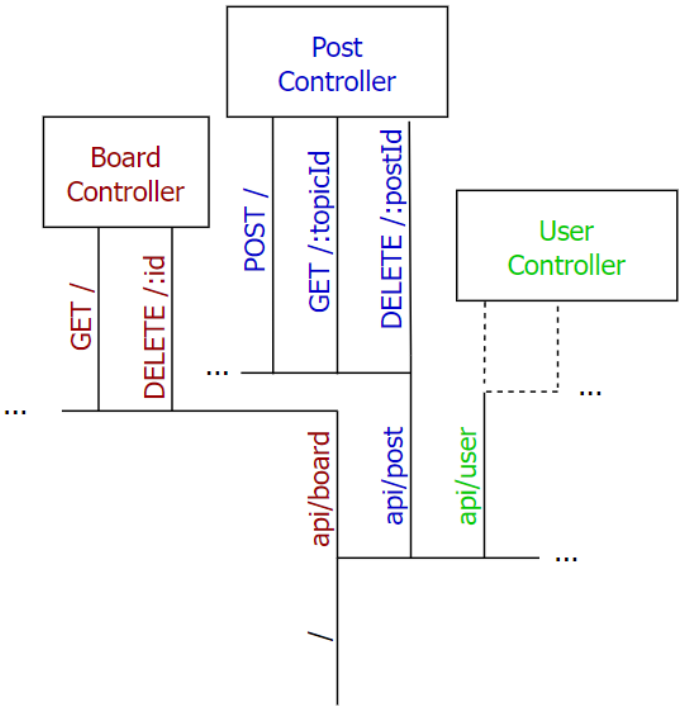
Jotta TypeORMin tarjoamaa ORM-toiminnallisuutta voidaan käyttää, täytyy luoda tietokantayhteys kirjaston createConnection-funktiolla kuvan 11 mukaisesti. Funktion sisälle määritellään ne osat, joissa ORM-toiminnallisuutta käytetään, jolloin TypeORM injektoi toiminnallisuuden niihin.

```
const app = express();

createConnection(DB_CONNECTION_SETTINGS)
  .then(() => {
    app.use(cookieParser());
    app.use(express.urlencoded( { extended: true } ) );
    app.use(express.json());
    app.use(requestLogger);
    app.use( path: '/', routes);
    app.use(unknownEndpoint);
    app.use(errorLogger);
    app.use(errorResponser);
  })
  .catch((error) => logger.error(error));

export default app;
```

KUVA 11. Express-middlewaret ja TypeORM-yhteyden muodostaminen.

4.2 Polut ja controller-taso

Applikaation resursseille on luotu omat polut APIin, joihin voi tehdä http-kyselyitä. API noutaa näitä resursseja ja tietokannasta ja toteuttaa tarvittavan toiminnallisuuden.



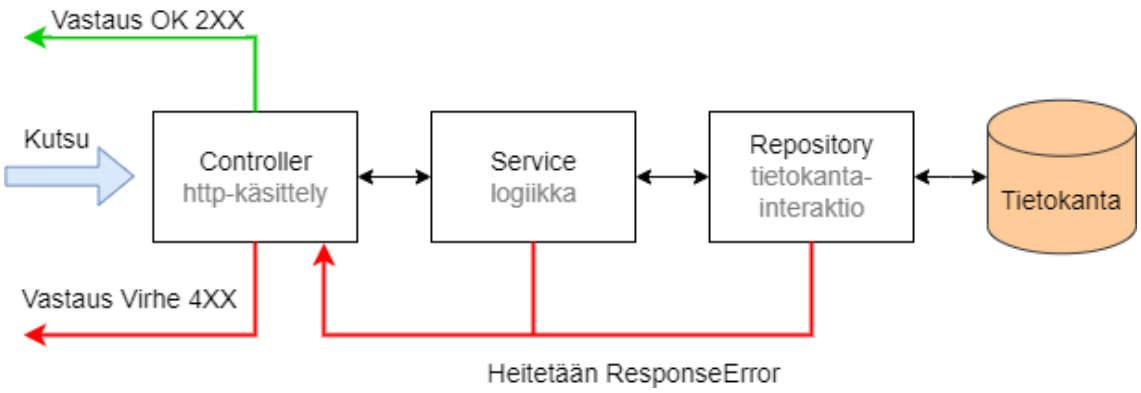
KUVIO 9. Polkuja APIssa

Samaa hierarkiaa on noudatettu Expressissä polkujen ketjuttamisessa. Kuvassa 10 on tuotu Express-applikaation päätasolle vain juuripolku, johon on ketjutettu router-objekteilla jokaisen resurssin vastaava polku, esimerkiksi api/post. Resurssin router-objektiin on vielä ketjutettu toinen router-objekti, joka sisältää kaikki tietyn resurssiin liittyvät polut ja määrittää, mitä controller-funktiota kutsutaan kussakin polussa.

```
const router = Router();

router.delete( path: '/:id', PostController.deletePost);
router.get( path: '/:topicId', PostController.findAllByTopicId);
router.post( path: '/', PostController.saveOne);

export default router;
```

KUVA 12. Esimerkiksi api/post -polkuun ketjutetut polut ja controller-funktiot.

Esimerkiksi kutsun tullessa api/post/:topicId -polkuun Express välittää kutsun, joka sisältää Request- ja Response-objektit. Näin kaikki http-kutsuun liittyvä käsittely voidaan tehdä controller-tasolla ja tarvittaessa kutsua ylempiä tasoja toteuttamaan logiikkaa.



KUVIO 10. Polkujen jälkeinen rakenne APIssa.

```
export const findAllByTopicId = async (req: Request, res: Response, next: NextFunction) => {
  try {
    res.send(await findPostsByTopicId(req.params.topicId));
  } catch (error) {
    next(error);
  }
};
```

KUVA 13. Controller-tason funktio, jolla haetaan viestit keskusteluaiheen tunnisteella

Virhetilanteessa service- ja repository-tasolta aiheutetaan virhe, joka controller-tasolla vastaanotetaan ja lähetetään next-funktiolla virhekäsittelyn hoitavalle middleware-funktiolle.

4.3 Service-, repository-taso ja ORM-tekniikan käyttö

TypeORM mahdollistaa kaikkien tietokantakyselyjen tekemisen TypeScriptillä ja entity-luokilla ilman SQL-kyselyitä. TypeORMin injektoitu ORM- ja repository-toiminnallisuus otetaan käyttöön käyttämällä getCustomRepository-funktiota, jolla haetaan olion instanssi, joka tarjoaa interaktion tietokannan kanssa entity-luokilla.

```
export const saveOne = async (
  topicName: string,
  boardName: string,
  userId: string
): Promise<Topic> => {
  const topicRepository = getCustomRepository(TopicRepository);

  const board = await BoardService.findBoardByBoardName(boardName);
  const postedUser = await UserService.findOne(userId);

  const topic = new Topic();
  topic.board = board;
  topic.topicName = topicName;
  topic.user = postedUser;

  const errors = await validate(topic);

  if (errors.length)
    throw new ResponseError( message: 'Topic body invalid', errorType: 'INVALID_REQUEST_BODY');
  else return await topicRepository.save(topic);
};
```

KUVA 14. Service-tasolla tallennetaan uusi keskusteluaihe.

Kuvassa 14 tallennetaan uutta keskusteluaihetta tietokantaan. Keskusteluaiheella on kaksi riippuvuutta muiden entity-luokkien kanssa: keskusteluaihe kuuluu tiettyyn keskustelualueeseen (monen suhde yhteen) ja keskusteluaihe kuuluu tietylle käyttäjälle (monen suhde yhteen). Jotta suhteet säilyvät oikein tietokannassa, täytyy keskusteluaihe-entitylle asettaa ennen tallentamista keskustelualue- ja käyttäjä-entity.

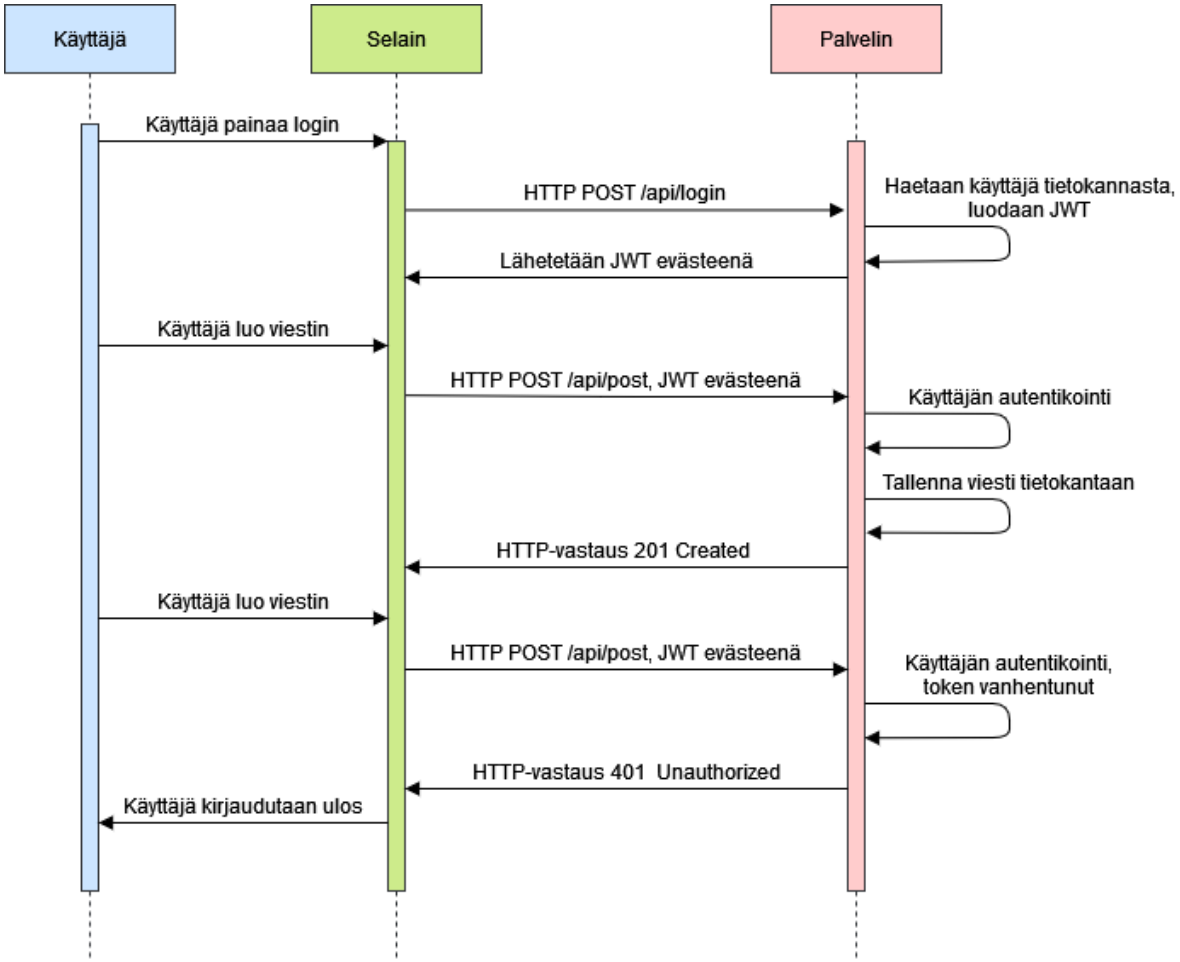
Koska jokaisella resurssilla on oma service- ja repository-taso, voidaan haluttu board- ja user-entity hakea suoraan näiden service-tasolta. Tällä vältetään duplikaattikoodin kirjoittaminen ja koodin ylläpidettävyys paranee, kun resurssia käsiteltävä toiminnallisuus on keskitetty tietyn resurssin service-tasolle. Nämä muista service-komponenteista haetut entity-oliot asetetaan uudelle topic-oliolle, jolloin topic-olion tallentaessa TypeORM luo automaattisesti tietokantaan tarvittavat vierasavaimet.

4.4 Autentikointi

Sovelluksessa on käyttäjiä ja tietyille käyttäjille tarkoitettu resursseja, joten APIssa täytyy pystyä tunnistamaan käyttäjä. Käyttäjän kirjautuessa vertaillaan http-kutsun mukana tulleita tietoja tietokannassa olevaan käyttäjään. Jos käyttäjä

löytyy ja suolattu salasana täsmää, luodaan käyttäjälle JWT, joka sisältää käyttäjän nimen ja tunnisteen. JWT lähetetään evästeenä selaimelle. Evästeelle ja JWT:lle asetetaan voimassaoloaika, milloin se voimassa palvelimella.

Evästeenä oleva JWT liitetään aina kutsuun, kun selain lähettää http-pyynnön palvelimelle. JWT dekoodataan, jolloin JWT:stä saadaan käyttäjän nimi ja tunniste. Käyttäjän nimellä ja tunnisteella tarkistetaan, onko tietokannassa kyseinen käyttäjä olemassa. JWT ei ole purettavissa palvelimen ulkopuolella eikä myöskään voi generoida palvelimen ulkopuolella, koska JWT:n generoituvaan tiivisteeseen lisättiin vain palvelimen tietämä salainen osa.



KUVIO 11. JWT:n kulku koko applikaation kontekstissa.

Kuviosta 11 nähdään esimerkkikulku JWT:n luomisesta ja kulkeutumisesta selaimen ja palvelimen välillä. Jos autentikointi ei onnistu esimerkiksi vanhentuneen tokenin takia, ei suoriteta kutsun pyydettyä toiminnallisuutta ja lähetetään 401-viesti (unauthorized) selaimelle, mikä viestii epäonnistuneesta autentikoinnista tai

sen puuttumisesta. Jos autentikointi on onnistunut, mutta yritetään toimia resurssin kanssa, johon ei ole oikeuksia, lähetetään 403-viesti (forbidden).

4.5 Virhekäsittely

Koska Express ei tarjoa oletuksena virheidenkäsittelylle mitään oletustoiminnallisuutta asynkronisessa koodissa, täytyy sellainen luoda itse. Mahdollisuuksia tämän toteuttamiseen on monenlaisia, tässä APIssa controller-funktiot ovat ympäröity try-catch-toiminnallisuudella, joka vastaanottaa ylemmiltä tasolta aiheutetut virheet ja toimittaa ne keskitetylle virhekäsittely-middlewarelle.

4.5.1 Kustomoitu virhe

Jotta virheiden käsittely helpottuu, luodaan oma virheluokka, jonka attribuutteina ovat viesti ja virheen tyyppi. Natiivin Node.js:n virheluokan laajentaminen on suhteellisen raskasta, joten hyödynnetään npm:stä löytyvää kevyttä kirjastoa nimeltä ts-custom-error, jolla on suoraviivaista luoda oma virheluokka.
Peritään kirjaston tarjoama CustomError-luokka, joka ei sisällä käytännössä mitään ja laajennetaan omalla halutulla toiminnallisuudella kuvan A mukaisesti.

```
export default class ResponseError extends CustomError {
  public constructor(message: string, public errorType: ErrorType) {
    super(message);
  }
}
```

KUVA 15. Laajennettu virheluokka.

Luodaan luokalle viesti ja virhetyyppiattribuutit. Virhetyyppi (ErrorType) attribuutti on joukko merkkijonovakioita, joiden avulla virhevastaus-middlewaressa valikoituu oikea virhekoodi lähetettävään vastaukseen.

4.5.2 Uuden virheen aiheuttaminen ja käsittely

Controller-tasolla kutsutaan ylempiä tasoja try-catch-rakenteella mahdollisten virhetilanteiden varalta. Jos virhettä ei tapahdu, lähetetään try-lohkossa onnistunut vastaus selaimelle.

Virhetilanteen tullessa, esimerkiksi kuvassa 15 ja 16 nimeä vastaavaa käyttäjää ei löydy, service- tai repository-tasolla aiheutetaan virhe, joka napataan controller-tasolla catch-lohkossa. Controller kutsuu virhekäsittely-middlewarea, jolle annetaan virhe parametrina. Kutsuttu middleware-funktio suorittaa virheen lokin kirjaamisen ja lähettää virheen seuraavalla middleware-funktiolle, joka suorittaa vastauksen lähettämisen selaimelle.

```
export const login = async (req: Request, res: Response, next: NextFunction) => {
  try {
    const { username, password } = req.body;

    await UserService.verifyUser(username, password);

    const token: string = await UserService.getToken(username);

    res.cookie(accessTokenName, token, {
      maxAge: 1000 * 60 * 60,
      httpOnly: true,
      secure: true
    });

    const user = await UserService.findUserByUsername(username);

    res.status( code: 200).send({ user });
  } catch (error) {
    next(error);
  }
};
```

KUVA 16. Controller-taso, joka vastaanottaa service- ja repository-tasolta aiheutetut virheet.

```
async findUserByUsername(param_username: string): Promise<User> {
  const user = await this.createQueryBuilder( alias: 'user')
    .where( where: 'user.username = :username', { username: param_username })
    .addSelect( selection: 'user.passwordHash')
    .getOne();

  if (!user) throw new ResponseError( message: 'User not found', errorType: 'ENTITY_NOT_FOUND');
  else return user;
}
```

KUVA 17. Aiheutettu virhe repository-tasolla

Vastauskoodiksi asetetaan virhetyyppiä vastaava vastauskoodi ja vastauksen sisällöksi asetetaan käsiteltävän virheen viesti. Kuvassa 18 on typistetty versio virhevastauksen hoitavasta errorResponser-funktiosta.

```
export const errorResponser = (err: ResponseError, req: Request, res: Response) => {
  const sendResponse = (res: Response, message: string, statusCode: number) =>
    res.status(statusCode).send(message);

  switch (err.errorType) {
    case 'AUTHORIZATION_FAILED':
      sendResponse(res, err.message, statusCode: 401);
      break;
    case 'ENTITY_NOT_FOUND':
      sendResponse(res, err.message, statusCode: 404);
      break;
    default:
      res.sendStatus( code: 500);
      break;
  }
};
```

KUVA 18. Typistetty funktio virheviestin lähettämiseen.

Virheviestin lähettäminen on keskitetty erillisiin middleware-funktioihin, jolloin mahdollisia virheviestejä ei tarvitse käsitellä controller-tasolla. Ohjelma jakautuu paremmin moduuleihin, missä kapseloidaan tietty toiminnallisuus tiettyyn paikkaan.

Ratkaisun hyvänä puolena voidaan pitää sitä, että controller-taso yksinkertaistuu, kun virhekäsittely on ulkoistettu. Toinen merkittävä etu on TypeScriptillä ohjelmoitaessa, että funktioiden paluutyypiksi ei tarvitse määritellä undefined tai null. Jos näin joutuisi tekemään, täytyisi alemmilla tasoilla huomioida funktioiden kaikki mahdolliset paluuarvot, mikä tekisi näiden tasojen rakenteesta monimutkaisempia.

Huonona puolena on, että service- ja repository-tasolta voi tulla muitakin virheitä kuin ResponseErroreita, esimerkiksi TypeORM-kirjaston aiheuttamia virheitä. Koska muilta virheluokilta ei löydy errorType-attribuuttia, vastataan niihin geneerisellä 500-koodilla ja ”Internal Server Error”-viestillä switch-case -rakenteen default-osiossa.

5 POHDINTA

On helppoa ymmärtää, miksi Node.js on suosittu. Node.js:n sisäinen tapahtumasilmukka ja siihen liittyvä toiminnallisuus mahdollistaa nopean palvelimen, joka skaalautuu helposti suuriin kutsumääriin. Toisaalta Node.js ei sovellu raskaaseen laskentaan, koska se estää tapahtumasilmukan toiminnan, jolloin kutsujen käsittely estyy.

Vaikka Node.js-kehitysalusta ja Express-sovelluskehys ovat suhteellisen uusia, silti niitä käytetään suurissa järjestelmissä ja yrityksissä. Esimerkiksi Twitter, Netflix, PayPal, eBay ovat ilmoittaneet käyttävänsä Express-sovelluskehystä.

Tämän lisäksi Node.js voi ohjelmoida JavaScriptillä, jolloin siirtymä web-ohjelmoinnista pienenee huomattavasti. Tosin JavaScriptin soveltuvuudesta palvelinohjelmointikieleksi ollaan montaa mieltä, yhtenä suurena ongelmana pidetään sitä, että JavaScriptissä ei ole tyypitystä funktioiden paluuarvoilla ja muuttujilla. Tähän ongelmaan osittainen ratkaisu on ollut TypeScript-ohjelmointikieli, joka laajentaa JavaScriptiä juuri tyypityksillä. Opinnäyteyötä tehdessä TypeScriptin tyypitys helpotti havaitsemaan bugeja jo ennen ajoa. Pääsääntöisesti koodin tyypittäminen vie vähemmän aikaa kuin mahdollisten outojen bugien selvittäminen.

Node.js:n ympärille on muodustunut hyvin laaja ekosysteemi. Kirjastoja ja sovelluskehyksiä on todella paljon, mikä nopeuttaa ohjelmointia. Tosin varjopuolena laajalle kirjastovalikoimalle on se, että on myös paljon heikkoja kirjastoja ja kirjastojen kehitys ja ylläpito voi perustua vapaaehtoistyöhön, jolloin kirjaston ylläpidosta tulevaisuudessa ei ole takeita. Tämä on varmasti ongelmallista tuotantotason ohjelmistoissa, joilla voi olla pitkä elinkaari.

Tiedonhaussa tuli myös jonkin verran vastaan keskusteluita, joissa mietittiin Node.js-kehitysalustan ja npm-pakettien turvallisuutta ja niistä löytyneitä haavoittuvuuksia. Tähän voi tietysti vaikuttaa valitsemalla vain tietynlaisia npm-paketteja ja yrittää pitää mahdollisimman vähäisinä riippuvuudet kolmannen osapuolen npm-paketteihin. npm-työkalusta löytyy myös toimintoja, joilla voi skannata haavoittuvuuksia projektiin asennetuista npm-paketeista.

Myös Expressissä on havaittavissa hieman samankaltaisia ongelmia. Sovelluskehys on laaja, muokattavissa ja ei pakota rakentamaan ohjelmaa tietyllä tavalla. Tämän takia tietoa etsiessä törmäsi hyvin erilaisiin ratkaisutapoihin, mikä teki tiedonhankinnasta vaikeampaa. Suuri vapaus kääntyykin hieman haitaksi, kun täytyy pysyä aikaisin valitussa arkkitehtuuriratkaisussa, mitä on vaikea muokata myöhemmin.

Node.js soveltuu erinomaisesti erilaisiin protoiluihin tai sovellusten koeponnistamiseen. Sillä on mahdollista luoda erittäin nopeasti sovellus erilaisia kirjastoja hyväksikäyttäen. On myös helppo kuvitella, että Node.js-kehitysalustan callback-toiminnallisuus tuo omia haasteita laajoissa applikaatioissa verrattuna perinteisempiin palvelinratkaisuihin esimerkiksi SpringBoot ja ASP.NET.

LÄHTEET

Bechtol, E. 2020. Node Service-oriented Architecture. Luettu 24.3.2022. https://www.codementor.io/@evanbechtol/node-service-oriented-architecture-12vjt9zs9i

Cleary, C. 2022. Why should you separate Controllers from Services in Node REST API's? Luettu 24.3.2022. https://www.coreycleary.me/why-should-you-separate-controllers-from-services-in-node-rest-apis

DEV. 2022. Introduction to Object-relational mapping: the what, why, when and how of ORM. Luettu 24.3.2022 https://dev.to/tinazhouhui/introduction-to-object-relational-mapping-the-what-why-when-and-how-of-orm-nb2

Express. 2017. Writing middleware for use in Express apps. Luettu 24.3.2022. https://expressjs.com/en/guide/writing-middleware.html

GeeksforGeeks, 2021. Introduction to TypeScript. Luettu 24.3.2022. https://www.geeksforgeeks.org/introduction-to-typescript/

Hammink, J. Poso, A. 2020. An introduction to PostgreSQL. Luettu 24.3.2022. https://aiven.io/blog/an-introduction-to-postgresql

Helsingin yliopisto. 2009. Ohjelmistojen mallintaminen, arkkitehtuuria ja rajapintoja. Luettu 24.3.2022. https://www.cs.helsinki.fi/u/pohjalai/ke09/ohma/slides/ohma_08-arkkitehtuuri-suunnittelua.pdf

Jansen, G. 2012. Thoughts on RESTful API Design. Luettu 24.3.2022. https://restful-api-design.readthedocs.io/en/latest/resources.html

JWT n.d. Introduction to JSON Web Tokens. Luettu 24.3.2022. https://jwt.io/introduction

KeyCDN. 2017. The Pros and Cons of 8 Popular Databases. Luettu 24.3.2022. https://www.keycdn.com/blog/popular-databases

Khanna, M. 2021. A Guide to Error Handling in Express.js. Luettu 24.3.2022 https://scoutapm.com/blog/express-error-handling

Malav, B. 2018, API Design for Microservices. Luettu 24.3.2022. https://medium.com/startlovingyourself/api-design-for-microservices-a4881036b05e

Mozilla. 2022a. Express/Node introduction. Luettu 24.3.2022. https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction

Mozilla. 2022b. Express Tutorial Part 4: Routes and controllers. Luettu 24.3.2022.
https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/routes

OpenJS Foundation. n.d.a. Introduction to Node.js. Luettu 24.3.2022.
https://nodejs.dev/learn

OpenJS Foundation. n.d.b. The Node.js Event Loop, Timers, and process.nextTick(). Luettu 24.3.2022.
https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/

REST API Tutorial. n.d.a. Guiding Principles of REST. Luettu 24.3.2022.
https://restfulapi.net/

REST API Tutorial. n.d.b. Statelessness in REST APIs. Luettu 24.3.2022
https://restfulapi.net/statelessness/

Smallcombe, M. 2021. SQL vs NoSQL: 5 critical Differences. Luettu 24.3.2022.
https://www.integrate.io/blog/the-sql-vs-nosql-difference/

Subramanian, V. 2019. Pro MERN Stack: Full Stack Web App Development with Mongo, Express, React and Node. Kappale 5. Express and GraphQL.

TypeORM, n.d. TypeORM. Luettu 24.3.2022
https://typeorm.io/#/

Wilson, D. n.d. How to do stuff RESTful. Luettu 24.3.2022.
https://restcookbook.com/Basics/caching/

Wirantono, M. 2020. LocalStorage vs Cookies: All You Need To Know About Storing JWT Tokens Securely in The Front-End. Luettu 24.3.2022
https://dev.to/cotter/localstorage-vs-cookies-all-you-need-to-know-about-storing-jwt-tokens-securely-in-the-front-end-15id