Tampereen ammattikorkeakoulu

Funktionaalinen ohjelmointi ja olio-ohjelmointi TypeScriptissä

Antti-Pekka Haarala

OPINNÄYTETYÖ
Maaliskuu 2022

Tietotekniikan tutkinto-ohjelma
Ohjelmistotekniikka

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietotekniikan tutkinto-ohjelma
Ohjelmistotekniikka

HAARALA, ANTTI-PEKKA:
Funktionaalinen ohjelmointi ja olio-ohjelmointi TypeScriptissä

Opinnäytetyö 35 sivua, joista liitteitä 6 sivua
Maaliskuu 2022

Opinnäytetyössä esiteltiin TypeScript-ohjelmointikieltä, sen eri ominaisuuksia ja kuinka sitä voidaan suorittaa backend-kehityksessä. Opinnäytetyön tarkoituksena oli tutustua, tutkia ja verrata funktionaalista ohjelmointia ja olio-ohjelmointia TypeScript-ohjelmointikielessä sekä ohjelmointirajapinnassa. Opinnäytetyön tavoitteena oli havainnoida ja selventää, kuinka ohjelmointirajapintaa, joka oli toteutettu TypeScript-ohjelmointikielellä ja joka kommunikoi MongoDB-tietokannan kanssa, voidaan hyödyntää olio-ohjelmoinnissa ja funktionaalisessa ohjelmoinnissa. Opinnäytetyön alussa työn tekijällä oli aikaisempaa tietämystä TypeScript-ohjelmointikielestä, ohjelmointirajapinnoista ja olio-ohjelmoinnista, mutta ei juuri tietämystä funktionaalisesta ohjelmoinnista ja sen soveltamisesta ohjelmointirajapinnassa.

Toteutetuista ohjelmointirajapinnoista olio-ohjelmoinnin konsepteja noudattava ohjelmointirajapinta on parempi ratkaisu verrattuna funktionaalisen ohjelmoinnin konsepteja noudattavaan ohjelmointirajapintaan. Vertailukohteena on komponenttien uudelleenkäytettävyys, niiden luettavuus, helppokäyttöisyys ja tekijän oma mielipide.

Erilaisten ohjelmointiparadigmojen soveltaminen ohjelmointirajapinnassa vaatii enemmän tutkimusta ja olemassa olevaa tietoa. Eräs johtopäätös opinnäytetyöstä oli se, että funktionaalisesta ohjelmoinnista ei ole tarpeeksi tieteellistä tietoa saatavilla. Opinnäytetyössä esitettyjä tuloksia voidaan tutkia ja laajentaa monipuolisesti, esimerkiksi funktionaalisen ohjelmoinnin soveltamista ohjelmointirajapintaan sekä laajentaa, mihin ohjelmointirajapintaan tietty paradigma soveltuu parhaiten.

Asiasanat: funktionaalinen ohjelmointi, olio-ohjelmointi, TypeScript, ohjelmointirajapinta

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in ICT Engineering
Software Engineering

HAARALA, ANTTI-PEKKA:
Functional Programming and Object-Oriented Programming in TypeScript

Bachelor's thesis 35 pages, appendices 6 pages
March 2022

This bachelor’s thesis discusses TypeScript programming language, its various features and how it can be performed in backend development. The purpose of this thesis was to examine and compare functional programming and object-oriented programming in the TypeScript programming language, as well as in the application programming interface. The aim of this thesis was to observe and clarify how an application programming interface, which was written with TypeScript and communicates with MongoDB database, can be done with functional and object-oriented programming. At the beginning of the thesis, the author had previous knowledge of TypeScript, application programming interfaces and object-oriented programming, but not much knowledge of functional programming and its implementation in application programming interface.

An application programming interface that follows the concepts of object-oriented programming is a better solution compared to an application programming interface that follows the concepts of functional programming. Reusability of components, readability, ease of use and author’s own opinion were used as a reference.

Applying different programming paradigms in the application programming interface requires more research and existing knowledge. One conclusion in the thesis was that there is not enough scientific information available on functional programming. The results presented in this thesis can be studied and expanded in various ways, for example the implementation of functional programming to the application programming interface and which programming paradigm is the most suitable for which application programming interface.

Key words: Functional programming, object-oriented programming, TypeScript, Application Programming Interface

SISÄLLYS

ERITYISSANASTO

API	Rajapinta, jonka avulla eri komponentit keskustelevat.
Konekieli	Tietokoneen suorittimen kieli.
JavaScript-moottori	Alusta JavaScript-ohjelmointikielen suorittamiseen.
API-päätepiste	Kommunikointikanava, joka tarjoaa pääsyn ohjelmointirajapintaan.
HTTP-protokolla	Protokolla, jota käytetään tiedonsiirtoon.
JSON	Tiedonvälityksessä käytettävä tiedostonmuoto.
Ad hoc	Kysely, jossa tiedetään haluttu tieto vasta ajon aikana.
URL	Verkkosivun osoite.
Olio	Kokoelma tietoa ja toiminnallisuutta.
Funktio	Uudelleen kutsuttava moduuli, joka toteuttaa haluttua logiikkaa.
Paradigma	Menetelmä rakentaa ohjelmaa.
Lähdekoodi	Ohjelman kuvaus tekstimuodossa.
Typechecker	TypeScript-ohjelmointikielen muuttujien tyypin ajonaikainen tarkistaja.
TSC	TypeScript-ohjelmointikielen kääntäjä.
Callback	Suoritettava funktio, joka odottaa toisen funktion suoritusta.
Header	HTTP-kutsun otsikko, joka sisältää lisätietoa kutsusta.
Body	HTTP-kutsun runko, joka sisältää tietoa erilaisessa muodossa.

1 JOHDANTO

TypeScript on Microsoftin kehittämä supersarja JavaScript-ohjelmointikielestä, mikä tuo mukanaan staattista tyypitystä, luokkia, sekä muita ominaisuuksia (Moiseev & Fain 2020). JavaScript-ohjelmointikieltä voi suorittaa Node.js-ajoympäristössä. Node.js-ajoympäristö suorittaa ohjelmointikieliä selaimen ulkopuolella itsenäisessä prosessissa. (Mead 2018, 10–11.) TypeScript-ohjelmointikielellä voi luoda erilaisia ohjelmia, kuten ohjelmointirajapintoja.

API, eli ohjelmointirajapinta on ohjelmointiliitäntä, jonka avulla erilaiset sovellukset pystyvät välittämään tietoa keskenään (Visma n.d.). Ohjelmointirajapinnan avulla voidaan kiteyttää kompleksisia järjestelmiä, tietokantoja ja toimintoja yksinkertaiseen muotoon, minkä vuoksi ohjelmointirajapinnat eivät tarvitse paljon aikaisempaa tietämystä tai kokemusta. (Preibisch 2018.) Ohjelmointirajapinnan toteuttamisessa voi noudattaa ohjelmointiparadigmaa, kuten funktionaalista ohjelmointia tai olio-ohjelmointia.

Ohjelmointiparadigma on lähdekoodin kirjoittamisen tyyliä ja järjestämistä tiettyyn muotoon (Learn Computer Science n.d.). Olio-ohjelmointi perustuu luokkiin, olioihin ja niiden toimintaan (Doherty 2020). Funktionaalinen ohjelmointi keskittyy funktioihin, niihin liittyviin konsepteihin ja erilaisiin säännöksiin (Remo 2019).

Tässä opinnäytetyössä toteutetaan kaksi erilaista ohjelmointirajapintaa funktionaalisella ohjelmoinnilla ja olio-ohjelmoinnilla, jotka toteutetaan TypeScript-ohjelmointikielellä. Ohjelmointirajapinta kommunikoi MongoDB-tietokannan kanssa. Opinnäytetyö kattaa käytetyt teknologiat ja konseptit, jonka jälkeen esitetään toteutus ja johtopäätös.

Tämän opinnäytetyön tavoitteena on tutkia TypeScript-ohjelmointikieltä, ja kuinka erilaisia ohjelmointiparadigmoja hyödynnetään TypeScript-ohjelmointikielellä toteutettuun ohjelmointirajapintaan. Lisäksi tavoitteena on verrata ja tutkia, kumpi soveltuu paremmin ohjelmointirajapinnan toteuttamiseen.

2 TYPESCRIPT JA AJOYMPÄRISTÖ

TypeScript on JavaScriptiin perustuva ohjelmointikieli, jolla voidaan toteuttaa frontend- ja backend-kehitystä. TypeScript täytyy kääntää JavaScriptiksi, jotta sitä voidaan suorittaa selaimessa tai itsenäisessä JavaScript-moottorissa. Microsoft on kehittänyt TypeScript-ohjelmointikielen ja se on julkaistu avoimena lähdekoodina vuonna 2012. (Moiseev & Fain 2020.)

2.1 JavaScript

JavaScript on Netscape Communications Corporationin ja Sun Microsystemsin luoma ohjelmointikieli, jonka tarkoitus oli tuoda toiminnallisuutta verkkosivustoihin. Netscape Communications Corporation, joka toimii nykyään nimellä Mozilla, toimitti JavaScript-ohjelmointikielen standardoitavaksi European Computer Manufacturer's Association -standardointiorganisaatiolle, mistä seurasi standardoitu versio JavaScriptistä, nimeltä ECMAscript. Termillä JavaScript yleensä viitataan ECMAscript standardin mukaiseen JavaScript-ohjelmointikieleen. (Flanagan 2020; Pollock 2019.)

JavaScript on prototyyppipohjainen, suorituksen aikana tulkittu verkkoympäristössä käytettävä ohjelmointikieli, jota enemmistö olemassa olevista verkkosivustoista hyödyntää. Lähes kaikki modernit selaimet sisältävät tulkin JavaScriptin suorittamista varten. Prototyyppipohjainen tarkoittaa sitä, että JavaScript on olio-ohjelmointiin perustuva ohjelmointikieli, mikä sallii erilaisten objektien luomisen ja käyttämisen. JavaScript soveltuu myös funktionaalisen ohjelmointiin erittäin hyvin. (Flanagan 2020; Pollock 2019.)

ECMAscript-standardia on päivitetty vuosittain vuodesta 2015 lähtien ja jokainen päivitys tuo mukanaan korjauksia tai uusia ominaisuuksia JavaScript-ohjelmointikieleen. ECMAscript 5 (tai niin sanottu ES5) antoi kehittäjälle mahdollisuuden poistaa käytöstä vanhaa ja virheellistä JavaScript-logiikkaa. ES6 toi mukanaan suurimmat muutokset JavaScriptiin, kuten luokat ja erilliset moduulit. (Flanagan 2020.)

JavaScript tukee dynaamista tyypitystä, jossa dynaaminen tyypitys tarkoittaa muuttujan tietotyypin muuttamista toiseen ohjelman suorituksen aikana. Tämä voi johtaa erilaisiin ongelmiin komplekseissa funktioissa, joissa vaaditaan tietty tietotyyppi jotakin toimintoa varten. JavaScript ei varoita kehittäjää tietotyypin muuntamiseen liittyvistä ongelmista. (Moiseev & Fain 2020.)

2.2 TypeScript

TypeScript on supersarja JavaScriptistä, eli se sisältää kaikki JavaScriptin ominaisuudet uusien ominaisuuksien lisäksi. Tämä tarkoittaa sitä, että JavaScript-koodi on validia TypeScript-lähdekoodia, mutta TypeScript-lähdekoodi ei ole validia JavaScript-lähdekoodia, jos kyseinen koodi käyttää TypeScriptin uusia ominaisuuksia. (Moiseev & Fain 2020.)

Sovellus, joka on kirjoitettu TypeScriptillä, täytyy kääntää JavaScriptiksi ensin, jonka jälkeen se voidaan suorittaa selaimessa tai JavaScript-moottorissa. Kääntäminen tarkoittaa ohjelman lähdekoodin ohjelmointikielen muuttamista toiseen ohjelmointikieleen. (Moiseev & Fain 2020.)

2.2.1 Tyypitys

TypeScriptin tärkein ominaisuus on staattinen tyypitys, jossa tyypitys tarkoittaa muuttujan, funktion tai olion tietotyyppiä. Staattinen tyypitys ei salli muuttujan tietotyypin muuttamista, kun taas dynaaminen tyypitys sallii muuttamisen tietotyypistä toiseen. TypeScript tarjoaa tämän lisäksi sekä eksplisiittistä että implisiittistä tyypitystä. (Cherny 2019; Moiseev & Fain 2020.)

Eksplisiittinen tyypitys on manuaalinen tyypitys, jossa ohjelmiston tekijä asettaa tyyppejä muuttujiin, olioihin tai funktioille. Ensiksi täytyy luoda muuttuja, jonka jälkeen asetetaan haluttu tyyppi kaksoispisteen jälkeen. Tämän jälkeen TypeScript tulkitsee muuttujan tietotyypin asetetuksi tietotyypiksi, kuten kuvassa 1 esitetään. (Cherny 2019.)

```
// numero
let yksi: number = 1;
// merkkijono
let kaksi: string = 'kaksi muuttuja';
// totuusarvomuuttuja
let kolme: boolean = true;
// numerotaulukko
let nelja: number[] = [4, 5];
```

KUVA 1. Eksplisiittinen tyypitys TypeScriptissä

Implisiittinen tyypitys on TypeScriptin automaattinen, muuttujan tyypin tunnistaminen Type Systemin avulla. Tämän jälkeen muuttujan tietotyyppiä ei voida muuttaa toiseen tietotyyppiin, kuten kuvassa 2 esitetään. Type System on kokoelma sääntöjä, joita Typechecker käyttää tyyppien asettamisessa. (Cherny 2019.)

```
// numero
let yksi = 1;
// merkkijono
let kaksi = 'kaksi muuttuja';
// totuusarvomuuttuja
let kolme = true;
// numerotaulukko
let nelja = [4, 5];

// Virhe: numero ei voi olla string
yksi = 'yksi muuttuja';
```

KUVA 2. Implisiittinen tyypitys TypeScriptissä

TypeScript sisältää erilaisia valmiita perustyyppejä, joita voidaan hyödyntää tiedon hallitsemisessa (Moiseev & Fain 2020). Moiseev ja Fain (2020) esittävät TypeScriptin tarjoamat perustyypit:

- string—For textual data
- boolean—For true/false values
- number—For numeric values

- symbol—A unique value created by calling the Symbol constructor
- any—For variables that can hold values of various types, which may be unknown when you're writing the code
- unknown—A counterpart of any, but no operations are permitted on an unknown without first asserting or narrowing it to a more specific type
- never—For representing unreachable code (we'll provide an example shortly)
- void—An absence of a value

2.2.2 Type-avainsana ja rajapintaluokat

TypeScriptissä voi luoda mukautetun tyypin tai objektityypin type-avainsanalla. Type-avainsanaa voi hyödyntää yksittäisen muuttujan tietotyypin määrittelyssä, kuin myös objektin tietotyypin määrittelyssä. Mukautetun tyypin tai objektityypin voi luoda osoittamalla tyyppiä johonkin haluttuun tietotyyppiin. Mukautetun objektityypin kentästä voi tehdä vaihtoehtoisen asettamalla kenttään kysymysmerkin. (kuva 3; Cherny 2019; Moiseev & Fain 2020.)

```
// Määritellään mukautettu tyyppi
type Nimi = string;

// Määritellään mukautettu objektityyppi
type Henkilo = {
    nimi: Nimi;
    ika?: number;
};
```

KUVA 3. Type-avainsana

TypeScriptissä on interface-syntaksi, jota käytetään pakottamaan tiettyjä muuttujia tai metodeja jollekin objektille. Interface eli rajapintaluokka toimii samalla tavalla kuin type-avainsana, lukuun ottamatta pieniä eroavaisuuksia. Rajapintaluokka määritellään kuvan 4 mukaisella tavalla. (Moiseev & Fain 2020.)

```
// Määritellään rajapintaluokka
interface Henkilo {
    nimi: string;
    ika: number;
};

// Määritellään objekti rajapintaluokalla
const henkilo: Henkilo = {
    nimi: 'test',
    ika: 19
}
```

KUVA 4. Interface-avainsana

2.2.3 Luokat TypeScriptissä

TypeScript tukee luokkia, joiden avulla voidaan luoda erilaisia olioita, joilla on kyseisessä luokassa määriteltyjä metodeja ja attribuutteja (kuva 5; Moiseev & Fain 2020). Luokat määritellään class-avainsanalla ja niitä voi periyttää toisesta luokasta extends-avainsanalla. Metodit ja attribuutit voivat olla suojattuja eri avainsanoilla, joita ovat public, private ja protected. (Cherny 2019.) Kun luokista muodostetaan olioita, luokka kutsuu rakentajaa (kuva 5; Moiseev & Fain 2020).

TypeScriptin luokkiin voidaan implementoida rajapintaluokkia, jotka määrittelevät luokkien metodien ja attribuuttien tietotyypit. Luokan, joka implementoi rajapintaluokkaa, täytyy sisältää rajapintaluokassa pakolliseksi määritellyt ominaisuudet. Rajanpintaluokat eivät voi määritellä metodien implementaatioita, ainoastaan metodien parametreja ja palautusarvoja. (kuva 5; Moiseev & Fain 2020.)

```
// Määritellään rajapintaluokka
interface HenkiloInterface {
    nimi: string;
    ika: number;
    tulostaTiedot(): void;
};

// Määritellään luokka hyödyntäen rajapintaluokkaa
class Henkilo implements HenkiloInterface {
    nimi: string;
    ika: number;

    constructor(nimi: string, ika: number) {
        this.nimi = nimi;
        this.ika = ika;
    }

    tulostaTiedot(): void {
        console.log(`Nimi: ${this.nimi}\n Ika: ${this.ika}\n`);
    }
};
```

KUVA 5. Rajapintaluokkaa implementoiva Henkilo-luokka

2.2.4 Virheentarkistus ja kääntäminen

TypeScript tarkistaa ohjelman lähdekoodia staattisesti mahdollisten virheiden varalta, jotka aiheuttaisivat ajonaikaisia virheitä. Staattinen koodianalyysi ei ota huomioon ajon aikana tapahtuvia virheitä, kuten ohjelman käyttäjän virheellisiä syötteitä, puskurin ylivuotovirheitä tai virheellisiä verkkoyhteyksiä. (Cherny 2019.) Staattinen virheentarkistus varoittaa ohjelmistokehittäjää syntaksiin ja tyypityksiin liittyvistä ongelmista ennen ohjelman suorittamista (Moiseev & Fain 2020; Cherny 2019).

TypeScript-lähdekoodin kääntäminen JavaScriptiksi tapahtuu kääntäjän (tsc) avulla, jossa kääntämistä voidaan konfiguroida erilaisilla asetuksilla (Moiseev & Fain 2020). Kääntäjän asetuksia on mahdollista muokata luomalla tiedosto (tsconfig.json), johon voidaan listata erilaisia vaatimuksia käännökselle. Kuvassa 6 on esimerkkejä asetuksista, jotka vaikuttavat käännökseen. Asetus baseUrl määrittää käännettävien tiedostojen sijainnin, outDir käännettyjen tiedostojen sijainnin, noEmitOnError estää käännöksen, jos käännöksessä on virheitä ja target valitsee käännöksen ECMAscript-version. (kuva 6; Moiseev & Fain 2020.)

```
{
  "compilerOptions": {
      "baseUrl": "src",
      "outDir": "./dist",
      "noEmitOnError": true,
      "target": "es5"
  }
}
```

KUVA 6. TypeScript asetuksia (Moiseev & Fain 2020 muokattu)

2.3 Node.js-ajoympäristö

Node on JavaScriptistä kehitetty ajoympäristö, jonka avulla voidaan suorittaa JavaScriptiä itsenäisenä prosessina selaimen ulkopuolella. JavaScriptillä oli ennen rajoitettu toiminallisuus, joka salli selaimessa tapahtuvia toimintoja, kuten URL-päivityksiä ja sivuston ulkonäön muokkausta. JavaScriptiä ei pystytty hyödyntämään tämän enempää, mutta Node toi mukanaan uusia ominaisuuksia, jotka ovat samankaltaisia muiden ohjelmointikielien kanssa. (Mead 2018, 10–11.)

Node.js:n avulla voidaan luoda ohjelmia, jotka käyttävät JavaScript-syntaksia, muokkaamaan käyttöjärjestelmän tiedostojärjestelmää, kutsumaan ja luomaan komentoja tietokannalle suoraan ja luoda palvelimia. Kyseiset toiminnot eivät olleet ennen mahdollisia JavaScriptillä, mutta ovat nykyään Node:n avulla. (Mead 2018, 11.)

Node ja JavaScript hyödyntävät samaa JavaScript-moottoria, joka on nimeltään V8 JavaScript Runtime Engine. Se on C++ ohjelmointikielellä kirjoitettu avoimen lähdekoodin moottori, joka kääntää JavaScript-lähdekoodia paljon nopeammaksi konekieleksi, jonka vuoksi Node-ohjelmat ovat yleensä todella nopeita. (Mead 2018, 11.)

2.3.1 NPM

NPM on Node.js-pakettien hallintaan liittyvä työkalu, jossa on maailman laajin ekosysteemi avoimen lähdekoodin kirjastoja (Mead 2018, 33). Node-ohjelmiin voidaan lisätä paketinhallintajärjestelmän (NPM) avulla kolmannen osapuolen paketteja, jotka voivat ratkaista yleisiä ongelmia Node-ohjelmissa. Kolmannen osapuolen paketit ovat yleensä testattuja ja todistettuja toimivaksi, sekä ne ovat dokumentoitu. Pakettien hyödyntäminen säästää myös runsaasti aikaa. (Mead 2018, 97.)

NPM-paketti koostuu yhdestä tai useammasta tiedostosta tai kansiosta, jossa on mukana tiedosto nimeltä package.json. Se kuvaa pakettia NPM-rekisterille. NPM-rekisteri on tietokanta, joka koostuu JavaScript-paketeista. Suurin osa näistä paketeista ovat Node-moduuleita tai ne sisältävät Node-moduuleita, jossa moduuli on joko tiedosto tai kansio, joka voidaan käyttöönottaa Node-ohjelmassa. (NPM. n.d.)

2.3.2 Yksisäikeinen tapahtumasilmukka

Node.js on yksisäikeinen, esteetön ja tapahtumapohjainen ajoympäristö, jossa on mahdollista suorittaa lähdekoodia asynkronisesti (Mead 2018, 256). Tämä tarkoittaa sitä, että Node.js palvelin pystyy vaihtamaan kutsusta toiseen tarvittaessa (Mead 2018, 259–260). Kuvassa 7 on esimerkki, kuinka funktio setTimeout tulostaa Rivi 2 myöhemmin verrattuna muihin tulostuksiin.

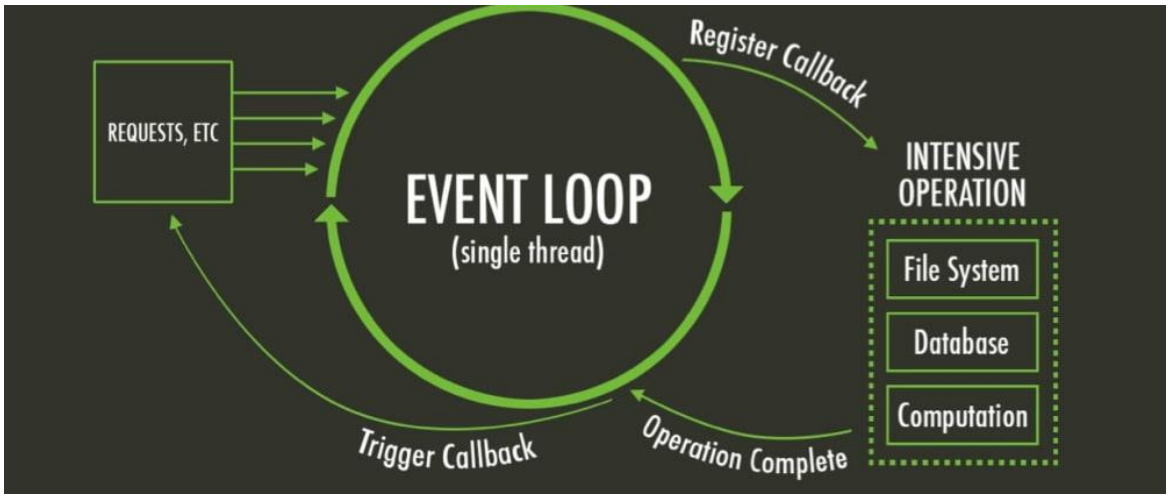
```
console.log('Rivi 1');

setTimeout(() => {
    console.log('Rivi 2');
}, 2000);

console.log('Rivi 3');
```

```
Rivi 1
Rivi 3
Rivi 2
```

KUVA 7. Asynkroninen tulostus

Tapahtumasilmukka (Event Loop) käsittelee asynkronista toteutusta Node-ohjelmassa seuraamalla kutsupinoa (Call Stack) ja callback-jonoa (Mead 2018, 274; Santos 2019). Kutsupino on yksinkertainen tietorakenne, joka seuraa suorituksessa olevia funktioita ja lauseita, kun taas callback-jono on jono callback-funktioita (Mead 2018, 263; Santos 2019).

Kuviossa 1 on esitetty, kuinka tapahtumasilmukka rekisteröi käyttäjän pyynnön callback-funktioksi. Callback-funktion ollessaan valmis, funktio siirtyy callback-jonoon, jossa kyseinen funktio odottaa tapahtumasilmukkaa. Kun kutsupino on tyhjä, tapahtumasilmukka siirtää callback-funktion kutsupinoon ja palauttaa käyttäjälle vastauksen. (Mead 2018, 275–277; Santos 2019.)



KUVIO 1. Node.js tapahtumasilmukka (Santos 2019)

3 OHJELMOINTIRAJAPINTA JA TIETOKANTA

Ohjelmointirajapinta, eli application programming interface (API) on ohjelmointiliitäntä, jonka avulla eri sovellukset voivat siirtää tietoa keskenään (Visma n.d.; Preibisch 2018). Ohjelmointirajapinnassa on yksinkertainen päätepiste kompleksiin järjestelmään, joka voi tarkoittaa eri kontekstissa esimerkiksi myyntiautomaattia, jossa käyttäjä valitsee haluamansa tuotteen ja automaatti antaa sen maksua vastaan (Preibisch 2018).

Ohjelmointirajapintaa voidaan soveltaa sellaisissa ohjelmissa, joissa tarvitaan ulkoista tietoa. Kuvassa 8 esitetään, kuinka ohjelmat voivat olla joko yksinkertaisia tai monimutkaisia, missä tarvitaan tietynlaista dataa. Ohjelma voi kutsua tietyillä parametreillä ulkoista ohjelmointirajapintaa HTTP-protokollan avulla. Tämän jälkeen ohjelmointirajapinta prosessoi kutsun, hakee tarvittaessa tietoa tietokannasta ja vastaa sitten ohjelmalle. (Preibisch 2018.)

KUVA 8. Yksinkertaisia laitteita ja ohjelmia (Preibisch 2018)

3.1 HTTP-protokolla

HTTP-protokolla on menetelmä, joiden avulla voidaan siirtää dataa käyttäjän ja palvelimen välillä. HTTP-viesti voi olla joko request tai response, jossa request on käyttäjän lähettämä pyyntö palvelimelle ja response on palvelimen vastaus. (Mozilla 2021.) Postman (2022) on luetellut seuraavat metodit:

- GET retrieves data from an API.
- POST sends new data to an API.
- PATCH and PUT update existing data.
- DELETE removes existing data.

Käyttäjän pyyntö, eli request sisältää metodin, osoitteen, HTTP-version, header ja vaihtoehtoisesti body:n. Palvelimen tai ohjelmointirajapinnan vastaus, eli response sisältää HTTP-version, vaihtoehtoisesti body:n, header, tilakoodin ja siihen kuuluvan selitteen, jotka kertovat käyttäjän pyynnön tilan. (Mozilla 2021.)

3.2 MongoDB-tietokanta

MongoDB on MongoDB Inc:n kehittämä horisontaalisesti laajeneva ilmainen tietokantaohjelmisto, jonka avulla voidaan tehokkaasti tallentaa ja hakea erilaista informaatiota. MongoDB tukee ad hoc -tyyppisiä kyselyitä, indeksointia ja informaation yhdistämistä reaaliajassa. MongoDB tukee useita eri ohjelmointikieliä olemassa olevien ajureiden avulla, kuten esimerkiksi JavaScriptiä MongoDB-paketin avulla. (MongoDB 2021.)

MongoDB tallentaa informaatiota dokumentteihin, jotka muistuttavat JSON-tiedostomuotoa. Tietokannassa olevat dokumentit voivat olla muodoltaan erilaisia ja vaihtaa informaation muotoa myöhemmin ajon aikana. MongoDB-dokumentin pystyy liittämään ohjelmassa luotuun olioon, joka helpottaa informaation kanssa työskentelyä. (kuva 9; MongoDB 2021.)

```
{
  _id: "5cf0029caff5056591b0ce7d",
  firstname: 'Jane',
  lastname: 'Wu',
  address: {
    street: '1 Circle Rd',
    city: 'Los Angeles',
    state: 'CA',
    zip: '90404'
  }
}
```

KUVA 9. MongoDB dokumentti (MongoDB 2021, muokattu)

4 OHJELMOINTIPARADIGMA

Ohjelmointikieli on ongelmanratkaisutyökalu, jonka avulla voidaan ratkaista mahdollisia ongelmia erilaisilla ohjelmointiparadigmoilla. Ohjelmointiparadigma on lähdekoodin tyyliä ja järjestämistä tiettyyn malliin, jossa jokainen paradigma puoltaa erilaista lähestymistapaa. Paradigma tarkoittaa tiettyä määrää suunnitteluperiaatteita, jotka määrittelevät suunnitellun ohjelman lähdekoodin rakenteen. Jokaisessa ohjelmistoparadigmassa on erilaiset hyödyt ja haitat. Käyttökohteet voivat kuitenkin erota toisistaan. (Learn Computer Science n.d.)

4.1 Funktionaalinen ohjelmointi

Funktionaalinen ohjelmointi on ohjelmointiparadigma, jossa käytetään funktioita ohjelman rakennuspalikoina (Remo 2019). Funktionaalinen ohjelmointi perustuu deklaratiiviseen ohjelmointiin, missä keskitytään haluttuun lopputulokseen ratkaisemalla ongelmia matemaattisilla funktioilla. Funktionaalinen ohjelmointi soveltuu moniytimisiin ja monisäikeisiin ympäristöihin. (Learn Computer Science n.d.)

Funktionaalisessa ohjelmoinnissa on sääntöjä, jotka määrittelevät funktioiden rakennetta. Funktion täytyy palauttaa argumenteista riippuva arvo eli arvo ei saa muuttua, jos argumentit pysyvät samoina, kuten kuvassa 10 esitetään. Funktionaalisessa ohjelmoinnissa asetetun muuttujan arvo on aina sama, eli muuttuja on vakio. Toteutetut funktiot voivat olla korkean asteen funktioita, eli ne hyödyntävät toisia funktioita joko argumenttina tai palautusarvona. (Learn Computer Science n.d.; Remo 2019.) Funktiot ovat toisistaan riippumattomia, eli funktionaalisessa ohjelmoinnissa ei ole tilaa (Remo 2019).

```
// Puhdas funktio, joka tarkistaa argumentin sisällön
// ja palauttaa totuusarvon, joka riippuu argumentista
function onkoAutaSivu(path: string): boolean {
    // Tarkistetaan onko argumentti /auta sivusto
    if (path === '/auta') {
        // Palautetaan totuusarvo kyllä
        return true;
    }

    // Jos argumentti ei ole /auta sivusto
    // palautetaan totuusarvo ei
    return false;
}
```

KUVA 10. Puhdas funktio

4.2 Olio-ohjelmointi

Olio-ohjelmointi on rakenteellista ohjelmointia, jossa ohjelman komponentit ovat olioita (Learn Computer Science n.d.). Kuvassa 11 on esitetty, kuinka olio-ohjelmoinnin komponentit, eli oliot, luodaan erilaisten luokkien avulla. Luokkaa voidaan käyttää uudelleen lähdekoodissa uusien olioiden luomisessa. Luokka on olion rakenne, joka sisältää attribuutteja ja metodeja. Luokat voivat hyödyntää toisia luokkia periytymisellä. (Doherty 2020; Cherny 2019.)

```
class Henkilo {
    nimi: string;

    // Constructor, joka kutsutaan oliota luodessa
    constructor(nimi: string) {
        this.nimi = nimi;
    }

    // Getter nimi muuttujalle
    getNimi() {
        return this.nimi;
    }

    // Setter nimi muuttujalle
    setNimi(nimi: string) {
        this.nimi = nimi;
    }

    // Metodi, joka sanoo nimen
    sanoNimi() {
        console.log(`Nimi: ${this.nimi}`);
    }
}

const newHenkilo = new Henkilo('Antti');

// Tulostaa Nimi: Antti
newHenkilo.sanoNimi();
```

KUVA 11. Henkilo luokka ja olio

Olio-ohjelmointi sisältää 4 erilaista konseptia, jotka ovat perintö, kapselointi, abstraktio ja polymorfismi. Perinnöllisyydellä tarkoitetaan luokkaa, joka on perinyt attribuutteja ja metodeja toiselta luokalta. Kapselointi on tiedon sitomista olioon, jossa altistetaan ainoastaan valittua tietoa. Abstraktio on ainoastaan tarvittavien metodien altistamista. Polymorfismilla tarkoitetaan sitä, että useat eri metodit voivat tehdä saman asian. (Doherty 2020.)

5 OHJELMOINTIPARADIGMOJEN VERTAILU TYPESCRIPTISSÄ

Tässä opinnäytetyössä toteutetaan kaksi erilaista ohjelmointirajapintaa, jotka kommunikoivat tietokannan kanssa. Ohjelmointirajapinnat toteutetaan TypeScript-ohjelmointikielellä. Kyseisiä ohjelmointirajapintoja voi kutsua HTTP-kutsujen avulla. Toteutetut ohjelmointirajapinnat esitetään, kuinka ne toimivat ja kuinka ne hyödyntävät erilaisia konsepteja.

Toteutettuja ohjelmointirajapintoja verrataan esittämällä kuinka hyvin luotuja komponentteja voi uudelleen käyttää. Miten luettava ja helppokäyttöinen toteutetun ohjelman lähdekoodi on ja kumpi toteutuksista on sopivampi vaihtoehto ohjelmointirajapintaan, joka hyödyntää tietokantaa.

5.1 Ohjelmointiparadigmojen soveltaminen ohjelmointirajapinnassa

Toteutetut ohjelmointirajapinnat hyödyntävät TypeScript-ohjelmointikieltä, Node.js-ajoympäristöä ja NPM-pakettia nimeltä Express, jonka avulla voi toteuttaa erilaisia ohjelmointirajapintoja (OpenJS Foundation n.d.). Ohjelmointirajapinnat käyttävät MongoDB-tietokantaa tiedon hallitsemisessa.

5.1.1 Funktionaalisen ohjelmoinnin toteutus

Funktionaalinen ohjelmointirajapinta hyödyntää erilaisia konsepteja funktionaalisesta paradigmasta, kuten korkean asteen funktioita, muuttujien muuttumattomuutta const-avainsanalla ja callback-funktioita. (liite 1.) Toteutettu ohjelmointirajapinta koostuu useasta tiedostosta ja kansiosta, joilla jokaisella on oma toiminnallisuutensa (kuva 12).

KUVA 12. Kansioiden ja tiedostojen rakenne

Ohjelmassa luodaan Express-prosessi, mikä asetetaan kuuntelemaan haluttua porttia. Tämän jälkeen luodaan ohjelmointirajapinnalle HTTP-metodiin GET vastaava polku, joka suorittaa halutun funktion getAllItems. (kuva 13.)

```
// Luodaan express prosessi
const app = express();
const port = process.env.PORT || 3000;
app.use(express.urlencoded({ extended: true }));
app.use(express.json());

// Asetetaan prosessille portti
app.listen(port, () => {
  console.log(`Running on port ${port}.`);
});

// Päätepiste get
app.get('/items', getAllItems);
```

KUVA 13. Express ja päätepiste (Liite 1)

Funktio getAllItems käsittelee ja vastaa ohjelmointirajapintaan kohdistuviin kutsuihin (liite 1). Funktio getAllItems hyödyntää korkean asteen funktiota getAllOperation, jolle on asetettu parametriksi callback-funktio queryDatabase. Palvelimen vastaus ja statuskoodi riippuvat korkean tason funktion getAllOperation palautusarvosta. (kuva 14.)

```
/**
 * Get päätepiste
 * Hyödyntää higher-order ja callback funktioita
 * Hakee kaikki dokumentit tietokannasta
 */
export async function getAllItems(req: Request, res: Response) {
    try {
        const result = await getAllOperation(queryDatabase);
        result ? res.status(200).send(result) : res.status(404).send('Not found');
    } catch (err) {
        res.status(404).send('Not found');
    }
}
```

KUVA 14. Funktio getAllItems (Liite 1)

Korkean asteen funktio getAllOperation ottaa callback-funktion parametriksi ja palauttaa kaikki dokumentit tietokannasta. Callback-funktiota, nimeltään connect, käytetään tietokantaan yhdistämisessä asettamalla kyseiselle funktiolle yhteys-parametrit, jonka jälkeen yhteys tallennetaan pysyvään muuttujaan. Muuttujaa käytetään dokumenttien hakemiseen MongoDB-funktion find avulla. (kuva 15.)

```
/**
 * Higher-order funktio
 * Hyödyntää queryDatabase funktiota
 * Hakee kaikki dokumentit tietokannasta
 */
export async function getAllOperation(connect: (COLLECTION_NAME: string, DB_CONN_STRING: string, DB_NAME: string) => Promise<mongoDB.Collection>): Promise<Item[]> {
    dotenv.config();
    const collection = await connect(
        process.env.COLLECTION_NAME ? process.env.COLLECTION_NAME : '',
        process.env.DB_CONN_STRING ? process.env.DB_CONN_STRING : '',
        process.env.DB_NAME ? process.env.DB_NAME : '',
    );

    return await collection.find({}).toArray() as Item[];
}
```

KUVA 15. Korkean asteen funktio getAllOperation (Liite 1)

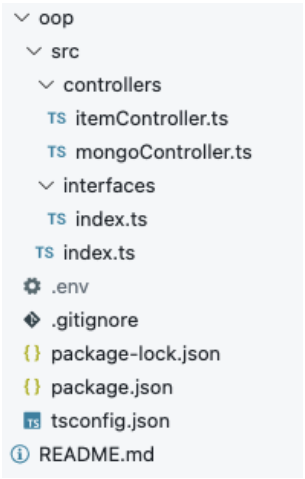
Callback-funktio queryDatabase yhdistää olemassa olevaan tietokantaan parametrien avulla, jonka avulla kyseistä funktiota voi uudelleen käyttää erilaisiin tietokantayhteyksiin. Muodostettu yhteys palautetaan palautusarvona korkean asteen funktiolle. (kuva 16.)

```
/**
 * Callback funktio, joka annetaan Higher-order funktiolle parametrina
 * Luo yhteyden tietokantaan parametrien avulla ja palauttaa yhteyden
 */
export async function queryDatabase(COLLECTION_NAME: string, DB_CONN_STRING: string, DB_NAME: string): Promise<mongoDB.Collection> {
    const collectionName = COLLECTION_NAME;
    const connectionString = DB_CONN_STRING;
    const databaseName = DB_NAME;

    const client: mongoDB.MongoClient = new mongoDB.MongoClient(connectionString);
    await client.connect();
    const db: mongoDB.Db = client.db(databaseName);

    return db.collection(collectionName);
}
```

KUVA 16. Callback funktio queryDatabase (Liite 1)

5.1.2 Olio-ohjelmoinnin toteutus

Olio-ohjelmointiin perustuva ohjelmointirajapinta hyödyntää erilaisia konsepteja, kuten luokkia, luokkien attribuutteja ja metodeja, luotuja olioita ja luokkarajapintojen implementointia. Ohjelma koostuu kahdesta eri luokasta, joita hyödynnetään tiedon hakemisessa. (liite 2.) Kansiorakenne on samanlainen kuin funktionaalisessa toteutuksessa (kuva 17).



KUVA 17. Kansioiden ja tiedostojen rakenne

Ohjelmassa luodaan samanlainen Express-prosessi kuin funktionaalisessa toteutuksessa. Funktion sijasta HTTP-metodilla GET kutsuttu polku toteuttaa globaalisti luodun olion itemController-metodia getAllItems, joka palauttaa vastauksen Express-prosessille (kuva 18).

```
// Luodaan express prosessi
const app = express();
const port = process.env.PORT || 3000;
app.use(express.urlencoded({ extended: true }));
app.use(express.json());

// Asetetaan prosessille portti
app.listen(port, () => {
  console.log(`Running on port ${port}.`);
});

// Luo uusi globaali itemController olio
const itemController = new ItemController();

// Päätepiste get
app.get('/items', itemController.getAllItems);
```

KUVA 18. Express ja päätepiste (Liite 2)

ItemController-luokan metodi getAllItems käsittelee ja vastaa Express-prosessin kutsuihin (liite 2). Metodi hyödyntää luokkaa MongoController, jossa luodaan metodissa asetettujen parametrien avulla yhteys haluttuun tietokantaan. Metodin getAllItems vastaus on riippuvainen luodun olion metodin vastauksesta. (kuva 19.)

```
getAllItems = async (req: Request, res: Response): Promise<void> => {
    try {
        const mongoController = new MongoController(this.connectionString, this.databaseName, this.collectionName);

        const documents: Item[] = await mongoController.fetchAllDocuments() as Item[];
        documents ? res.status(200).send(documents) : res.status(404).send('Not found');
    } catch (error) {
        res.status(500).send('Internal server error');
    }
}
```

KUVA 19. Metodi getAllItems (Liite 2)

MongoController-luokan metodi fetchAllDocuments luo yhteyden tietokantaan, jonka jälkeen tietokannan kokoelmasta haetaan ja palautetaan metodin kutsujalle kaikki saatavilla olevat dokumentit MongoDB find -funktion avulla (kuva 20).

```
async fetchAllDocuments(): Promise<Document[]> {
    await this.client.connect();
    const documents: Document[] = await this.collection.find({}).toArray() as Document[];
    return documents;
}
```

KUVA 20. Metodi fetchAllDocuments (Liite 2)

5.2 Toteutettujen ohjelmien johtopäätökset ja vertailu

Toteutetut ohjelmointirajapinnat hyödyntävät ohjelmointiparadigmojen konsepteja eri tavalla. Funktionaalisessa toteutuksessa hyödynnetään korkean asteen funktioita, muuttujien muuttumattomuutta ja callback-funkioita. Olio-ohjelmointia noudattava ohjelmointirajapinta hyödyntää erilaisia luokkia, rajapintaluokkien implementointia niistä luotuja olioita.

Molemmissa toteutuksissa voidaan hyödyntää toteutettuja komponentteja uudestaan erilaisissa käyttökohteissa. Funktionaalisessa toteutuksessa on callback-funktio queryDatabase, joka hyväksyy yhteystiedot olemassa olevaan tietokantaan parametrien avulla (liite 1). Olio-ohjelmoinnin toteutuksessa hyödynnetään luokkaa MongoController, mikä hyväksyy rakentajassa parametrit olemassa olevaan tietokantaan (liite 2). Tämän takia funktiota queryDatabase ja luokkaa MongoController pystytään käyttämään uudestaan eri tietokantoihin yhdistämisessä (liite 1; liite 2).

Luettavuuden ja helppokäyttöisyyden perusteella olio-ohjelmointi on parempi vaihtoehto näistä, esimerkiksi aloittelevalle ohjelmistokehittäjälle. Olio-ohjelmoinnin konseptit ovat yksinkertaisia verrattuna funktionaaliseen ohjelmointiin. Kyseisessä ohjelmoinnissa on helpompi luoda uudelleen käytettäviä komponentteja ja erotella näiden komponenttien toiminnallisuus toisistaan. Esimerkiksi olio-ohjelmoinnin toteutuksessa ovat kaikki MongoDB:n toiminallisuus yhdessä luokassa, kun taas funktionaalisessa toteutuksessa toiminnallisuus on useassa erilaisessa tiedostossa ja funktiossa (liite 1; liite 2).

Olio-ohjelmointi on mieluisampi vaihtoehto, kun on kyse ohjelmointirajapinnan toteuttamisesta. Olio-ohjelmointi soveltuu enemmän tiedon hallitsemiseen konseptien, kuten luokkien ja olioiden vuoksi. Funktionaalista paradigmaa on vaikea noudattaa ympäristössä, missä on useita irtonaisia komponentteja, kuten tietokantoja. Funktionaalinen paradigma soveltuu ennemmin matemaattisiin ja itsenäisiin sovelluksiin.

6 POHDINTA

Tässä opinnäytetyössä esiteltiin ohjelmointiparadigmojen soveltamista ohjelmointirajapinnassa. Opinnäytetyössä toteutettiin kaksi erilaista ohjelmointirajapintaa, jotka kommunikoivat MongoDB-tietokannan kanssa. Ohjelmointirajapinnat toteutettiin opinnäytetyössä TypeScript ohjelmointikielellä.

Opinnäytetyöprosessi alkoi marraskuussa 2021, jolloin tehtiin Tampereen ammattikorkeakoulun kanssa opinnäytetyösopimus. Tämän jälkeen alkoi teoriatiedon etsiminen erilaisista lähteistä ja sen kirjoittaminen. Opinnäytetyö valmistui aikataulun mukaisesti maaliskuussa 2022. Ennen opinnäytetyötä ei ollut aikaisempaa tietämystä paradigmojen soveltamisesta ohjelmointirajapintaan, jonka vuoksi teoriatiedon soveltaminen tutkielmaan oli haasteellista.

Sovellusten toteuttaminen oli haastava osuus opinnäytetyössä, varsinkin funktionaalisen ohjelmointirajapinnan toteuttaminen. Konseptit ja säännöt funktionaalisessa paradigmassa rajoittivat sovelluksen kehitystä, jonka vuoksi sovellus ei noudata funktionaalista ohjelmointia puhtaasti. Saatavilla oleva tieteellinen tieto funktionaalisesta ohjelmointirajapinnasta oli myös puutteellista tai hyvin vähäistä. Olio-ohjelmointia noudattava sovelluksen kehitys onnistui huomattavasti paremmin oman kokemuksen ja saatavilla olevan tiedon takia. Kuitenkin sain opinnäytetyön ansiosta kasvatettua tietotaitoa kyseisistä aiheista.

Tämän opinnäytetyön tuloksia voidaan laajentaa ja tutkia enemmän, kuten esimerkiksi funktionaalisen ohjelmoinnin soveltamista ohjelmointirajapintaan ja laajentaa minkälaiseen ohjelmointirajapintaan kukin paradigma soveltuu.

LÄHTEET

Cherny, B. 2019. Programming TypeScript: Making Your JavaScript Applications Scale. Boston: O'Reilly Media, Inc. Viitattu 2.2.2022. Vaatii käyttöoikeuden. https://andor.tuni.fi/permalink/358FIN_TAMPO/176jdvt/cdi_askewsholts_vlebooks_9781492037620

Doherty, E. 2020. What is object-oriented programming? OOP explained in depth. Verkkosivu. Viitattu 4.3.2022. https://www.educative.io/blog/object-oriented-programming

Flanagan, D. 2020. JavaScript: The Definitive Guide. 7. painos. Boston: O'Reilly Media, Inc. Viitattu 2.2.2022. Vaatii käyttöoikeuden. https://andor.tuni.fi/permalink/358FIN_TAMPO/176jdvt/cdi_safari_books_v2_9781491952016

Learn Computer Science. n.d. Programming Paradigm. Verkkosivu. Viitattu 4.3.2022. https://www.learncomputerscienceonline.com/programming-paradigm/

Mead, A. 2018. Learning Node.js Development Learn the fundamentals of Node.js, and deploy and test Node.js applications on the web. Birmingham: Packt Publishing. Viitattu 12.2.2022. Vaatii käyttöoikeuden. https://andor.tuni.fi/permalink/358FIN_TAMPO/176jdvt/cdi_askewsholts_vlebooks_9781788396349

Moiseev, A. & Yakov, F. 2020. TypeScript Quickly. New York: Manning Publications. Viitattu 2.2.2022. Vaatii käyttöoikeuden. https://andor.tuni.fi/permalink/358FIN_TAMPO/176jdvt/cdi_safari_books_v2_9781617295942

MongoDB. 2021. What Is MongoDB? Verkkosivu. Viitattu 27.2.2022. https://www.mongodb.com/what-is-mongodb

Mozilla. 2021. HTTP Messages. Verkkosivu. Viitattu 12.2.2022. https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages#http_requests

NPM. n.d. NPM Docs. Verkkosivu. Viitattu 8.2.2022. https://docs.npmjs.com/

OpenJS Foundation. n.d. Express. Verkkosivu. Viitattu 12.3.2022. https://expressjs.com/

Pollock, J. 2019. JavaScript: A Beginner's Guide. 5. painos. New York: McGraw-Hill. Viitattu 6.2.2022. Vaatii käyttöoikeuden. https://andor.tuni.fi/permalink/358FIN_TAMPO/176jdvt/cdi_proquest_ebookcentral_EBC6255900

Postman. 2022. Sending your first request. Verkkosivu. Viitattu 12.2.2022. https://learning.postman.com/docs/getting-started/sending-the-first-request/

Preibisch, S. 2018. API Development: A Practical Guide for Business Implementation Success. New York: Apress. Viitattu 12.2.2022. Vaatii käyttöoikeuden. https://andor.tuni.fi/permalink/358FIN_TAMPO/176jdvt/cdi_askewsholts_vlebooks_9781484241400

Remo, J. 2019. Hands-On Functional Programming with TypeScript. Birmingham: Packt Publishing. Viitattu 4.3.2022. Vaatii käyttöoikeuden. https://andor.tuni.fi/permalink/358FIN_TAMPO/1j3mh4m/alma9911130733605973

Santos, L. 2019. Node.js Under The Hood #3 – Deep Dive Into the Event Loop. Verkkosivu. Viitattu 8.2.2022. https://dev.to/khaosdoctor/node-js-under-the-hood-3-deep-dive-into-the-event-loop-135d

Visma. n.d. API – Mikä on API? Verkkosivu. Viitattu 12.2.2022. https://www.visma.fi/epasseli/kirjanpidon-sanakirja/a/api/

LIITTEET

Liite 1. Funktionaalista paradigmaa hyödyntävä ohjelmointirajapinta

```
DB_CONN_STRING="mongodb://127.0.0.1:27017"
DB_NAME="opinnaytetyo"
COLLECTION_NAME="items"
```

```
import { ObjectId } from "mongodb";

export type Item = {
    _id?: ObjectId,
    name: string,
    email: string
}
```

```
import express from 'express';
import { getAllItems, getItem } from './controllers/getFunction';
import { insertItem } from './controllers/postFunction';
import { deleteItem } from './controllers/deleteFunction';

// Luodaan express prosessi
const app = express();
const port = process.env.PORT || 3000;
app.use(express.urlencoded({ extended: true }));
app.use(express.json());

// Asetetaan prosessille portti
app.listen(port, () => {
  console.log(`Running on port ${port}.`);
});

// Päätepiste get
app.get('/items', getAllItems);

// Päätepiste get
app.get('/item', getItem);

// Päätepiste post
app.post('/item', insertItem);

// Päätepiste delete
app.delete('/item', deleteItem);
```

```
import { Request, Response } from 'express';
import { queryDatabase } from "./mongo_functions/connectFunctions";
import { deleteOperation } from './mongo_functions/queryFunctions';

/**
 * Delete päätepiste
 * Hyödyntää higher-order ja callback funktioita
 * Poistaa yhden dokumentin tietokannasta _id parametrin avulla
 */
export async function deleteItem(req: Request, res: Response) {
    try {
        const result = await deleteOperation(queryDatabase, req.query._id as string);
        result ? res.status(202).send('Item deleted') : res.status(500).send('Internal server error');
    } catch (err) {
        res.status(404).send('Not found');
    }
}
```

```
import { Request, Response } from 'express';
import { queryDatabase } from "./mongo_functions/connectFunctions";
import { getAllOperation, getOneOperation } from './mongo_functions/queryFunctions';

/**
 * Get päätepiste
 * Hyödyntää higher-order ja callback funktioita
 * Hakee kaikki dokumentit tietokannasta
 */
export async function getAllItems(req: Request, res: Response) {
    try {
        const result = await getAllOperation(queryDatabase);
        result ? res.status(200).send(result) : res.status(404).send('Not found');
    } catch (err) {
        res.status(404).send('Not found');
    }
}

/**
 * Get päätepiste
 * Hyödyntää higher-order ja callback funktioita
 * Hakee yhden dokumentin tietokannasta
 */
export async function getItem(req: Request, res: Response) {
    try {
        const _id: string = req.query._id ? req.query._id as string : '';
        const result = await getOneOperation(queryDatabase, _id);
        result ? res.status(200).send(result) : res.status(404).send('Not found');
    } catch (err) {
        res.status(404).send('Not found');
    }
}
```

```
import { Request, Response } from 'express';
import { Item } from "../interfaces/data";
import { queryDatabase } from "./mongo_functions/connectFunctions";
import { createOperation } from "./mongo_functions/queryFunctions";

/**
 * Post päätepiste
 * Hyödyntää higher-order ja callback funktioita
 * Luo yhden dokumentin tietokantaan
 */
export async function insertItem(req: Request, res: Response) {
    try {
        const body = req.body as Item;
        const result = await createOperation(queryDatabase, body);
        result ? res.status(201).send('Item inserted') : res.status(500).send('Internal server error');
    } catch (err) {
        res.status(500).send('Internal server error');
    }
}
```

```
import * as mongoDB from "mongodb";

/**
 * Callback funktio, joka annetaan Higher-order funktiolle parametrina
 * Luo yhteyden tietokantaan parametrien avulla ja palauttaa yhteyden
 */
export async function queryDatabase(COLLECTION_NAME: string, DB_CONN_STRING: string, DB_NAME: string): Promise<mongoDB.Collection> {
    const collectionName = COLLECTION_NAME;
    const connectionString = DB_CONN_STRING;
    const databaseName = DB_NAME;

    const client: mongoDB.MongoClient = new mongoDB.MongoClient(connectionString);
    await client.connect();
    const db: mongoDB.Db = client.db(databaseName);

    return db.collection(collectionName);
}
```

```
import { Item } from "../../interfaces/data";
import { ObjectId } from "mongodb";
import * as mongoDB from "mongodb";
import * as dotenv from "dotenv";

/**
 * Higher-order funktio
 * Hyödyntää queryDatabase funktiota
 * Hakee kaikki dokumentit tietokannasta
 */
export async function getAllOperation(connect: (COLLECTION_NAME: string, DB_CONN_STRING: string, DB_NAME: string) => Promise<mongoDB.Collection>): Promise<Item[]> {
    dotenv.config();
    const collection = await connect(
        process.env.COLLECTION_NAME ? process.env.COLLECTION_NAME : '',
        process.env.DB_CONN_STRING ? process.env.DB_CONN_STRING : '',
        process.env.DB_NAME ? process.env.DB_NAME : '',
    );

    return await collection.find({}).toArray() as Item[];
}

/**
 * Higher-order funktio, joka hyödyntää queryDatabase funktiota
 * Hakee yhden dokumentin tietokannasta _id parametrin avulla
 */
export async function getOneOperation(connect: (COLLECTION_NAME: string, DB_CONN_STRING: string, DB_NAME: string) => Promise<mongoDB.Collection>, _id: string): Promise<Item> {
    dotenv.config();
    const collection = await connect(
        process.env.COLLECTION_NAME ? process.env.COLLECTION_NAME : '',
        process.env.DB_CONN_STRING ? process.env.DB_CONN_STRING : '',
        process.env.DB_NAME ? process.env.DB_NAME : '',
    );

    const findQuery = { _id: new ObjectId(_id) };
    const item = await collection.findOne(findQuery) as Item;
    return item;
}

/**
 * Higher-order funktio, joka hyödyntää queryDatabase funktiota
 * Luo yhden dokumentin tietokantaan
 */
export async function createOperation(connect: (COLLECTION_NAME: string, DB_CONN_STRING: string, DB_NAME: string) => Promise<mongoDB.Collection>, _object: Item): Promise<boolean> {
    dotenv.config();
    const collection = await connect(
        process.env.COLLECTION_NAME ? process.env.COLLECTION_NAME : '',
        process.env.DB_CONN_STRING ? process.env.DB_CONN_STRING : '',
        process.env.DB_NAME ? process.env.DB_NAME : '',
    );

    const result = await collection.insertOne(_object);

    if (result.acknowledged) {
        return true;
    }

    return false;
}

/**
 * Higher-order funktio, joka hyödyntää queryDatabase funktiota
 * Poistaa yhden olemassa olevan dokumentin tietokannasta
 */
export async function deleteOperation(connect: (COLLECTION_NAME: string, DB_CONN_STRING: string, DB_NAME: string) => Promise<mongoDB.Collection>, _id: string): Promise<boolean> {
    dotenv.config();
    const collection = await connect(
        process.env.COLLECTION_NAME ? process.env.COLLECTION_NAME : '',
        process.env.DB_CONN_STRING ? process.env.DB_CONN_STRING : '',
        process.env.DB_NAME ? process.env.DB_NAME : '',
    );

    const findQuery = { _id: new ObjectId(_id) };
    const result = await collection.deleteOne(findQuery);

    if (result.acknowledged) {
        return true;
    }

    return false;
}
```

Liite 2. Olio-ohjelmointi paradigmaa hyödyntävä ohjelmointirajapinta

```
DB_CONN_STRING="mongodb://127.0.0.1:27017"
DB_NAME="opinnaytetyo"
COLLECTION_NAME="items"
```

```
import * as mongoDB from "mongodb";
import { ObjectId, Document } from "mongodb";
import { Request, Response } from 'express';

export type Item = {
    _id?: ObjectId,
    name: string,
    email: string
}

export interface ItemInterface {
    collectionName: string;
    databaseName: string;
    connectionString: string;
    getAllItems(req: Request, res: Response): Promise<void>;
    getItem(req: Request, res: Response): Promise<void>;
    insertItem(req: Request, res: Response): Promise<void>;
    deleteItem(req: Request, res: Response): Promise<void>;
}

export interface MongoInterface {
    collection: mongoDB.Collection;
    client: mongoDB.MongoClient;
    db: mongoDB.Db;
    fetchAllDocuments(): Promise<Document[]>;
    fetchDocument(_id: string): Promise<Document>;
    insertDocument(_object: Object): Promise<boolean>;
    deleteDocument(_id: string): Promise<boolean>;
}
```

```
import express from 'express';
import ItemController from './controllers/itemController';

// Luodaan express prosessi
const app = express();
const port = process.env.PORT || 3000;
app.use(express.urlencoded({ extended: true }));
app.use(express.json());

// Asetetaan prosessille portti
app.listen(port, () => {
  console.log(`Running on port ${port}.`);
});

// Luo uusi globaali itemController olio
const itemController = new ItemController();

// Päätepiste get
app.get('/items', itemController.getAllItems);

// Päätepiste get
app.get('/item', itemController.getItem);

// Päätepiste post
app.post('/item', itemController.insertItem);

// Päätepiste delete
app.delete('/item', itemController.deleteItem);
```

```
import * as mongoDB from "mongodb";
import { ObjectId, Document } from "mongodb";
import { MongoInterface } from "../interfaces";

/**
 * MongoController luokka joka implementoi MongoInterface
 * Hyväksyy rakennuksessa yhteysparametrit mihin tahansa tietokantaan
 * Voi hakea, syöttää ja poistaa dokumentteja tietokannasta
 */
export default class MongoController implements MongoInterface {
    collection: mongoDB.Collection
    client: mongoDB.MongoClient;
    db: mongoDB.Db

    constructor(connectionString: string, databaseName: string, collectionName: string) {
        this.client = new mongoDB.MongoClient(connectionString);
        this.db = this.client.db(databaseName);
        this.collection = this.db.collection(collectionName);
    }

    async fetchAllDocuments(): Promise<Document[]> {
        await this.client.connect();
        const documents: Document[] = await this.collection.find({}).toArray() as Document[];
        return documents;
    }

    async fetchDocument(_id: string): Promise<Document> {
        await this.client.connect();
        const findQuery: Object = { _id: new ObjectId(_id) };
        const document: Document = await this.collection.findOne(findQuery) as Document;
        return document;
    }

    async insertDocument(_object: Object): Promise<boolean> {
        await this.client.connect();
        const result = await this.collection.insertOne(_object);

        if (result.acknowledged) {
            return true;
        }

        return false;
    }

    async deleteDocument(_id: string): Promise<boolean> {
        await this.client.connect();
        const findQuery: Object = { _id: new ObjectId(_id) };
        const result = await this.collection.deleteOne(findQuery);

        if (result.acknowledged) {
            return true;
        }

        return false;
    }
}
```

```
import { Item, ItemInterface, MongoInterface } from '../interfaces';
import MongoController from './mongoController';
import * as dotenv from "dotenv";
import { Request, Response } from 'express';

/**
 * ItemController luokka joka implementoi ItemInterface
 * Hyödyntää MongoController oliota tiedon hakemisessa
 * Käsittelee ja vastaa Express kutsuihin
 */
export default class ItemController implements ItemInterface {
    collectionName: string;
    databaseName: string;
    connectionString: string;

    constructor() {
        dotenv.config();
        this.collectionName = process.env.COLLECTION_NAME ? process.env.COLLECTION_NAME : '';
        this.databaseName = process.env.DB_NAME ? process.env.DB_NAME : '';
        this.connectionString = process.env.DB_CONN_STRING ? process.env.DB_CONN_STRING : '';
    }

    getAllItems = async (req: Request, res: Response): Promise<void> => {
        try {
            const mongoController = new MongoController(this.connectionString, this.databaseName, this.collectionName);

            const documents: Item[] = await mongoController.fetchAllDocuments() as Item[];
            documents ? res.status(200).send(documents) : res.status(404).send('Not found');
        } catch (error) {
            res.status(500).send('Internal server error');
        }
    }

    getItem = async (req: Request, res: Response): Promise<void> => {
        try {
            const mongoController = new MongoController(this.connectionString, this.databaseName, this.collectionName);
            const _id: string = req.query._id ? req.query._id as string : '';

            const document: Item = await mongoController.fetchDocument(_id) as Item;
            document ? res.status(200).send(document) : res.status(404).send('Not found');
        } catch (error) {
            res.status(500).send('Internal server error');
        }
    }

    insertItem = async (req: Request, res: Response): Promise<void> => {
        try {
            const mongoController = new MongoController(this.connectionString, this.databaseName, this.collectionName);
            const _object: Object = req.body;

            const isInserted: Boolean = await mongoController.insertDocument(_object);
            isInserted ? res.status(201).send('Item inserted') : res.status(500).send('Internal server error');
        } catch (error) {
            res.status(500).send('Internal server error');
        }
    }

    deleteItem = async (req: Request, res: Response): Promise<void> => {
        try {
            const mongoController = new MongoController(this.connectionString, this.databaseName, this.collectionName);
            const _id: string = req.query._id ? req.query._id as string : '';

            const isDeleted: Boolean = await mongoController.deleteDocument(_id);
            isDeleted ? res.status(202).send('Item deleted') : res.status(500).send('Internal server error');
        } catch (error) {
            res.status(500).send('Internal server error');
        }
    }
}
```