OULUN AMMATTIKORKEAKOULU

Panu Paakkari

OHJELMISTOPOHJA WEB-SOVELLUSTEN KEHITYKSEEN

OHJELMISTOPOHJA WEB-SOVELLUSTEN KEHITYKSEEN

Panu Paakkari
Opinnäytetyö
Syksy 2020
Tietotekniikan tutkinto-ohjelma
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietotekniikan tutkinto-ohjelma, ohjelmistokehitys

Tekijä: Panu Paakkari
Opinnäytetyön nimi: Ohjelmistopohja web-sovellusten kehitykseen
Työn ohjaaja: Kari Laitinen
Työn valmistumislukukausi ja -vuosi: Syksy 2020 Sivumäärä: 38

Tämä työ on tehty oululaiselle Buutti Oy-nimiselle ohjelmistokonsultointia (Buutti Consulting) ja -koulutusta (Buutti Education) tarjoavalla yritykselle. Buutti Oy:n pääosaaminen keskittyy sulautettujen järjestelmien ja web-sovellusten kehittämiseen.

Opinnäytetyön tavoitteena oli luoda web-sovelluskehitykseen käytettävä pohja, jonka avulla asiakasprojektien aloittaminen on sujuvaa ja kehitysympäristön asentamiseen vaadittava aika saadaan mahdollisimman pieneksi. Toteutuksessa käytettiin hyväksi Docker-konttiteknologiaa sekä tyypitettyä TypeScript-ohjelmointikieltä.

Työn lopputuloksena tuotettiin kokonaisuus, joka sisälsi yhtenäistetyn kehitysympäristön sekä CI/CD-tuotantolinjan ohjelmiston testausta ja julkaisua varten. Työ otettiin välittömästi käyttöön ja sitä jatkokehitetään toteutettavien projektien aikana esiintyvien tarpeiden perusteella.

Asiasanat: ohjelmistokehitys, web-sovellus, NodeJS, React, TypeScript, Docker

ABSTRACT

Oulu University of Applied Sciences
Information and communication technologies, Software development

---

Author(s): Panu Paakkari
Title of thesis: Software infrastructure for web application development
Supervisor(s): Kari Laitinen
Term and year when the thesis was submitted: Autumn 2020 Number of pages: 38

This work has been made for Oulu-based company called Buutti Oy. Buutti Oy offers software consulting and training services for other companies and schools. Buutti Oy is mainly operating on embedded systems and web application development.

Goal for this thesis was to produce a base for web application development, which can be used to reduce startup cost of customer projects. Main technologies used in the thesis were Docker container technology and TypeScript programming language, which is a typed superset of JavaScript.

The result was a complete system for web application development, which includes a solid development environment regardless what operating system is used. Complete system was taken immediately in use and it will be developed based on the needs, that will arise from projects where the base has been in use.

Keywords: software development, web application, NodeJS, React, TypeScript, Docker

SISÄLLYS

SANASTO

API	Application Programming Interface, ohjelmointirajapinta
CI/CD	Continuous Integration/Continuous Delivery, jatkuva integraatio ja toimitus
I/O	Input/Output, systeemin vastaanottama ja lähettämä signaali tai data.
NodeJS	Javascript-ohjelmistojen suoritusympäristö.
React	JavaScript-kehitysympäristö käyttöliittymien luomiseen
REST	Representational State Transfer, web-palveluiden arkkitehtuurinen tyyli
TypeScript	Ohjelmointikieli, joka laajentaa JavaScript-ohjelmointikielen ominaisuuksia.

1 JOHDANTO

Tämä opinnäytetyö toteutettiin oululaiselle yritykselle nimeltä Buutti Oy. Buutti on ohjelmistoalan konsultointiyritys, joka on perustettu vuonna 2017. Se työllistää tällä hetkellä n. 40 henkilöä erilaisissa ohjelmistokonsultointi- ja koulutustehtävissä. Työn tarkoituksena oli tuottaa Buutille valmis web-sovelluksille soveltuva projektirunko, jonka avulla asiakasprojektien aloitus on joustavampaa.

Tavoitteena työllä oli helpottaa asiakasprojektien ylläpidettävyyttä ja kehitystä yhtenäisellä kehitysympäristöstä riippumattomalla rungolla, jossa kaikki tarvittavat työkalut ja web-sovelluksen kehitykseen tarvittavat teknologiat ovat helposti asennettavissa ja suoritettavissa. Projektipohjaan kuuluu palvelin, tietokanta sekä käyttöliittymä. Tietokantana on käytetty PostgreSQL-tietokantaa, palvelintoteutuksessa käytössä on Node.js suoritusympäristö ja käyttöliittymä on toteutettu React.js-kehitysympäristöllä. Sekä palvelin että käyttöliittymä on ohjelmoitu käyttäen TypeScript-ohjelmointikieltä.

2 KÄYTETYT TEKNOLOGIAT

2.1 TypeScript

Työn ohjelmointikielenä on käytetty TypeScript-kieltä. TypeScript on Microsoftin kehittämä JavaScript-kielen laajennus, joka lisää kehittäjien kannalta hyödyllisiä ominaisuuksia JavaScript-ohjelmointikieleen. Suurimpana ominaisuutena voidaan pitää tyypityksien lisäämistä muuttujille ja funktioille. Tässä luvussa käydään läpi muutamia tärkeitä ominaisuuksia, jotka helpottavat ohjelmistokehitystä ja laadukkaan ohjelmistokoodin kirjoittamista.

2.1.1 TypeScript-kääntäjä

Yleisesti käytetyistä kääntäjistä poiketen TypeScriptillä kirjotettua ohjelmaa ei käännetä binääritiedostoksi, vaan TypeScript-kääntäjä kääntää tiedostot JavaSript-kielelle. Kääntämisen aikana suoritetaan tyyppitarkistus mahdollisten virheiden varalta. Jos koodissa on virheitä, ei ohjelman kääntäminen onnistu, vaan kääntäjä ilmoittaa virheestä esimerkiksi seuraavalla tavalla:

example1.ts(1,7): error TS2322: Type 'number' is not assignable to type 'string'.

Virheestä voidaan nähdä, missä tiedostossa ja millä rivillä virhe on sekä mikä virheen aiheuttaa. Kyseisessä virheessä kääntäjä ilmoittaa ohjelmoijalle, että virhe on tiedostossa example1.ts rivillä 1 merkissä 7 ja virheen aiheuttaa number-tyyppisen muuttujan asettaminen string-tyypin muuttujaan. Jos kääntämisen jälkeen ei ilmene virheitä, luodaan käännetyt tiedostot tsconfig.json-nimisessä tiedostossa määritettyyn polkuun tai projektin juureen, jos polkua ei ole määritelty. Kääntäjä luo käännetyistä tiedostoista saman nimiset JavaScript-tiedostot. Tämän jälkeen ohjelma voidaan suorittaa käyttäjän valitsemassa suoritusympäristössä, joka on yleisimmin internet-selain tai NodeJS-suoritusympäristö. (2, linkki -> 2. TypeScript: A 10_000 Foot View. -> The Compiler.)

2.1.2 Typescript-projektin asetukset

TypeScript-projektin ja kääntäjän asetuksia voidaan muokata luomalla projektin juureen tiedosto tsconfig.json. Tiedostossa määritellään projektille ja kääntäjälle asetuksia, jonka perusteella kääntäjä suorittaa käännöksen JavaScript-kielelle. Alla esimerkki mahdollisesta asetustiedostosta:

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true
  }
}
```

Tässä tiedostossa määritellään kääntäjälle seuraavia ominaisuuksia. Kääntäjä kääntää kaikki *.ts-päätteiset tiedostot EcmaScript-version 5 mukaisella standardilla, käyttäen CommonJS-moduuliformaattia. Kääntäjän tyyppitarkistus on asetettu tiukimpaan tilaan käyttämällä asetusta strict. Tämä asettaa useita muita tarkistussääntöjä samalla muuttujalla. Kääntäjälle kerrotaan myös, että ES6-moduulien importsyntaksi on sallittu, CommonJS-moduulisyntaksin lisäksi. Moduuli voidaan lisätä käyttämällä seuraavaa syntaksia:

```
import * as path from "path";
```

Tyyppitarkistus on poistettu *.d.ts-päätteisistä tiedostoista, sekä tiedostojen nimeäminen on pakotettu samankaltaiseksi. Näiden sääntöjen lisäksi kääntäjälle voidaan määritellä useita muita asetuksia, jotka löytyvät Typescript-dokumentaatiosta. (1, linkki What is a tsconfig.json.)

2.1.3 Tyyppijärjestelmä

Tyyppijärjestelmä on ennalta määritetty joukko sääntöjä, joita ohjelmiston kääntäjä noudattaa tarkistaessaan virheitä tyyppitarkistuksen aikana. Tyyppijärjestelmiä on yleisesti kahdenlaisia: tyyppi-

järjestelmä, jossa ohjelmoija kertoo täsmällisesti, mitä tyyppiä määritetyt muuttujat ovat, ja tyyppijärjestelmä, jossa tyyppitarkistaja tarkistaa muuttujien tyypin automaattisesti. TypeScript käyttää tyyppijärjestelmää, jossa käyttäjä voi määritellä muuttujien tyypit itse tai antaa tyyppitarkistajan tunnistaa tyypin automaattisesti käyttäjän puolesta. Seuraavassa esimerkissä käyttäjä on määritellyt muuttujien tyypit itse sekä antanut muuttujille tyyppiä vastaavat arvot. (2, linkki -> 3. All About Types -> The ABCs of Types.)

```
const string: string = "string";
const number: number = 1;
const stringArray: string[] = ["string", "lista"];
```

Kun TypeScript-kääntäjän halutaan tunnistavan muuttujan tyypin automaattisesti, voi muuttujan tyypityksen jättää pois.

```
const string = "string";
const number = 1;
const stringArray = ["string", "lista"]
```

2.1.4 TypeScriptin ja JavaScriptin eroavaisuudet

JavaScriptissä datatyypit ovat dynaamisia. Tällä tarkoitetaan, että JavaScriptissä yksi muuttuja voi pitää sisällään useaa erityyppistä dataa ja datan tyypit tunnistetaan vasta, kun ohjelmaa suoritetaan. Kuten aiemmin mainittiin, TypeScript-ohjelma käännetään ja datatyypit tarkistetaan kääntämisen aikana, jolloin ohjelmoijalle ilmoitetaan virheistä jo ennen ohjelman suorittamista.

JavaScript on myös heikosti tyypitetty ja JavaScript muuttaa muuttujien tyyppiä suorituksen aikana parhaaksi katsomallaan tavalla. Tämä saattaa aiheuttaa ongelmia ohjelmaa suoritettaessa, koska operaatioiden lopputulos voi olla jotakin, mitä ohjelmoija ei välttämättä ollut tarkoittanut. TypeScript ei salli tällaisten operaatioiden suorittamista, vaan ilmoittaa virheestä käännöksen aikana. (2, 2. TypeScript: A 10_000 Foot view -> The Type System -> TypeScript Versus JavaScript)

JavaScript-esimerkki:

```
const a = 1;
const b = ["1"];
const c= a + b;
```

```
console.log(c);
```

Kyseinen ohjelma antaa vastaukseksi ”11”. JavaSript tunnistaa muuttujien tyypit string ja number ja olettaa, että ohjelmoija haluaa yhdistää kyseiset muuttujat. Tässä vaiheessa JavaScript muuttaa number-tyypin ja string[]-tyypin string-tyyppisiksi sekä suorittaa operaation, joka yhdistää tulokseksi ”11”.

TypeScript sen sijaan ilmoittaa samasta ohjelmasta virheellä:
example3.ts:3:10 - error TS2365: Operator '+' cannot be applied to types 'number' and 'string[]'.

JavaScriptissä tämä aiheuttaa vaikeasti löydettäviä virheitä ohjelmassa. Ohjelmoijan täytyy olla myös erittäin hyvin tietoinen kaikista muuttujista, joita ohjelmassa käytetään. Tämä vaikeuttaa suuremman kehittäjätiimin ottamista mukaan projektiin. TypeScriptillä ei ole saman tyyppisiä ongelmia, vaan koodin hallinta on helpompaa myös suuremmalla kehittäjätiimillä.

2.2 NodeJS

Projektin palvelin on kirjoitettu käyttäen NodeJS-suoritusympäristöä ja sen ydinkirjastojen lisäksi muun muassa ExpressJS-kirjastoa. NodeJS on asynkroninen tapahtumapohjainen alusta JavaScript-ohjelmien suorittamiseen web-selaimen ulkopuolella. Se on rakennettu Googlen V8-JavaScript- ja WebAssembly.moottorin päälle. Sama moottori on käytössä muun muassa Google Chrome Web-selaimessa ja se on kirjoitettu C++-ohjelmointikielellä.

NodeJS perustuu yksisäikeiseen ohjelmointimalliin, jossa ohjelman suoritus ei blokkaa I/O-operaatioita, vaan ohjelmaa suoritettaessa prosessori suorittaa useita tehtäviä yhtäaikaisesti. Jos jokin operaatio vaatii tietoja, jotka eivät vielä ole saatavilla, prosessori jatkaa muiden tehtävien suorittamista. Prosessorille ilmoitetaan, kun tarvittava tieto on saapunut, jonka jälkeen aiemmin suoritettu tehtävä voi jatkua. (3, linkki Chapter 2. Node programmin fundamentals.)

2.2.1 Node Package Manager

NodeJS sisältää pakettienhallintajärjestelmän, Node Package Manager lyhemmin npm, jolla hallitaan NodeJS-projektin kirjastoja. Npm-komentorivityökalulla käyttäjä voi lisätä projektiin kirjastoja,

jotka ladataan npm-rekisteristä. Yleisesti saatavilla olevia kirjastoja on valtava määrä, ja tämä on osaltaan auttanut NodeJS:n suosion kasvussa.

NodeJS-projekti määritellään komennolla *npm init*, joka luo tiedoston nimeltä package.json. Tämä tiedosto sisältää tietoja projektin tekijästä, lisenssistä ja käytetyistä kirjastoista. Samaan tiedostoon voi myös lisätä komentoja esimerkiksi sovelluksen käynnistystä ja julkaisua varten. Seuraava esimerkki sisältää *npm init*-komennolla luodun package.json-tiedoston.

```
{
  "name": "example",
  "version": "1.0.0",
  "description": "Example of simple package.json file",
  "main": "index.ts",
  "scripts": {
    "start": "node dist/index.js",
    "build": "tsc index.ts"
  },
  "author": "Example Author",
  "license": "ISC",
  "devDependencies": {
    "@types/node": "^14.14.6",
    "typescript": "^4.0.5"
  }
}
```

Tiedostosta käy ilmi projektin nimi, kuvaus, sovelluksen ensisijainen käynnistystiedosto, suoritettavissa olevat skriptit, tekijä, lisenssi sekä käytetyt kirjastot. Projektin käynnistys tapahtuu komennolla *npm start,* joka kyseisesä skriptissä ajaa komennon node dist/index.js ja käynnistää sovelluksen. Ennen projektin käynnistystä on index.js-tiedosto luotava. Se tapahtuu komennolla *npm run build*. Komento ajaa TypeScript-kääntäjän tiedostolle index.ts ja luo siitä tiedoston index.js. Projektissa on käytetty TypeScript-kirjastoa, joka on lisätty kehitysvaiheessa tarvittaviin riippuvuuksiin. @types/node-kirjasto lisää TypeScript-tyypitykset NodeJS:n ydinmoduuleihin.

2.2.2 NodeJS-standardikirjastot

NodeJS sisältää tarvittavat työkalut palvelinympäristön luomiseen. Standardikirjastot ovat saman tyyppisiä kuin muissakin palvelinympäristökielissä. Kirjastot mahdollistavat muun muassa tiedostojen sekä verkkoliikenteen käsittelyn, joka ei ole tavallisessa seilainympäristössä mahdollista pelkillä JavaScript-työkaluilla. Seuraavassa esimerkissä on yksinkertainen http-palvelin kirjoitettuna TypeScript-kielellä. (3, linkki Chapter 2. Node programmin fundamentals)

```
import http from "http";

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((_req, res) => {
    res.statusCode = 200;
    res.setHeader('Content-Type', 'text/plain');
    res.end('Hello World');
});

server.listen(port, hostname, () => {
    console.log(`Server running at http://${host-
name}:${port}/`);
});
```

Ohjelman suoritus käynnistää palvelimen osoitteessa 127.0.0.1:3000 tai localhost:3000 ja tulostaa sivulle tekstin *”Hello World”*. Käynnistyksen yhteydessä konsoliin tulostetaan teksti *”Server running at http://127.0.0.1:3000/”*.

2.3 React

Tämän työn käyttöliittymätoteutus on tehty käyttäen React kehitysympäristöä. React on Facebookin kehittämä JavaScript-kehitysympäristö käyttöliittymän luomiseen. React pohjautuu komponenttilähtöiseen kehitykseen, jossa jokaisella komponentilla voi olla oma tila. Tämä helpottaa sovelluksen monimutkaista tilan hallintaa. Tilan vaihtuessa React renderöi komponentin automaattisesti

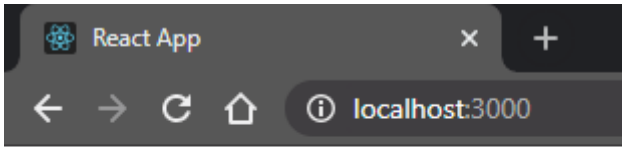
uudelleen. Tilaa hallitaan komponentin parametreillä sekä sen sisäisellä tilalla. Yhdistettynä TypeScriptin kanssa komponenttien ja sovelluksien tekeminen on laadukkaampaa tiukan tyypityksen ansiosta. (4.)

2.3.1 React-komponentit

Stateless funktionaalinen komponentti

Jokainen React-komponentti sisältää render-funktion, joka ottaa sisään renderöitävän elementin datan ja palauttaa näytettävän komponentin. Esimerkeissä käytetään XML-tyylistä TSX-syntaksia. Seuraavassa esimerkissä luodaan yksinkertainen komponentti, jolla ei ole omaa tilaa, vaan renderöitävä data asetetaan komponentin parametriksi. Tällaista komponenttia kutsutaan stateless-komponentiksi. Kuvassa 1 näytetään alla olevan ohjelmakoodin luoma sovellus.

```
import React from 'react';
interface HelloExampleProps {
    name: string;
}

const HelloExample: React.FC<HelloExam-
pleProps> = props => {
    return (
        <div>
            <p>Hello, {props.name}</p>
        </div>
    );
}

const App = () => {
    return (
        <div>
            <HelloExample name="John" />
        </div>
    );
}

export default App;
```



Hello, John

Kuva 1. Renderöity React-komponentti

Stateful funktionaalinen komponentti

Stateful-komponentti sisältää sisään otettavien parametrien lisäksi oman tilan. Kun tila päivittyy, renderöidään komponentti uudelleen. Jos usealla komponentilla on sama tila, siirretään komponenttien tila niiden ensimmäiselle yhteiselle isäntäkomponentille. Seuraavassa esimerkissä luodaan kellonaikakomponentti, joka hallitsee omaa tilaa ja renderöi näkyville kellonajan.

```
import React, { useEffect, useState } from 'react';

const Clock = () => {
    const [time, setTime] = useState(new Date());

    useEffect(() => {
        const clockTick = setInterval(() => {
            const newTime = new Date();
            setTime(newTime);
        }, 1000);
        return () => {
            clearInterval(clockTick);
        };
    }, []);

    return (
        <div style={{ padding: "12px" }}>
            <b>{time.toLocaleTimeString()}</b>
        </div>
    );
};
export default Clock;
```

Esimerkki sisältää useState- ja useEffect-nimiset funktiot, joita kutsutaan nimellä Hook. Nimensä mukaisesti useState-hook hallitsee komponentin tilaa. Se palauttaa käyttäjälle taulukon, joka sisältää tilan ja funktion, jolla tilaa päivitetään.

UseEffect-hookissa käsitellään komponentin sivuvaikutuksia. Esimerkissä luodaan ajastin, joka sekunnin välein päivittää komponentin tilan. UseEffect-hookin palautuksessa kyseinen ajastin poistetaan, jotta komponentin päivitystä ei luoda, kun komponenttia ei näytetä.

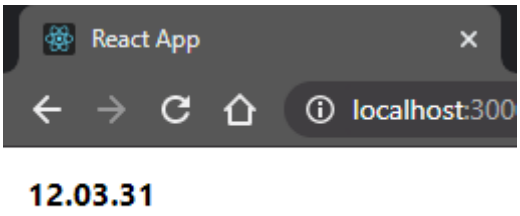
Luokkakomponentti

Reactissa on käytössä myös luokkakomponentti. Tässä työssä luokkakomponentteja ei kuitenkaan käytetty, sillä työssä pyritään noudattamaan funktionaalisia ohjelmointiperiaatteita, jota varten on funktionaaliset komponentit. Luokkakomponenteista on silti hyvä tietää ja seuraavassa esimerkissä luodaan kello käyttäen luokkakomponenttia.

```
import React, { Component } from 'react';

type ClockState = {
    time: Date;
}

export default class ClockClass
      extends Component<{}, ClockState, number>
      {
    constructor(props: {}) {
        super(props);
        this.state = {
            time: new Date()
        };
    }
    timerID: NodeJS.Timeout | null = null;

    clockTick() {
        this.setState({
            time: new Date()
        });
    };
```

```
    componentDidMount() {
        this.timerID = setInterval(() =>
           this.clockTick(), 1000);
    }

    componentWillUnmount() {
        if (this.timerID)
            clearInterval(this.timerID);
    }

    render() {
        return (
            <div style={{ padding: "12px" }}>
                <b>{this.state.time
                      .toLocaleTimeString()}
                </b>
            </div>
        );
    }
}
```

Esimerkissä huomataan, kuinka useEffect-hook on korvattu komponentin elinaikametodeilla componentDidMount ja componentWillUnmount. Molemmat esimerkit tuottavat täsmälleen samanlaisen komponentin näytettäväksi. (Kuva 2.)



Kuva 2. React-kellokomponentti

2.4 Docker

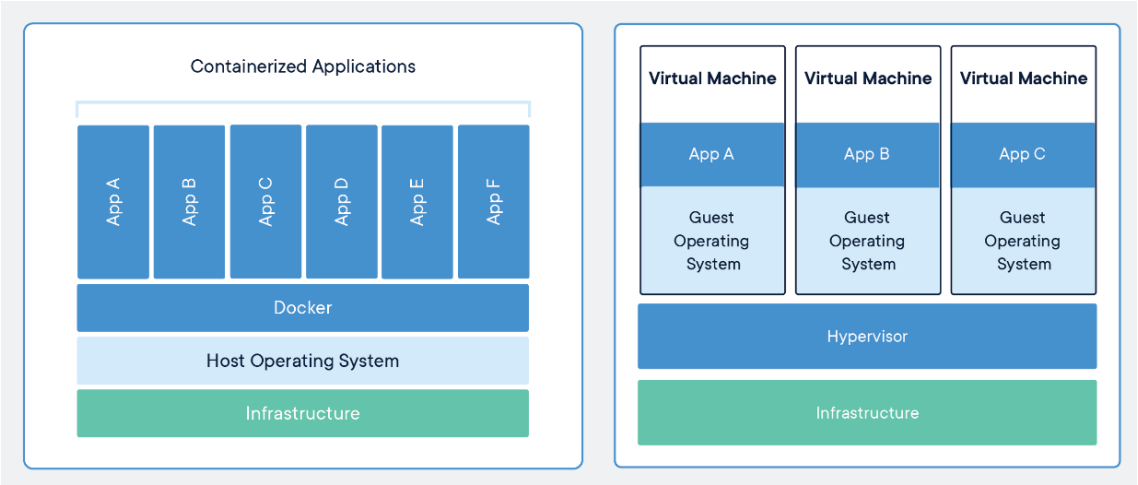
Tässä työssä käytettiin Docker-konttiteknologiaa, joka mahdollistaa helpon ohjelmiston toimittamisen tuotantoympäristöön. Docker on konttiteknologiaan perustuva, eristetyn ympäristön hallinta järjestelmä, joka on amerikkalaisen Docker Inc. yrityksen ylläpitämä.

2.4.1 Kontti

Kontilla tarkoitetaan standardoitua ohjelmistoyksikköä, joka sisältää kaikki ohjelmiston vaatimat riippuvuudet. Tällä tavalla ohjelmiston suorittaminen onnistuu ympäristöstä riippumatta. Kontin suoritusta varten tarvitaan ohjelmistokuvake, joka on suoritettava ohjelmistopaketti, sisältäen kaiken tarvittavan ohjelmiston suorittamista varten. Ohjelmistokuvakkeen suorittaminen luo kontin, joka suoritetaan eristetyssä ympäristössä käyttöjärjestelmästä riippumatta.

2.4.2 Docker ja eristetty ympäristö

Kuvassa 3 on kuvattu Docker-ympäristön ja tavanomaisen virtuaaliympäristön eroavaisuuksia. Docker-ympäristössä jokainen sovellus suoritetaan omasta kuvakkeestaan, joka ajetaan Docker-suoritusympäristössä. Nämä kuvakkeet ovat pienempiä kuin tavanomaiset virtuaaliympäristöt, sillä ne ei sisällä abstraktiota kokonaisesta fyysisestä laitteistosta, ainoastaan abstraktion sovelluskerroksesta. (5, linkit Why Docker? -> What is a container?)



Kuva 3. Docker-ympäristö ja virtualisoituympäristö (5, linkit Why Docker? -> What is a container?)

Kuvassa 4 luodaan yksinkertainen Docker-sovelluskuvake, joka tulostaa konsoliin tekstin Hello World! Kuvake käyttää pohjana Alpine Linux-kuvaketta. Luodun kuvakkeen koko on ainoastaan 5 MB.

```
FROM alpine
CMD ["echo", "Hello World!"]

docker build --tag hello-world:latest .

docker run hello-world:latest
Hello World!
```

Kuva 4. Dockerfile, kuvakkeen luonti ja suoritus

2.5 Testaus ja testiautomaatio

Työn testiautomaatiossa käytettiin hyväksi Gitlab-ohjelmistokehitys alustaa. Testit luotiin käyttäen Jest.js- ja Cypress-testikehitysympäristöjä, jotka tarjoavat hyvät työkalut palvelinohjelmiston sekä käyttöliittymän testaukseen.

2.5.1 Gitlab

Versionhallintaan ja testiautomaation tekemiseen käytettiin Gitlab-sovelluskehitysalustaa, joka mahdollistaa integraatiolinjan tekemisen yksinkertaisilla Gitlab-CI/CD-työkaluilla. Testiautomaatio luodaan käyttäen *.gitlab-ci.yml* tiedostoa, jonne määritellään ajettavat komennot ja suoritusympäristöt. Komennot suoritetaan Gitlab Runner-sovelluksessa, joka suorittaa CI/CD-tuotantolinjan tehtäviä. Tehtävien suorituksessa voi joko käyttää Docker-konttia tai suorittaa ajettavat tehtävät shell-skriptinä. Työssä käytetään Docker-kontteja, joten myös CI/CD-tuotantolinja suoritetaan Docker-ympäristössä. Kuvassa 5 on yksinkertainen automaatiotestaus määritelmä .gitlab-ci.yml-tiedostossa.

```yaml
# Official framework image. Look for the different tagged releases at:
# https://hub.docker.com/r/library/node/tags/
image: node:latest

# Pick zero or more services to be used on all builds.
# Only needed when using a docker container to run your tests in.
# Check out: http://docs.gitlab.com/ee/ci/docker/using_docker_images.html#what-is-a-service
services:
  - mysql:latest
  - redis:latest
  - postgres:latest

# This folder is cached between builds
# http://docs.gitlab.com/ee/ci/yaml/README.html#cache
cache:
  paths:
    - node_modules/

test_async:
  script:
    - npm install
    - node ./specs/start.js ./specs/async.spec.js

test_db:
  script:
    - npm install
    - node ./specs/start.js ./specs/db-postgres.spec.js
```

Kuva 5. .gitlab-ci.yml-tiedosto NodeJS-ympäristöön (7)

2.5.2 Jest

Yksikkö- ja integraatiotestit kirjoitettiin pääasiassa käyttäen Jestjs-kirjaston tarjoamia työkaluja. Jestjs tarjoaa hyvät ominaisuudet funktioiden tuottaman tuloksen vertailuun sekä ulkoisten kirjastojen yksinkertaiseen jäljittelemiseen. Seuraavassa esimerkissä on malli yksinkertaisesta yksikkötestistä kirjoitettuna Jestjs-kirjaston avulla. Kuvassa 6 on suoritetun testin tuottama raportti.

Testattava funktio:

```
export const sum =
      (num1: number, num2: number) => num1 + num2;
```

Ajettava testitiedosto ja testin tulos:

```
import { sum } from "../index";

describe("Test index.ts file", () => {
  it("Should sum two numbers together", () => {
```

```
            const result = sum(5, 5);
            expect(result).toEqual(10);
        });
    })
```

```
PASS tests/index.test.ts
  Test index.ts file
    ✓ Should sum two numbers together (5 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.002 s
Ran all test suites.
```

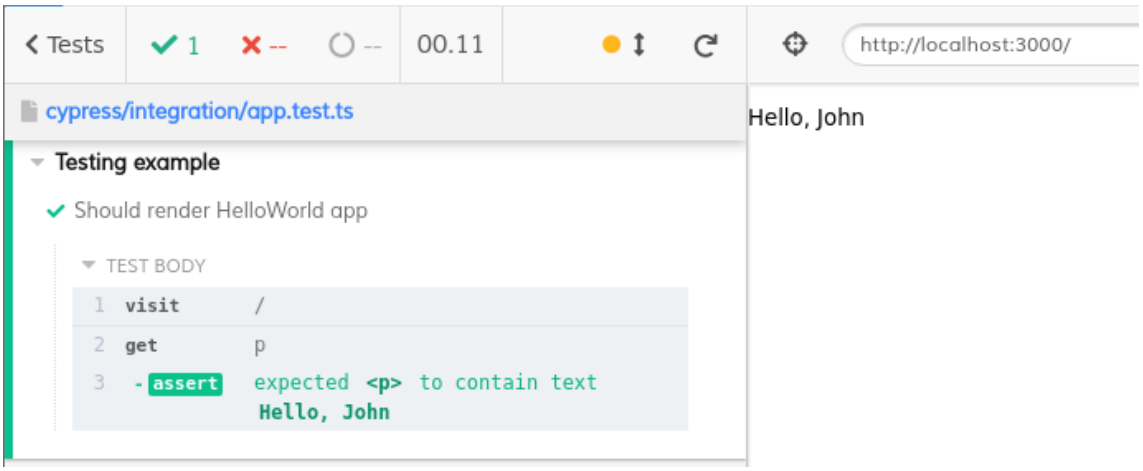
Kuva 6. Jest-yksikkötestin tulos

2.5.3 Cypress

Työn systeemitason testaus toteutettiin Cypress-kirjastoa käyttäen. Kirjasto tarjoaa kattavat työkalut projektien täydelliseen käyttökokemuksen testaukseen ja raportoi tuloksista kuvina, videoina ja virheviesteinä. Testauksella pyritään havaitsemaan ongelmat ohjelmiston suorituksessa ennen sen julkaisua tuotantoympäristöön. Seuraavassa esimerkissä on kirjoitettu yksinkertainen testi aiemman esimerkin Hello-sovellukselle, joka tarkistaa käyttöliittymän tuottaman näkymän oikeaksi.

```
describe("Testing example", () => {
    it("Should render HelloWorld app", () => {
        cy.visit("/")
        cy.get("p")
            .should("contain.text", "Hello, John")
    })
})
```

Testistä raportoidaan tulokset Cypress käyttöliittymään sekä kuvina että tekstinä. Käyttöliittymä avataan komennolla *cypress open* Testiautomaatiolinjalla testit suoritetaan ilman näkyvää käyttöliittymää, niin kutsutussa headless-tilassa. Tällöin testeistä raportoidaan tulokset tekstinä ja kuvat sekä videot tallennetaan ainoastaan testeistä, jotka eivät ole suoriutuneet onnistuneesti. Headless-tilan voi suorittaa komennolla *cypress run*. Kuvissa 7 ja 8 on testeistä syntyviä testiraportteja.



Kuva 7. Cypress-testiraportti käyttöliittymässä

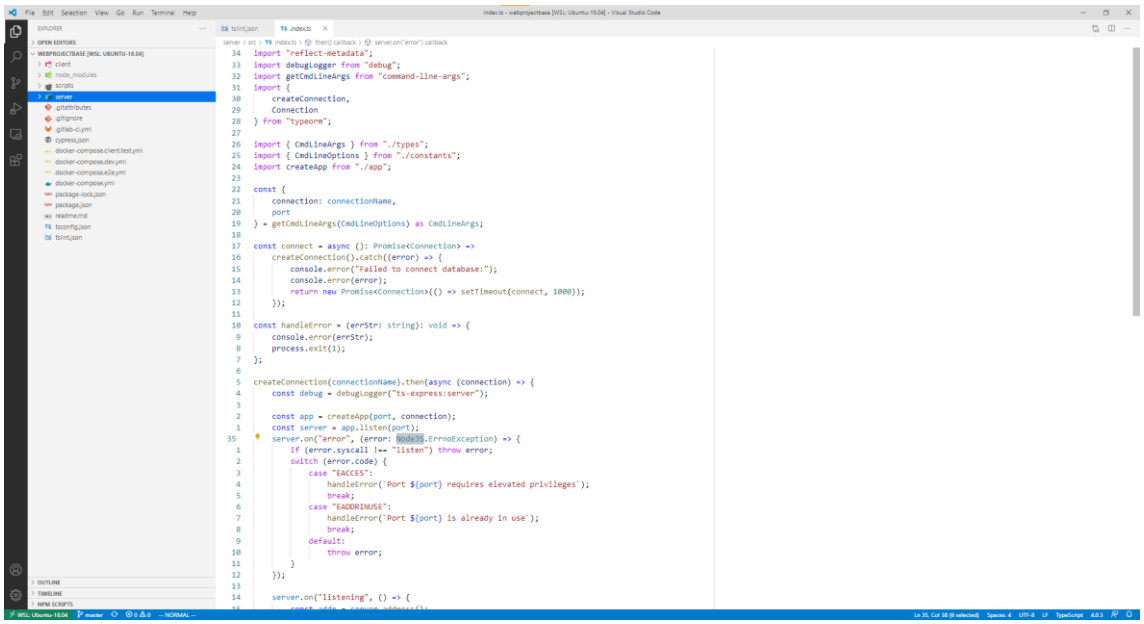
```
(Run Starting)

  Cypress:    5.6.0
  Browser:    Electron 85 (headless)
  Specs:      1 found (app.test.ts)


Running:  app.test.ts                                                (1 of 1)

Testing example
  ✓ Should render HelloWorld app (311ms)

1 passing (323ms)

(Results)

  Tests:        1
  Passing:      1
  Failing:      0
  Pending:      0
  Skipped:      0
  Screenshots:  0
  Video:        false
  Duration:     0 seconds
  Spec Ran:     app.test.ts


(Run Finished)

       Spec                                  Tests  Passing  Failing  Pending  Skipped
  ✓  app.test.ts                  318ms        1        1        -        -        -
  ✓  All specs passed!            318ms        1        1        -        -        -
```

Kuva 8. Cypress-testiraportti headless-tilassa

3 KÄYTETYT TYÖVÄLINEET

3.1 Visual Studio Code

Työn ohjelmistokehitysympäristönä käytettiin ilmaista Visual Studio Code-tekstieditoria (kuva 9), johon on tarjolla laajasti erilaisia lisäosia, joilla voi helpottaa ohjelmiston kehitystä. Ohjelmiston laadun parantamiseksi käytössä oli muun muassa tslint-lisäosa, joka kertoo kirjoitetun koodin tyylivirheistä ja pitää eri kehittäjien kirjoittaman koodin yhden mukaisena koko projektin sisällä. Säännöt projektin sisällä määritellään -tslintrc-tiedostoon, josta lisäosa tarkistaa käytettävät koodistandardit.



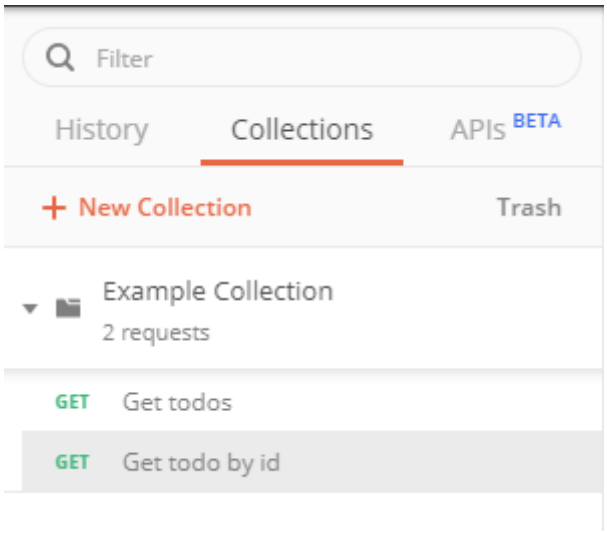
Kuva 9. Visual Studio Code-tekstieditori

3.2 Ubuntu 18.04

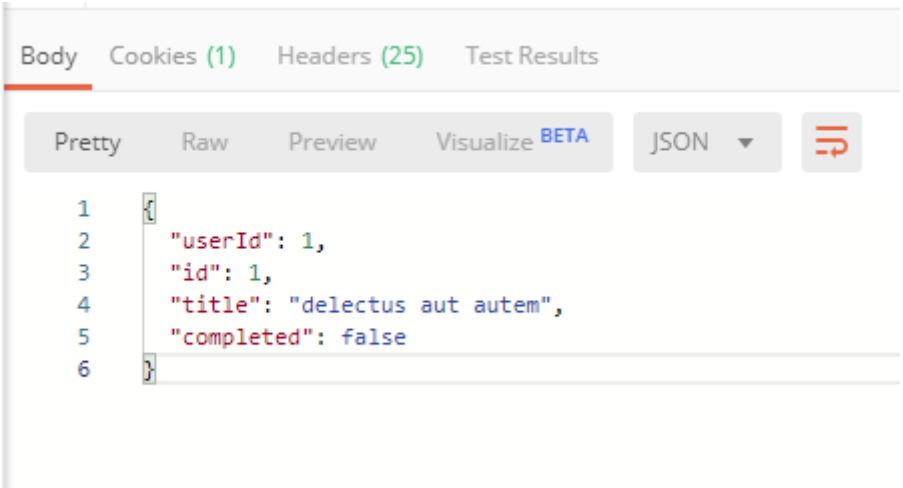
Käyttöjärjestelmänä työn toteutuksen aikana oli Ubuntu 18.04, joka tarjoaa luotettavan ympäristön suoritettaville sovelluksille sekä kehitysympäristöille. Työ pyrittiin toteuttamaan kuitenkin niin, että kehittäjä voi käyttää käyttöjärjestelmänä mitä tahansa yleisimmistä käyttöjärjestelmistä. Tämä mahdollisti Docker-ympäristössä suoritettavat ohjelmistokuvakkeet sekä suoritettavat ohjelmat. Lähes kaikki sovellukset suoritettiin Ubuntun shell-komentokehotteesta ja suoritettavia prosesseja varten kirjotettiin shell-skriptejä työn automatisointia varten.

3.3 Postman

Palvelimen rajapintojen dokumentointiin ja testaukseen käytettiin hyväksi Postman-sovellusta, joka mahdollistaa käytettyjen rajapintakutsujen tallentamisen, lähettämisen sekä vastausten esikatselun. Tallentamista varten Postmanissa pystytään luomaan projektikohtainen kokoelma (Kuva 10), jonne REST-rajapintojen kutsut voidaan määrittää. Kutsuja varten voi luoda myös automatisoituja testejä, mutta tässä projektissa rajapintojen testaukseen ei käytetty Postmanin tarjoamaa mahdollisuutta, vaan sitä käytetiin ainoastaan vastausten esikatseluun ja dokumentaation luomiseen. Kuvassa 11 on esitetty Postman-käyttöliittymässä esikatseltava http-pyynnön vastaus.



Kuva 10. Esimerkki API Collection



Kuva 11. Esimerkki vastauksen esikatselusta

4 PROJEKTIN TOTEUTUS

4.1 Arkkitehtuuri

4.1.1 REST-API

REST-API:n arkkitehtuuri noudattaa perinteistä MVC-mallia, model-view-controller, jossa tietokantataulut voidaan tulkita REST-API:n model-osana, API:n kutsuttavia osoitteita mallin view-osiona sekä logiikkaa kutsuttavien osoitteiden takana mallin controller-osana.

4.1.2 Tietokanta ja TypeORM

Taulut mallinnettiin käyttäen TypeORM-kirjastoa, jolla on mahdollista luoda tietokannan vaatimat taulut sekä relaatiot käyttämällä TypeScript-ohjelmointikieltä. Taulut tallennettiin PostgreSQL-tietokantaan, joka ajettiin kehitysvaiheessa eristetyssä sovelluskontissa. Tuotantovaiheessa tietokanta siirretään pilvipalvelutarjoajan tietokantapalvelimelle.

TypeORM-kirjaston lisäksi apuna on käytetty class-transformer- ja class-validator-kirjastoja. Class-transformer-kirjastolla voidaan määrittää, mitkä luokan muuttujat paljastetaan, kun tieto palautetaan tietokannasta rajapinnalle ja muutetaan TypeScript-objektiksi. Esimerkiksi käyttäjän salasana voidaan jättää pois objektista käyttämällä dekoraattoria @transformer.Exclude(). Dekoraattorit ovat funktioita, jotka suoritetaan annetulle parametrille sitä luodessa. Seuraavassa esimerkissä luodaan tietokantataulu käyttäen TypeORM-kirjastoa. Kuvassa 12 on mallinnettu tietokantataulu, joka on luotu käyttäen TypeORM-kirjastoa.

```
@typeorm.Entity()
@transformer.Exclude()
export class User {

    @transformer.Expose({ toPlainOnly: true })
    @validator.IsOptional()
    @typeorm.PrimaryGeneratedColumn()
    public id: number;

    @transformer.Expose({ toPlainOnly: true })
    @validator.IsOptional()
    @typeorm.UpdateDateColumn({
        type: "timestamp with time zone"
    })
    public createdAt: number;

    @transformer.Expose({ toPlainOnly: true })
    @validator.IsOptional()
    @typeorm.UpdateDateColumn({
        type: "timestamp with time zone"
    })
    public updatedAt: number;

    @typeorm.Column()
    public hash: string;

    @typeorm.Column()
    public salt: string;

    @transformer.Expose()
    @validator.IsDefined(errors.isDefined)
    @validator.IsString(errors.isString)
    @validator.IsNotEmpty(errors.isNotEmpty)
    @validator.MaxLength(225, errors.maxLength)
    @typeorm.Column()
    public fullName: string;

    @transformer.Expose()
    @validator.IsDefined(errors.isDefined)
    @validator.IsString(errors.isString)
    @validator.MaxLength(225, errors.maxLength)
    @validator.IsEmail({}, errors.isEmail)
    @typeorm.Column({ unique: true })
    public email: string;
}
```

Kuva 12. Tietokantataulu luotuna TypeORM-kirjastolla.

Esimerkki class-transformer-kirjaston exclude-ominaisuudesta.

```
@transformer.Exclude()
public password: string;
```

```
const getUser = (userId: number, repository: Reposi-
tory<User>) =>
    repository.findOneOrFail(userId)
        .then(user => classToPlain(user))
        .catch(err => console.error(err));
```

Esimerkkikoodilla tietokannasta palautuu käyttäjä, mutta käyttäjän salasanaa ei palauteta vastauksessa.

```
{
    "id": 2,
    "createdAt": "2020-11-22T12:32:14.614Z",
    "updatedAt": "2020-11-22T12:32:14.614Z",
    "fullName": "Testi",
    "email": "testi@testi.com"
}
```

4.1.3 API-polut ja ExpressJS

API:n rajapintana toimivat url-polut luotiin käyttämällä ExpressJS-kirjastoa, joka toimii yksinkertaisena väliohjelmistojen ja NodeJS-standardikirjastojen rajapintana. Polkuja luotaessa käytettiin REST-periaatteiden mukaisia tapoja ja polut suunniteltiin sellaiseksi, että uusien polkujen lisääminen on helppoa ja samaa rakennetta pysytään käyttämään monipuolisesti myös muiden polkujen luomisessa. Seuraavassa esimerkissä on luotu polku *"users ja login"*.

```
import { Connection } from "typeorm";
import { Router } from "express";
import createUserRoutes from "./userRoutes";
import createLoginRoute from "./loginRoute";
import passport = require("passport");

export default (connection: Connection): Router => {
    const router = Router();
    router.use("/login", createLoginRoute());
    router.use("/users", createUserRoutes(connection));
    router.use(
        "/*",
        passport.authenticate("jwt", { session: false })
    );
    return router;
};
```

Esimerkkikoodissa funktio ottaa parametrina tietokantayhteyden ja palauttaa router-objektin, johon lisätään polku *"/users"* sekä *"/login"*. Polkuja käytetään käyttäjien luomisessa sekä käyttäjien hakemisessa tietokannasta. Polku *"/*"* tarkoittaa, että kaikki tämän jälkeen määritellyt polut menevät

väliohjelmiston läpi, joka tarkistaa käyttäjän käyttöoikeudet kyseisille poluille. Seuraavaksi määritellään *"/users"*-polut http-metodeja varten. Kyseisessä esimerkissä on määritelty metodit GET ja POST. GET-metodi määritellään sekä kokonaiselle käyttäjätaululle että yksittäiselle käyttäjälle id-parametrin perusteella. Näiden lisäksi jokaiselle polulle määritellään kontrollerifunktiot, jotka palauttavat vastaukset kutsutulle rajapinnalle. Seuraavassa esimerkissä on määritelty polut metodeja varten sekä lisätty välisovellus tarvittaville metodeille oikeuksien tarkistamista varten.

```
import { Connection } from "typeorm";
import { Router } from "express";

import createUserController from "../controllers/userControl-
ler";
import { makeRequestHandler } from "./utils";
import passport = require("passport");

export default (connection: Connection): Router => {
    const {
        getAll,
        getOne,
        post
    } = createUserController(connection);
    const router = Router();
    router.get("/",
        passport.authenticate("jwt", { session: false }),
        makeRequestHandler(getAll)
    );
    router.get(
        "/:id",
        passport.authenticate("jwt", { session: false }),
        makeRequestHandler(getOne)
    );
    router.post("/", makeRequestHandler(post));
    return router;
};
```

4.1.4 Kontrollerit

Kontrollerit kuvaavat nimensä mukaisesti MVC-mallin Controller-osaa. Ne hallitsevat logiikan url-polkujen takana. Projektiin luotiin kontrollerit jokaiselle tarvittavalle http-metodille REST periaatteita noudattaen. Jokainen resurssi ja sen tarvittavat http-metodit ottavat kutsun ja saavat vastauksen JSON-muodossa. Tietokantahakujen tekemiseen käytettiin TypeORM-kirjaston tarjoamia rajapintoja. Seuraavassa esimerkissä on kirjoitettu logiikka käyttäjälistan hakemiseksi.

```
const makeGetAll = (repository: Repository<User>):
ControllerMethod =>
    async (): Promise<ControllerResponse> =>
        repository.find()
            .then((users) => createStatusOkResponse(
                users.map(user => classToPlain(user))
            ));
```

Esimerkin koodi hakee tietokannan käyttäjätaulusta listan käyttäjistä ja lähettää listan käyttäjistä JSON-vastauksena. Vastaus viedään class-transformer-kirjaston classToPlain-funktion kautta, joka varmistaa, että käyttäjästä palautuu vain tarpeelliset tiedot.

4.1.5 Käyttöliittymä

Projektiin ei luotu varsinaista käyttöliittymää, mutta sitä varten tehtiin valmis pohja. Pohjaan määriteltiin kansiorakenne, joka sisältää kansiot näkymille ja komponenteille. Osalle näkymistä luotiin valmis runko, sekä niitä varten luotiin reititys, joka ohjaa käyttäjän url-osoitteen perusteella oikeaan näkymään. Valmiita näkymärunkoja pystytään käyttämään sellaisenaan hyväksi uusia näkymiä luodessa. Seuraavassa esimerkissä on luotu komponentti, joka sisältää näkymille luodun reitityksen ja reititys on otettu käyttöön käyttämällä HashRouter-komponenttia.

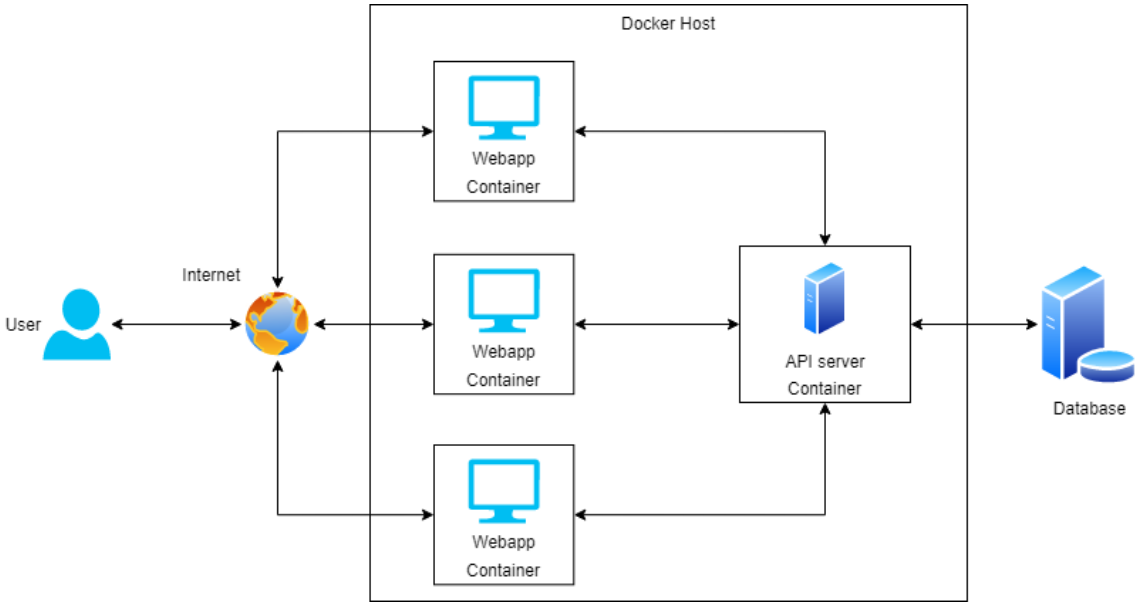
```
import About from "./pages/About/About";
import Home from "./pages/Home/Home";
import { RoutePath } from "./types";

const Routes: RoutePath[] = [
```

```
    { component: Home, path: "/", exact: true, text: "Home" },
    { component: About, path: "/about", text: "About" }
];

export default Routes;

import React from "react";
import { HashRouter } from "react-router-dom";
import "./App.scss";
import PageContent from "./components/page-content/PageContent";
import Routes from "./routes";

const App: React.FC = () => (
    <div className="app">
        <HashRouter>
            <PageContent pages={Routes} />
        </HashRouter>
    </div>
);

export default App;
```

Testejä varten määriteltiin valmiiksi tarvittavat kirjastot, sekä yksittäisten komponenttien kehittämistä varten Storybook-kirjasto, jolla komponentti pystytään eristämään kokonaisesta käyttöliittymästä. Käyttöliittymää varten kirjoitettiin myös hieman logiikkaa esimerkiksi kirjautumista varten, joka on useimmissa projekteissa samankaltainen.

4.1.6 Kokonaisuus

Kokonaisuutena arkkitehtuuri noudattaa perinteistä web-sovellusarkkitehtuuria, joka on helposti skaalattavissa käyttötarkoitukseen sopivaksi. Pääsy tietokantaan eristettiin, niin että kommunikointi muista osoitteista kuin REST-API:n palvelimelta on estetty. Kommunikaatio tietokantaan tapahtuu ainoastaan REST-API:n kautta tehtävien kutsujen avulla. Tietokantaa ei käytetä Docker-kontissa, vaan palvelimena käytetään joltain yleiseltä pilvipalvelutarjoajalta saatavilla olevaa tietokantaa. Eristetty Docker-ympäristö mahdollistaa muun muassa kuvan 13 kaltaisen arkkitehtuurin luomisen, jossa yhdelle API:lle voi olla yhteydessä useampi eri sovellus.



Kuva 13. Web-sovelluksen arkkitehtuuri

4.2 Yhtenäinen kehitysympäristö

Työssä pyrittiin luomaan käyttöjärjestelmästä riippumaton kehitysympäristö, jotta kehittäjien olisi helppo aloittaa projekteissa riippumatta käytettävissä olevasta laitteistosta. Tähän pyrittiin käyttämällä Dockerin tarjoamaa mahdollisuutta eristetyistä ympäristöistä. Työhön määriteltiin valmiiksi komennot, joilla sovellus saatiin rakennettua sekä ajettua tuotannonomaisessa ympäristössä myös omalla laitteella.

Projektia rakentaessa luodaan sovelluskuvake, joka sisältää kaikki tarvittavat riippuvuudet sovelluksen ajamiseen. Kuvakkeen pohjaksi valittiin Docker-Hubista saatavilla oleva NodeJS version 10 kuvake. Tarvittavat tiedostot siirretään sovelluskuvakkeelle ja ajetaan tarvittavat npm komennot, joilla projektiin lisätään kaikki tarvittavat kirjastot. Kuvassa 14 on Dockerfile, jolla määritellään toimenpiteet kuvakkeen luomiseksi.

```
FROM node:10
RUN apt update
RUN apt install -y postgresql-client
WORKDIR /app/client
COPY client/src/ /app/client/src
COPY client/public /app/client/public
COPY client/package.json /app/client
COPY client/tsconfig.json /app/client
RUN npm install --production --no-optional
WORKDIR /app/server
RUN mkdir /app/server/scripts
RUN mkdir /app/server/src
RUN mkdir /app/server/dist
COPY server/package.json /app/server
COPY server/tsconfig.json /app/server
COPY server/tslint.json /app/server
COPY server/jest.config.ts /app/server
COPY server/src/ /app/server/src
COPY server/test /app/server/test
COPY server/ormconfig.json /app/server
COPY server/scripts/ /app/server/scripts
COPY server/__mocks__/ /app/server/__mocks__
RUN npm install --production
```

Kuva 14. Dockerfile

Tämän jälkeen kuvake on suoritettavissa Docker-komennoilla, jotka käyttäjän helpottamiseksi on määritelty valmiiksi package.json tiedostoon, josta ne voidaan ajaa npm-komennoilla. Package.json (kuva 15) sisältää useita komentoja, jotka helpottavat kehittäjää suorittamaan ja rakentamaan sovelluskuvakkeita eri käyttötarkoituksia varten.

```
"scripts": {
    "start": "npx cross-env ./scripts/local_server_env.sh",
    "start:dev": "npx cross-env ./scripts/start_dev.sh",
    "build": "npx cross-env ./scripts/build.sh",
    "test": "npm run test:server:container && npm run test:client:container && npm run test:e2e:container",
    "test:server:container": "npx cross-env ./scripts/test_server_env.sh",
    "test:client:container": "npx cross-env ./scripts/test_client_container.sh",
    "test:e2e:container": "npx cross-env ./scripts/test_e2e_container.sh",
    "test:unit": "npm run test:server:unit && npm run test:client:unit",
    "test:server:unit": "npx cross-env ./scripts/run_unit_tests_server.sh",
    "test:client:unit": "npx cross-env ./scripts/run_unit_tests_client.sh",
    "lint": "npm run lint:client && npm run lint:server",
    "lint:fix": "npm run lint:client:fix && npm run lint:server:fix",
    "lint:client": "tslint --project client --format verbose",
    "lint:client:fix": "tslint --project client --format verbose --fix",
    "lint:server": "tslint --project server --format verbose",
    "lint:server:fix": "tslint --project server --format verbose --fix",
    "docker:down": "docker-compose down",
    "docker:build": "npm run docker:down && docker-compose build",
    "docker:up": "docker-compose up --force-recreate",
    "docker:up:exit": "npm run docker:up -- --exit-code-from server",
    "install:all": "./scripts/install_all.sh"
},
```

Kuva 15. Package.json-tiedoston scriptit

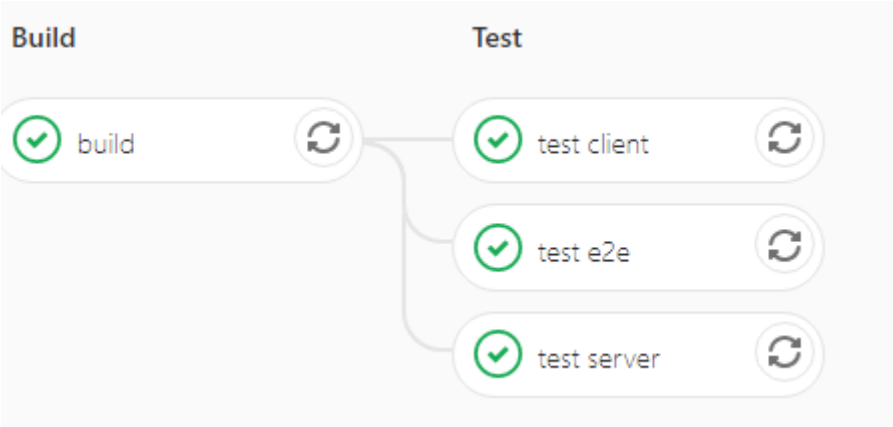
4.3 Funktionaalinen ohjelmointi

Työssä pyrittiin noudattamaan funktionaalisen ohjelmoinnin periaatteita, jonka mukaan funktiot määritellään puhtaiksi funktioiksi (engl. Pure Functions). Puhtailla funktioilla tarkoitetaan funktioita, jotka eivät sisällä sivuvaikutuksia sekä palauttavat annetuilla parametreilla aina saman tuloksen. Funktionaalisen ohjelmoinnin mukaan työssä pyrittiin myös välttämään jaettua tilaa sekä muuttujien mutaatiota. Tällä tavalla helpotettiin testaamista sekä suoritettujen funktioiden lopputuloksen ennustettavuutta. Funktionaalinen ohjelmointi helpottaa myös sovelluskoodin lukemista, koska funktiot pyritään määrittämään selittävään muotoon.

4.4 Laadunvarmistus

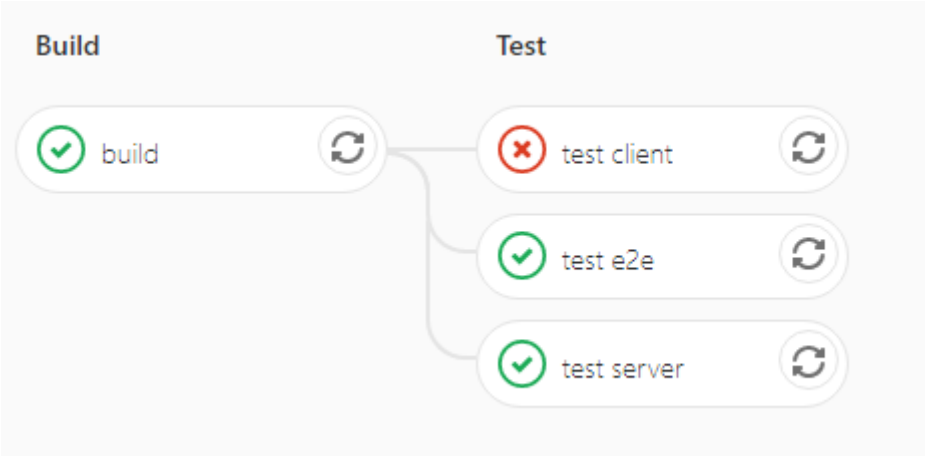
Koodin laatua varmistettiin luomalla tarkat säännöt koodin formaatille, käyttämällä tslint-kirjaston tarjoamia sääntöjä sekä pitämällä testikattavuus korkeana. Jokaisesta testatusta koodirivistä raportoitiin ja kehittäjällä tarjottiin selkeä palaute, jos koodista oli jäänyt rivejä testaamatta. Rajaksi kattavuudelle määriteltiin 80 %:n osuus koodin riveistä. Jotta uudet muutokset voitiin viedä julkaistavaan versioon, täytyi muutosten mennä läpi tuotantolinjasta. Tuotantolinja sisälsi CI-järjestelmän, jossa ajettiin yksikkötestaus, integraatiotestaus ja järjestelmätestaus.

CI-järjestelmä toteutettiin käyttäen Gitlab-CI:tä, jossa ajettiin testit Docker-ympäristössä. Testien eteneminen raportoitiin kehittäjälle ja kehittäjä pystyi reagoimaan, mikäli testien suorittamisessa ilmeni ongelmia. Palautteen saamisessa aikaa kului noin 5 minuuttia, jonka jälkeen testiraportit olivat luettavissa Gitlabin versionhallinnassa. Kuvissa 16 ja 17 on esimerkki onnistuneesta sekä epäonnistuneesta testiajosta.



Kuva 16. Onnistunut testiajo

Koska testit ajettiin tuotantolinjalla Docker-ympäristössä, pystyi käyttäjä suorittamaan samat testit myös omassa kehitysympäristössään ennen muutosten viemistä Gitlabin versionhallintaan, josta testit suoritettiin. Testien suorittamista varten määriteltiin package.json-tiedostoon valmiit komennot, jotta kehittäjien oli helppoa ja nopeaa saada palaute tehdyistä muutoksista.



Kuva 17. Epäonnistunut testiajo

Julkaisulinjaa ei luotu tässä vaiheessa, sillä se luodaan projektikohtaisesti asiakasvaatimusten perusteella, jotta sovellus pystytään viemään eri palvelutarjoajien palvelimille. Kaikille suurimmille pilvipalvelualustoille on kuitenkin mahdollista luoda samankaltainen Docker-ympäristössä ajettava palvelinkokonaisuus.

5 YHTEENVETO JA POHDINTA

Työn tavoitteena oli luoda Buutti Oy:lle web-sovelluspohja, jolla asiakasprojektien aloittaminen on sujuvampaa eikä ylimääräistä aikaa kulu alun kehitysympäristöjen asentamiseen. Työn tavoitteessa onnistuttiin ja lopputuloksena tuotettiin käyttötarkoitukseen soveltuva web-sovelluspohja, joka helpottaa asiakasprojektien aloittamista sekä ylläpitämistä. Toteuttamisessa käytettiin NodeJS-suoritusympäristöä ja sekä REST-API että käyttöliittymä toteutettiin käyttäen TypeScript-ohjelmointikieltä.

Työn toteutuksen aikana ongelmia muodostui Windows-ympäristössä suoritettavien komentojen ja skriptien kanssa sekä tiedostopäätteiden normalisoinnissa, joka onnistuttiin ratkaisemaan käyttäen apuna versionhallinnan tiedostopäätteiden automaattista normalisointia.

Projektipohjaa on käytetty jo muutamissa projekteissa, joiden aikana on huomattu muutamia kehityskohteita, joilla pohjaa voidaan parantaa. Esimerkiksi käyttöliittymän kuvaketta pitää pystyä pienentämään sitä varten paremmin soveltuvalla pohjakuvakkeella. Kehittäjien työskentelyn helpottamiseksi integraatiotestit täytyy pystyä ajamaan automaattiesti omassa ympäristössä, kun kirjoitetun testin lähdekoodi muuttuu. Shell-skriptit täytyy muuttaa Windows-ympäristössä ajettavaan muotoon sekä projektin eri osille täytyy lisätä automaattinen versiointi.

Työn aikana perehdyin Docker-ympäristöjen pystyttämiseen sekä kokonaisuudessaan syvennyin tarkemmin konttiteknologian käyttämiseen ohjelmistokehityksessä. Docker ja konttiteknologia soveltuivat erinomaisesti tämän kaltaisen projektin toteutukseen, sekä sen avulla pystyttiin toteuttamaan myös CI/CD-tuotantolinja, joka oli suoritettavissa myös kehittäjän omassa ympäristössä.

Gitlab-versionhallintajärjestelmä ja sen CI/CD-ympäristö olivat ennestään tuttuja, mutta en ollut aiemmin luonut valmista tuotantolinjaa, jossa kaikki tarvittavat testit suoritetaan. CI/CD-järjestelmä osoittautui helppokäyttöiseksi ja sen avulla oli helppoa luoda toimiva kokonaisuus, jolla kehittäjä sai palautetta lyhyessä ajassa.

Kokonaisuutena projektin aikana pääsin kehittämään osaamistani laajasti eri ohjelmistokehityksen osa-alueilla ja koen tästä olevan erittäin paljon hyötyä tulevalla ohjelmistokehittäjän uralla. Buutti

Oy tarjosi loistavan mahdollisuuden opetella teknologioita työn ohessa ja lopputuloksena käyttöön saatiin myös yritystä hyödyttävä kokonaisuus, joten työtä voidaan pitää onnistuneena.

LÄHTEET

1. Typescript documentation. 2020. Microsoft. Saatavissa: https://www.typescript-lang.org/docs/handbook/. Hakupäivä 2.11.2020.

2. Cherny, Boris 2019. Programmin TypeScript. Kalifornia: O'Reilly Media Inc. Saatavissa: https://learning.oreilly.com/library/view/programming-typescript/9781492037644/ (vaatii käyttöoikeuden). Hakupäivä 13.12.2020.

3. Meck, Bradley – Young, Alex – Cantelon, Mike 2017. Node.js in Action, Second Edition. New York: Manning Publications. Saatavissa: https://learning.oreilly.com/library/view/nodejs-in-action/9781617292576/ (vaatii käyttöoikeuden). Hakupäivä 13.12.2020.

4. Getting Started. 2020 React. Saatavissa: https://reactjs.org/docs/. Hakupäivä 14.11.2020.

5. Docker. Saatavissa: https://docker.com/. Hakupäivä 5.12.2020.

6. Docker documentation. 2020. Docker Inc. Saatavissa: https://docs.docker.com/. Hakupäivä 15.11.2020.

7. Caiazza, Alessio – Maeda, Shinya 2020. Nodejs.gitlab-ci.yml. Saatavissa: https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Nodejs.gitlab-ci.yml. Hakupäivä 16.11.2020.

8. Jansen, Remo H. 2019. Hands-On Functional Programmin with TypeScript. Birmingha: Packt Publishing. Saatavissa: https://learning.oreilly.com/library/view/hands-on-functional-programming/9781788831437/ (vaatii käyttöoikeuden). Hakupäivä 13.12.2020.