Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Timi Liljeström

Verkko-ohjelmointi MEAN-ohjelmisto-pinolla

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

12.3.2020

Tekijä Otsikko Sivumäärä Aika	Timi Liljeström Verkko-ohjelmointi MEAN-ohjelmistopinolla 36 sivua 12.3.2020
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Mobile Solutions
Ohjaajat	Palvelunjohtaja Katja Saarela Yliopettaja Kari Aaltonen

Verkkosivuja ja sovelluksia kehitettäessä käytetään yleensä jonkinlaista ohjelmistopinoa, joka mahdollistaa sovelluksen itsenäisen toiminnan ilman tarvetta muille ulkoisille sovelluksille. Insinöörityön tarkoituksena oli luoda kokonaan tyhjästä käyttöliittymä ja palvelin verkkosivustolle, jonka ohjelmistopinoksi valikoitui MEAN-ohjelmistopino. Ohjelmistopinon komponentit (MongoDB, Express.js, Angular, Node.js) ovat kaikki JavaScript-pohjaisia, ja tarkoituksena oli opiskella niiden toiminnallisuutta ja teknisiä puolia sekä raportoida sovelluskehityksen eri vaiheista ja tuoda esille ohjelmistopinon hyviä puolia, mikä mahdollisesti auttaisi valitsemaan sopivimman monista sovelluskehitykseen saatavilla olevista ohjelmistopinoista ja opettaisi full-stack-sovelluksen kehitysprosessia.

Työtä varten käytettiin monia eri lähteitä ja sovellettiin niiden tarjoamaa sisältöä, jotta kehitysprosessi sujuisi mahdollisimman nopeasti ja vaivattomasti. Insinöörityöraportissa on koodiesimerkkejä kuvastamassa työn eri vaiheita, jotka tehtiin ottamalla huomioon ajankohtaiset ja suositut käytännöt verkko-ohjelmoinnissa.

Yksi ohjelmistopinon suurimmista hyödyistä on sen yksikielisyys sekä komponenttien vahvuudet ja niiden valmiit sisäänrakennetut kirjastot, jotka tekevät ohjelmistopinosta helposti lähestyttävän ja sovelluskehitysprosessista nopean ja hyödyllisen.

Avainsanat	MEAN-ohjelmistopino, verkko-ohjelmointi, ohjelmointirajapinta

Author	Timi Liljeström
Title	Full-stack development with MEAN-stack
Number of Pages	36 pages
Date	12 March 2020
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Mobile Solutions
Instructors	Katja Saarela, Director Consulting Services Kari Aaltonen, Principal Lecturer

Web and application development process is often done with some sort of programming or software stack that facilitates the program's independency without the need for other outer programs for it to work. The goal for this project was to create a fully functional website (server-side and client-side) from scratch and focus on full-stack development process with MEAN-stack (MongoDB, Express.js, Angular, Node.js) of which components are all JavaScript based technologies. The project's purpose was also to study and teach the components' architecture, use cases, benefits and to report of the full-stack development process and its different phases to bring out the positive sides and possibly help to choose a technology stack from the many available ones out there in the market.

The project was made keeping an eye on popular and ever changing technologies and by reading and applying the information from different sources on the internet, the development process was much faster and easier. Coding was a big part of the process, which is why the work includes important snippets to explain some of the functionality.

Some of the main useful results that came up in the process was the ability to write programs in one programming language, and the components' strengths and built-in libraries that made the stack easily approachable and the development process fast and profitable.

Keywords	MEAN-stack, web development, API

Sisällys

Lyhenteet

MEAN	MongoDB, Express.js, Angular, Node.js. Ohjelmistopino, jota käytetään mm. verkkosovelluksien ohjelmointiin.
CLI	Command Line Interface. Komentoliittymä, jossa käyttäjä voi suorittaa komentoriviltä erilaisia suoritettavia toimintoja.
NPM	Node Package Manager. Paketinhallintatyökalu, joka mahdollistaa mm. JavaScript-kirjastojen asentamisen komentoriviltä.
JSON	JavaScript Object Notation. Yksinkertainen tiedonvälityksessä käytetty avoimen standardin tiedostomuoto.
SQL	Structured Query Language. Kyselykieli, jolla voidaan tehdä erilaisia hakuja, muutoksia ja lisäyksiä relaatiotietokantaan.
NoSQL	Not only Structured Query Language. Käsite, jolla kuvataan perinteisestä relaatiomallista poikkeavia tietokantoja.
HTTP	Hypertext Transfer Protocol. Protokolla, jota selaimet ja WWW-palvelimet käyttävät tiedonsiirtoon.
API	Application Programming Interface. Ohjelmointirajapinta, jonka avulla ohjelmat voivat tehdä erilaisia pyyntöjä ja vaihtaa tietoja keskenään.
HTML	Hypertext Markup Language. Avoimen standardin kuvauskieli, jota käytetään mm. internetsivujen sisällön kirjoittamiseen (otsikot, leipäteksti).
CSS	Cascading Style Sheets. WWW-dokumenteille kehitetty tyylilaji, jota käytetään dokumenttien esittämiseen.
DOM	Document Object Model. Tapa kuvata dokumenttien (esim. HTML:n) puun kaltainen rakenne objekteina, jotka kuvaavat tiettyä osaa dokumentissa.

SPA	Single Page Application. Sovellus, joka kommunikoi verkkoselaimen kanssa dynaamisesti päivittämällä vain tiettyjä osia verkkosivulla.
URL	Uniform Resource Locator. Merkkijono, jolla kerrotaan jonkin tietyn tiedon, kuten WWW-sivun, paikka tai osoite.
CRUD	Create, Read, Update, Delete. Perustoiminnot, joilla voidaan muokata sovelluksen tietokantaa, johon tiedot on varastoitu.
ODM	Object Data Model. Malli, jonka avulla voidaan kuvata erilaisia reaalimaailman entiteettejä ja niiden ominaisuuksia.

1 Johdanto

Verkkosivut ovat tärkeitä mm. tiedonhakuun ja -jakoon käytettäviä lähteitä tai työkaluja, joita suunniteltaessa ja kehittäessä voidaan käyttää monta eri lähestymistapaa tai ohjelmistopinoa. Suuri kirjo erilaisia teknologioita voi vaikeuttaa projektissa käytettävien työkalujen valintaa varsinkin, jos osaaminen rajoittuu vain harvaan teknologiaan tai ohjelmointikieleen. Tätä ongelmaa pyrkii ratkaisemaan suosiotaan kovasti kasvattava MEAN-ohjelmistopino.

MEAN-sana tulee komponenteista, jotka kaikki tukevat JavaScript-ohjelmointikieltä, eli MongoDB, Express.js, Angular ja Node.js. Ohjelmistopinoa käytetään usein full-stack kehityksessä, joka sisältää sekä palvelimen (back-end) että käyttöliittymän (front-end) kehityksen. Tämä tarkoittaa, että kyseisellä ohjelmistopinolla voidaan kehittää ohjelmistoja tai sovelluksia, kuten verkkosivuja, pelkästään JavaScript-pohjaisilla teknologioilla, mikä helpottaa kehittäjien työtä rajaamalla osaamisaluetta ja kehitystyötä yhteen ohjelmointikieleen.

Verkkosivut ja monet muutkin sovellukset voidaan yleensä jakaa kolmeen eri osaan, käyttöliittymään, palvelimeen ja tietokantaan. MEAN-ohjelmistopinossa kullakin sen komponentilla on tämänkaltaisille sovelluksille tärkeä tehtävä. MongoDB toimii sovelluksen tietokantana, Express.js-kehys helpottaa palvelimen kehitystä, Angular-kehys edistää käyttöliittymän kehitystä ja Node.js mahdollistaa JavaScript-ohjelmointikielen käytön palvelimella.

Tein opinnäytetyöni CGI:llä, ja tarkoituksena oli pyrkiä perehtymään lisää MEAN-ohjelmistopinon komponenttien käyttötarkoituksiin sekä kuvaamaan niiden toiminnollisuuksia, teknisiä ominaisuuksia ja hyötyjä. Tarkoituksena oli myös kehittää kokonaan tyhjästä verkkosivuston käyttöliittymä ja palvelin, jotta muodostuisi selkeä kuva sovelluskehityksestä kyseisellä ohjelmistopinolla. Opinnäytetyöraportissa kuvataan kehityksen tärkeät vaiheet ja pyritään kokoamaan hyödyllinen full-stack-sovelluskehityksen opas, johon kootaan työn vaiheista hyödyllisiä koodinpätkiä ja kuvia, joita voi mahdollisesti hyödyntää myöhemmissä projekteissa tai sovelluskehityksen opiskelussa.

2 Sovelluskehitys ja ohjelmistopinot

Ohjelmistopinolla tarkoitetaan yleensä joukkoa, joka koostuu erilaisista ohjelmistojärjestelmistä tai komponenteista, joita tarvitaan kokonaisen uuden alustan tai sovelluksen luomiseen. Nämä komponentit työskentelevät yhdessä muodostaen toimivan kokonaisuuden, jotta kehitetty alusta tai sovellus ei tarvitse muita ulkoisia sovelluksia toimiakseen. Verkkokehityksessä käytettyihin ohjelmistopinoihin sisältyy yleensä käyttöjärjestelmä, verkkopalvelin, tietokantapalvelin ja ohjelmointikieli. (1.)

Sovelluskehityksessä puhutaan yleensä käyttöliittymän kehityksestä (front-end) ja palvelinpuolen kehityksestä (back-end). Käyttöliittymän puolella hoidetaan vuorovaikutus sovellusta käyttävän käyttäjän kanssa ja palvelinpuolella puolestaan hoidetaan erilaiset toiminnollisuudet, kuten esimerkiksi yrityslogiikka, palvelimen isännöinti ja tietokantatoiminta. (2.)

Sovelluskehityksessä on mahdollisuutena käyttää useita eri ohjelmistopinoja, joista jokaisella on omat puolensa. Projektissa käytettyjä teknologioita valittaessa voi olla vaikeaa päättää, mikä olisi paras vaihtoehto kehitettävälle sovellukselle. Yleensä näitä valintoja tehdessä korostuu aikaisempi ohjelmointikokemus, sovelluksen ylläpidettävyys, skaalautuvuus ja turvallisuus. Lisäksi kehitettävän tuotteen tai sovelluksen kannalta on hyvä selvittää, että valittu ohjelmistopino tukee sovelluksen ominaisuuksia.

Verkkosivujen kehityksessä usein tärkeässä roolissa on tällä hetkellä hyvin suosittu JavaScript. Stack Overflow-verkkosivun tilastoissa JavaScript oli vuonna 2018 eniten käytetty ohjelmointikieli jo kuuden vuoden ajan (3). Tämän vuoksi projektin ohjelmistopinoksi valittiin MEAN-ohjelmistopino, jonka kaikki komponentit ovat JavaScript-pohjaisia. Tämä yksikielisyys mahdollistaa kehittäjien osallistumisen projektin eri osa-alueisiin, kuten käyttöliittymän ja palvelinpuolen kehitykseen. Yrityksillä tämä mahdollistaa esimerkiksi sen, että ohjelmoijia ei ole välttämätöntä palkata erikseen tekemään käyttöliittymän tai palvelinpuolen kehitystä. JavaScript on ennestään tuttu monelle kehittäjälle, joten mahdollisia MEAN-ohjelmistopinon osaajia pitäisi löytyä melko helposti ja siirtyminen MEAN-ohjelmistopinoon ei välttämättä vaadi yhtä paljon perehtymistä kuin kokonaan uuteen ohjelmointikieleen siirtyminen. Näiden ansiosta tiimin yhteistyö, projektin hallinta, laatu, käytetty aika ja kustannukset saattavat parantua jossain määrin. (4.)

3 MEAN-ohjelmistopino

MEAN-ohjelmistopinoa käytetään yleensä dynaamisten verkkosivujen tai -sovelluksien kehityksessä, ja sen MEAN-lyhennys tulee avoimen lähdekoodin komponenteista, jotka kaikki tukevat JavaScript-ohjelmointikieltä: MongoDB, Express, Angular ja Node.js. Ohjelmistopino mahdollistaa sovelluksen käyttöliittymän ja palvelinpuolen kehityksen yhdellä ohjelmointikielellä, ja sen tarkoituksena on tarjota yksinkertainen ja helppokäyttöinen alusta full-stack-JavaScript-sovelluksien kehitykselle. MEAN-sanan keksi Valeri Karpov blogipostauksessaan vuonna 2013. (5; 6.)

MEAN-ohjelmistopinon toiminnallisuus (kuva 1) on seuraavanlainen:

1. Käyttäjä tekee jonkin toiminnon käyttöliittymällä (Angular), minkä jälkeen toiminto käsitellään ja pyyntö lähetetään Node.js-palvelimelle.

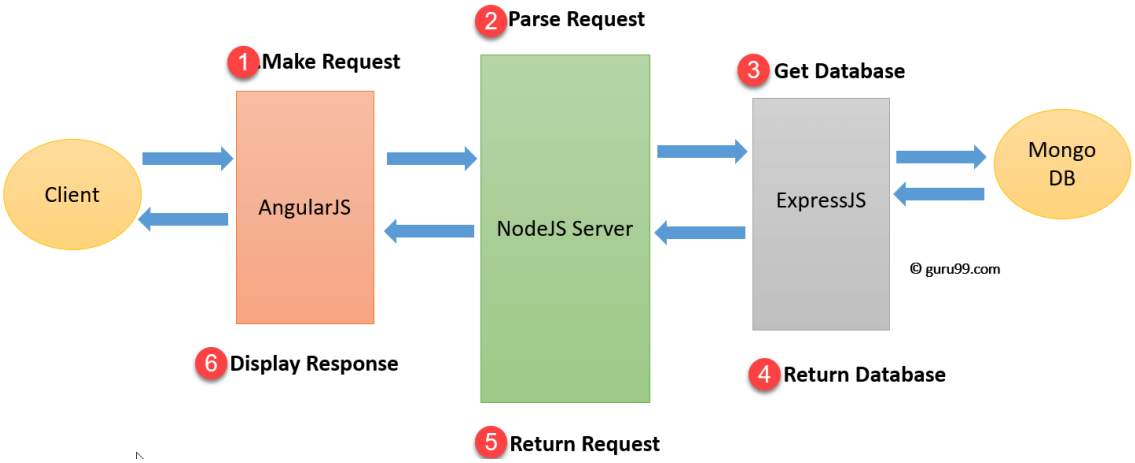
2. Node.js-palvelin vastaanottaa ja jäsentää pyynnön.

3. Express.js toimii väliohjelmistona tai rajapintana, jonka avulla tietokantahakuja ja muutoksia suoritetaan vastaanotetun pyynnön mukaan.

4. Tieto noudetaan MongoDB-tietokannasta ja palautetaan Express.js-väliohjelmiston käsiteltäväksi.

5. Node.js-palvelin palauttaa käyttöliittymältä lähetettyyn pyyntöön vastauksen, joka sisältää tehtyjen toimenpiteiden lopputuloksen.

6. Tietokannasta haettu ja vastauksessa palautunut tieto esitetään käyttäjälle käyttöliittymän puolella. (7.)



Kuva 1. MEAN-ohjelmistopinon arkkitehtuuri (7).

3.1 Node.js-runtime-ajoympäristö

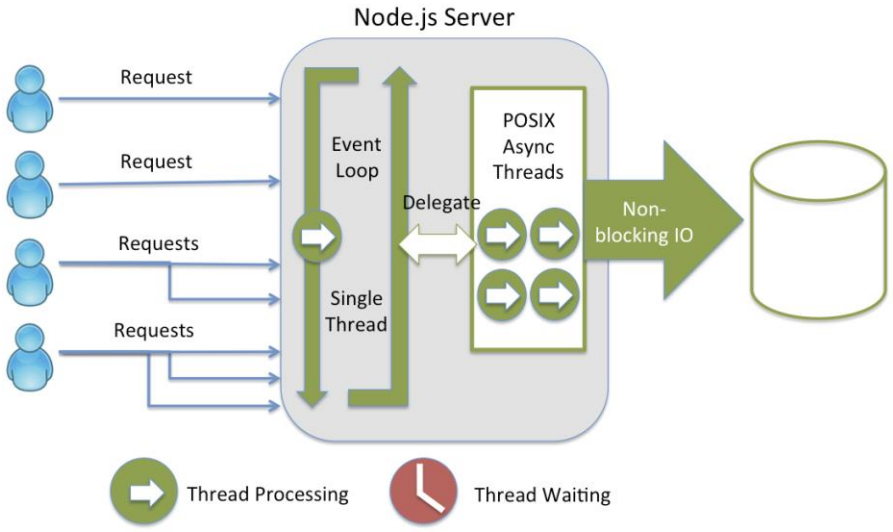
Node.js on JavaScript-koodille tarkoitettu alustariippumaton avoimen lähdekoodin runtime-ajoympäristö. Siinä JavaScript-koodi suoritetaan palvelimella eikä käyttäjän verkkoselaimessa, mikä on usein ollut tapana. Tämä mahdollistaa dynaamisten verkkosivujen tekemisen ennen niiden lähettämistä käyttäjän verkkoselaimeen. (8.)

Node.js:n avulla voidaan tehdä verkkosovelluksia pelkästään yhdellä ohjelmointikielellä, JavaScriptillä, mikä mahdollistaa esimerkiksi MEAN-ohjelmistopinon käytön. Node.js syntyi vuonna 2009 Ryan Dahlin toimesta, ja se on rakennettu Google Chromen V8 JavaScript -moottorille. V8 on kehitetty C++-ohjelmointikielellä, ja se kääntää JavaScript-lähdekoodin suoraan natiiviksi konekoodiksi käyttäen ajonaikaista kääntämistä sen sijaan, että tulkitsisi sitä etukäteen. (8; 9.)

Node.js:n tärkeitä ominaisuuksia:

- Asynkronisuus ja tapahtumapohjaisuus. Node.js-palvelimen ei tarvitse jäädä odottamaan saapuvaa tietoa (Event Loop, kuva 2) esimerkiksi ohjelmointirajapinnasta, vaan se siirtyy aina jonkin kutsun tehtyään seuraavaan tehtävään. Kun aikaisemmasta kutsusta on tullut vastaus, se tekee sille määrätyn toiminnon.
- Nopeus. Node.js on rakennettu Google Chromen V8 JavaScript -moottorille, ja tämän ansiosta se suorittaa koodin erittäin nopeasti.

- Yksisäikeisyys ja skaalautuvuus. Node.js käyttää tapahtumasilmukkaa yksisäikeisesti (Single Thread, kuva 2). Tämä auttaa palvelinta vastaamaan kutsuihin estämättä niitä ja tekee siitä erittäin skaalautuvan. Tämä eroaa perinteisistä palvelimista siten, että ne tekevät rajallisen määrän säikeitä kutsujen käsittelyyn. (8.)



Kuva 2. Node.js-palvelimen toiminnallisuus (10).

Asennus

Asennusohjelman saa ladattua Node.js:n omilta verkkosivuilta (https://nodejs.org/en/). Ohjelmaa ladattaessa valitaan operoiva käyttöjärjestelmä, ja kun se on suoritettu, Node.js on valmis käytettäväksi. Komentorivillä (Windows) voi suorittaa erilaisia komentoja (esimerkkikoodi 1), jotka ilmoittavat Node.js:n ja NPM-paketinhallintatyökalun onnistuneesta asennuksesta kertomalla niiden versionumerot.

```
> node -v
v12.13.0
> npm -v
6.12.0
```

Esimerkkikoodi 1. Komentorivillä suoritetut komennot kertovat Node.js:n ja NPM-paketinhallintatyökalun versiot.

Node Package Manager

Node Package Manager (NPM) on paketinhallintatyökalu, joka asentuu automaattisesti Node.js:n asentamisen yhteydessä. Se mahdollistaa ja helpottaa erilaisten JavaScript-kirjastojen ja -sovellusten asentamista suoraan komentoriviltä. Työkalulla asennetut paketit listautuvat automaattisesti tiedostoon nimeltä package.json, josta kehittäjä pystyy helposti tarkastelemaan, mitä paketteja milläkin versiolla projektissa on käytössä.

3.2 Angular-kehys

Angular on avoimen lähdekoodin verkkokehys, jota käytetään erilaisten käyttöliittymien kehityksessä (front-end development). Sen toiminta pohjautuu TypeScript-ohjelmointikieleen, jonka on kehittänyt ja jota ylläpitää Microsoft. TypeScript on JavaScriptin pääjoukko, joka tarjoaa lisäominaisuuksia, kuten olio-ohjelmointia ja vahvaa tyypitystä, joiden avulla koodin uudelleenkäyttö ja ylläpito helpottuvat. (11.)

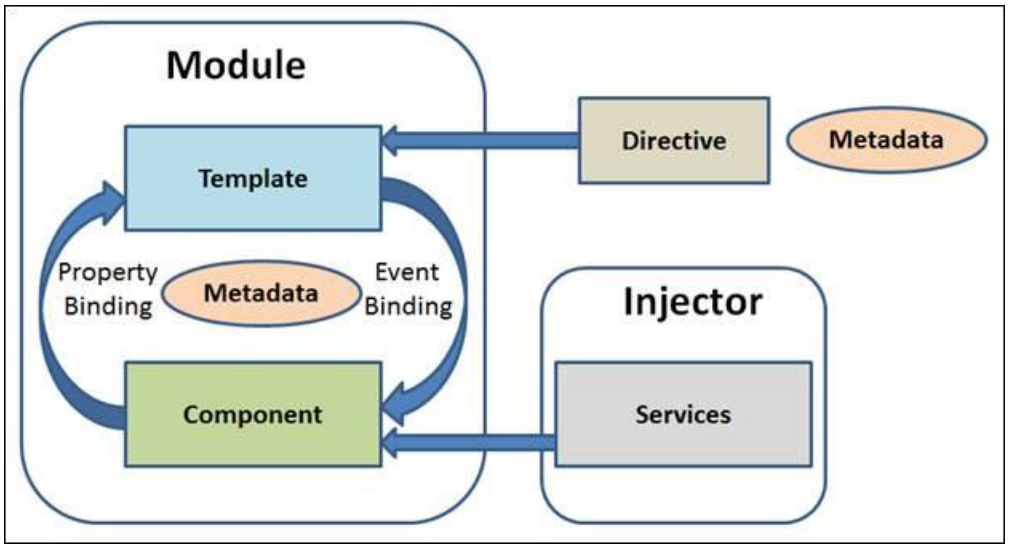
Angularista on kaksi eri versiota, Angular (tai Angular 2+) ja AngularJS, joita ei pidä sekoittaa keskenään niiden eroavaisuuksien takia. Angular on kokonaan uudelleen kirjoitettu aikaisemmasta versiosta AngularJS:stä, ja sen kehitystä johtaa Googlen Angular-tiimi. Ensimmäinen versio julkaistiin 14. syyskuuta 2016. (12.)

Sovelluksen rakennusvaiheessa TypeScript-koodi käännetään JavaScript-koodiksi, minkä jälkeen koodi ladataan ja suoritetaan käyttäjän verkkoselaimessa. Tämä mahdollistaa sen dynaamisen käyttöliittymän. (13.)

Angular-arkkitehtuurin (kuva 3) pääominaisuuksia ovat

- moduulit (Module), jotka ovat rakenteeltaan samanlaisia kuin ohjelmointiluokat ja joista löytyy jonkin tietyn tehtävän suorittamiseen tarvittava koodi
- komponentit (Component), joiden tehtävänä on vastata sovelluksen tietyn sivun tai osan logiikasta
- sivupohjat (Template), joiden tehtävänä on määritellä komponenttien ulkonäkö tai näkymä
- metatieto (Metadata), jota käytetään yleensä laajentamaan tietyn luokan toiminnallisuutta

- tietojen sitominen (Data Binding), joka on yksi tärkeimmistä ominaisuuksista ja joka toimii siltana mallin ja näkymän yhdistämiselle
- direktiivit (Directive), joita käytetään HTML:n tehostamiseen
- palvelut (Services), joita käytetään tiedon jakamiseen sovelluksessa, useimmiten silloin, kun jotain tiettyä toiminnallisuutta käytetään useassa eri moduulissa
- riippuvuusinjektio (Dependency Injection), jota käytetään lisäämään tai liittämään komponenttien toiminnollisuuksia ohjelman ajoaikana (13).



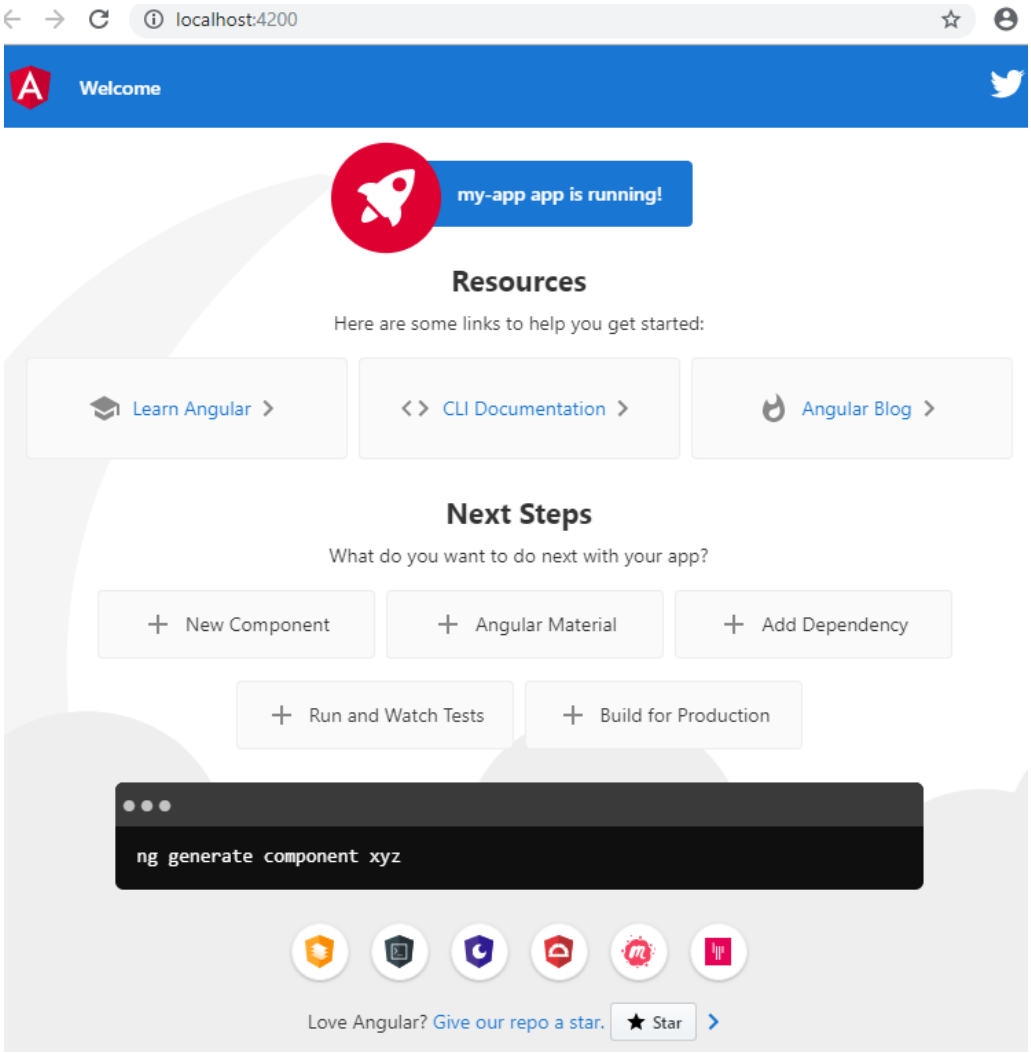
Kuva 3. Angular-arkkitehtuurin tärkeimmät ominaisuudet (13).

Käyttöönotto

Angular CLI-komentoliittymä on tärkeä työkalu Angular-sovelluksien kanssa työskentelemisessä. Sen mukana tulee komentoja, joiden avulla muun muassa projektin, sekä myöhemmin projektissa käsiteltävien palveluiden ja komponenttien, luonti helpottuu. Komentoliittymän asennus, projektin luonti ja sovelluksen kokoaminen (engl. build) ja käynnistys lokaalisti selaimelle (kuva 4) suoritetaan komentoriviltä (esimerkkikoodi 2).

```
> npm install -g @angular/cli
> ng new my-app
> cd my-app
> ng serve --open
```

Esimerkkikoodi 2. Angular-projektin luonti ja käyttöönotto.



Kuva 4. Uusi Angular-projekti käynnistettynä lokaalisti selaimelle (http://localhost:4200/).

3.3 MongoDB-tietokanta

MongoDB on yksi tämän hetken suosituimmista ja käytetyimmistä modernien sovelluksien tietokannoista. Se on alustariippumaton, eikä se ole sidonnainen mihinkään tiettyyn käyttöjärjestelmään tai laitteistoalustaan. Se on luokiteltu NoSQL-tietokantaohjelmistoksi, jonka käyttö perustuu tietokannassa sijaitseviin dokumentteihin, jotka tekevät tietokantakyselyistä erittäin skaalautuvia ja joustavia. (14; 15.)

Tieto varastoidaan JSON-tiedostomuodon (JavaScript Object Notation) tapaisissa dokumenteissa, mikä tarkoittaa, että dokumenttien eri kentät voivat vaihdella ja tietorakennetta voi muuttaa ajan myötä (15).

Esimerkkikoodeissa 3 ja 4 vertaillaan JSON-tiedostomuodon ja MongoDB-tiedostomuodon eroja. Koodi kuvaa Käyttäjä-dokumenttia, josta tulee ilmi henkilön eri tietoja, kuten nimi ja osoite.

```
{
      "firstName": "Timi",
      "lastName": "Liljeström",
      "address": {
            "streetAddress": "Example Address 123",
            "city": "Helsinki",
            "postalCode": "00100"
      }
}
```

Esimerkkikoodi 3. JSON-tiedostomuodossa oleva Käyttäjä-dokumentti.

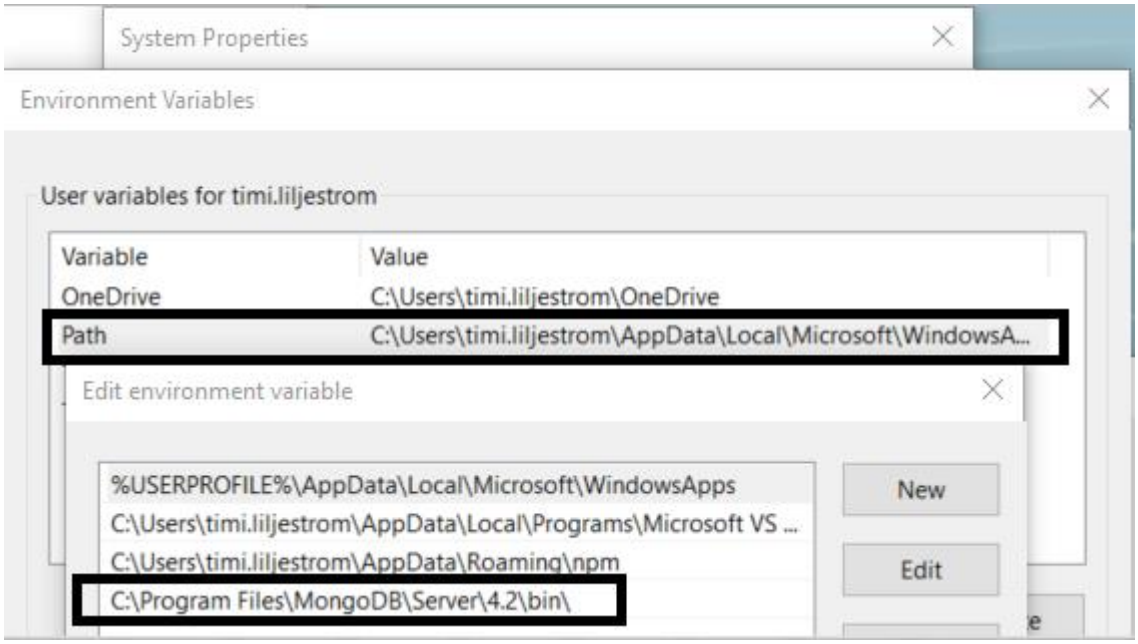
```
{
      _id: "5cf0029caff5056591b0ce7d",
      firstName: "Timi",
      lastName: "Liljeström",
      address: {
            streetAddress: "Example Address 123",
            city: "Helsinki",
            postalCode: "00100"
      }
}
```

Esimerkkikoodi 4. MongoDB-tiedostomuodossa oleva Käyttäjä-dokumentti.

MongoDB:n kehittäminen aloitettiin jo vuonna 2007, ja ensimmäinen versio tietokannasta julkaistiin vuonna 2009. Sen on kehittänyt yhdysvaltalainen yritys nimeltä MongoDB Inc., joka on aiemmin tunnettu myös nimellä 10gen. (16.)

Asennus

MongoDB-palvelin on ilmainen, ja sen asennetaan MongoDB:n verkkosivuilta (https://www.mongodb.com/download-center/community). Kun asennusohjelma on ladattu ja suoritettu, palvelimen ”bin”-kansion tiedostopolku tulee lisätä Windowsin ympäristömuuttujiin (kuva 5).



Kuva 5. MongoDB-palvelin lisättynä ympäristömuuttujiin.

3.4 Express-kehys

Express (tai Express.js) on minimaalinen ja joustava verkko-ohjelmoinnissa käytetty kehys Node.js-ajoympäristölle. Sitä käytetään muun muassa verkkosovelluksien ja ohjelmointirajapintojen (API) kehityksessä. (17.)

Express tarjoaa useita eri toimintoja, jotka tekevät verkkosovelluksien kehityksestä nopeampaa ja helpompaa kuin sovelluskehitys pelkästään Node,js:lla. MEAN-ohjelmistopinossa sen toiminta rajautuu lähinnä palvelinpuolen kehitykseen. (18.)

Expressin pääominaisuuksia:

- Väliohjelmistofunktioiden käyttö muun muassa HTTP-protokollan pyyntöjen ja vastauksien käsittelyn helpottamiseksi.
- Reitityksen määrittely erilaisiin toimintoihin HTTP-metodin tai URL-osoitteen mukaan.
- HTML-sivujen dynaaminen esittäminen annettujen argumenttien perusteella. (19.)

Esimerkkikoodi 5 kuvaa yksinkertaista Express.js-sovellusta, joka kuuntelee yhteyksiä portissa 3000 ja lähettää vastauksen ”Hello world”, jos pyyntö lähetetään /home-sivulle. Sovelluksen käynnistys testausta varten onnistuu komentoriviltä.

```
let express = require("express");
let app = express();

// middleware function (väliohjelmistofunktio)
app.use("/home", function(req, res, next){
     console.log("Request " + req.method + " " + req.url);
     next();
});

app.get("/home", function(req, res) {
     res.send("Hello world");
});

app.listen(3000);

// sovelluksen käynnistys komentoriviltä
> node app.js
```

Esimerkkikoodi 5. Yksinkertainen esimerkki Express.js-sovelluksesta (20; 21).

Expressin kehityksen aloitti vuonna 2009 TJ Holowaychuk, ja ensimmäinen versio julkaistiin 16. marraskuuta 2010. Vuonna 2014 oikeudet projektiin hankki ja pääkehittäjäksi vaihtui StrongLoop, joka myöhemmin siirtyi IBM:n alaiseksi yhtiöksi vuonna 2015. (22.)

Asennus

Expressin asentamiseen tarvitaan Node.js sekä NPM-paketinhallintatyökalu. Asennus tehdään komentoriviltä (esimerkkikoodi 6), minkä jälkeen Express listautuu automaattisesti package.json-tiedostoon, joka on luotu aikaisemmin projektin luonnin yhteydessä.

```
> npm install express
```

Esimerkkikoodi 6. Express.js asennetaan komentoriviltä NPM-paketinhallintatyökalun avulla.

4 Käyttöliittymän toteutus

Insinöörityöprojektin tärkeänä tavoitteena oli muun muassa opiskella JavaScript-pohjaisia MEAN-ohjelmistopinon teknologioita, perehtyä sovelluskehityksen elinkaareen sekä kehittää sovelluksen käyttöliittymä käyttäen jo valmiita kehyksiä tai kirjastoja, jotka helpottaisivat ja nopeuttaisivat prosessia. Käyttöliittymän toteutuksessa tärkeimpiä ja hyödyllisimpiä olivat Angular-kehys ja Bootstrap-kirjasto, jotka tarjosivat paljon valmiita elementtejä nopeuttamaan kehitystä.

Sivuston toiminnollisuuksin kuului monelle verkkosivustolle tuttuja ominaisuuksia, kuten navigoiminen usean näkymän välillä, käyttäjän rekisteröinti ja todentaminen, sisällön lisäys lähettämällä palvelimen rajapintaan erilaisia pyyntöjä tarvittavine tietoineen sekä lomakkeiden validointi ennen lähetystä.

4.1 Bootstrap-kehys

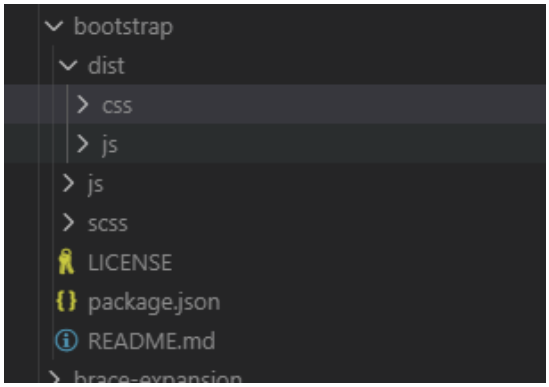
Käyttöliittymän toteutus aloitettiin näkymien suunnittelusta, ja koska sovelluksen kuului toimia kaikenkokoisilla laitteilla (engl. responsive web design), päätettiin kehityksessä käyttää Bootstrap-kehystä (https://getbootstrap.com/), joka soveltuu hyvin tämänkaltaisille mukautetuille verkkosivustoille.

Kehys on avoimeen lähdekoodiin perustuva HTML-, JS- ja CSS-kirjasto, jota käytetään käyttöliittymäkehityksen mukautuvassa, niin sanotussa ”mobile-first”, suunnitteluperiaatteessa. Se tarjoaa valmiita komponentteja ja tyylejä, joiden avulla sovelluksen ulkoasuun ei tarvinnut käyttää erityisen paljon aikaa.

Kehys on riippuvainen jQuerystä ja toisinaan Popperista, jotka asennettiin projektiin kirjastojen asennusta helpottavalla NPM-paketinhallintatyökalulla (esimerkkikoodi 7), joka lataa paketit automaattisesti projektin node_modules-kansion alle (kuva 6).

```
> npm install bootstrap jquery popper
```

Esimerkkikoodi 7. Bootstrapin, jQueryn ja Popperin asentaminen projektin juurikansioon onnistuu komentoriviltä NPM-paketinhallintatyökalun avulla.



Kuva 6. Asennettuna Bootstrap sijaitsee Angular-projektin node_modules-kansiossa.

Ladatut paketit sisältävät tiedostoja, jotka tarjoavat projektiin erilaisia toiminnollisuuksia (JS) ja tyylejä (CSS), mutta sovellus ei osaa löytää niitä ilman sijaintien määrittelyä projektin angular.json-tiedostossa (esimerkkikoodi 8).

```
// angular.json > projects > my-app > architect > build > options > styles
"styles": [
      "node_modules/bootstrap/dist/css/bootstrap.css",
      "src/styles.css"
],
// angular.json > projects > my-app > architect > build > options > scripts
"scripts": [
      "node_modules/jquery/dist/jquery.min.js",
      "node_modules/bootstrap/dist/js/bootstrap.min.js"
]
```

Esimerkkikoodi 8. Ladattujen pakettien tiedostosijainnit lisättynä angular.json-tiedostoon.

4.2 Angular-komponentit

Komponentit ovat Angular-sovelluksien tärkeimpiä rakennusosia, jotka kommunikoivat toistensa kanssa ja ovat yleensä jollain tavalla liitettynä sovelluksen hierarkian ylimpään HTML-tiedostoon (yleensä index.html), jonka kautta kaikkien komponenttien sisältö esitetään.

Komponentit koostuvat yleensä kolmesta eri osasta:

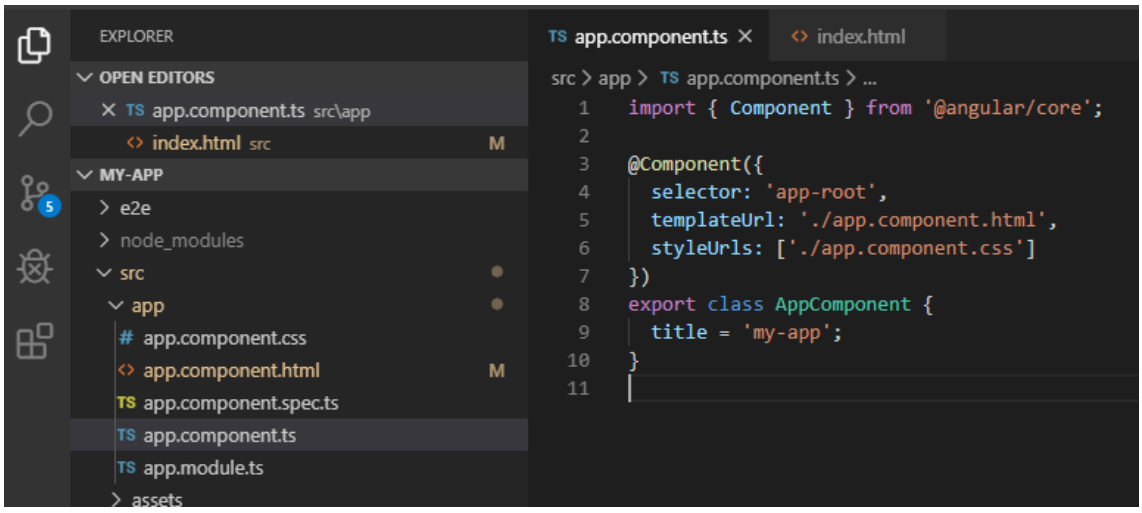
- HTML-mallista, jolla määritellään käyttöliittymän ja sen komponenttien rakenne

- CSS-mallista, jolla määritellään käyttöliittymän ja sen komponenttien tyylit, ts. ulkonäkö
- TypeScript-logiikasta, jonka avulla käyttöliittymän osia ja komponentteja (DOM) manipuloidaan (23).

Angular-sovelluksessa komponentit sijaitsevat yleensä src/app-kansiossa muodostaen puun kaltaisen hierarkian. Komponenttien käyttö mahdollistaa verkkosovelluksen hallinnoimisen rakenteellisesti, mikä tarkoittaa, että jokainen näytöllä tai sivulla oleva osa, kuten navigointipalkki, ylätunniste, runko ja alatunniste, voivat olla komponentteina omia kokonaisuuksiaan ja näin ollen sivulla olevia osioita voi päivittää eriaikaisesti ilman tarvetta päivittää koko sivua. Komponentteja voi myös käyttää uudelleen eri tarkoituksiin.

Jokaisen komponentin logiikka sijaitsee component.ts-tiedostossa, joka sisältää ainakin seuraavat koodinpätkät (kuva 7):

- import-lausekkeen, joka määrittelee komponentin
- komponentin sisustaja-objektin, joka mahdollistaa komponentin käytön muissa rakennetiedostoissa (selector) sekä kertoo sen rakenteeseen (templateUrl) ja tyyleihin (styleUrls) käytettyjen tiedostojen sijainnin
- export-lausekkeen, joka mahdollistaa komponentin käytön muiden komponenttien logiikkatiedostoissa.



Kuva 7. Angular-sovelluksen hierarkia ja app-komponentti.

Jokainen luotu komponentti on oma kokonaisuutensa, jonka toiminnallisuus tulee käytettäväksi muille komponenteilla vasta, kun se on lisätty sovelluksen NgModule-luokan jäseneksi (esimerkkikoodi 10). Tämä lisäys tapahtuu automaattisesti, jos komponentti luodaan Angular-komentoliittymän (CLI) avulla (esimerkkikoodi 9). Luonnin vaiheet ovat seuraavat:

- Komponentin nimeä vastaava alakansio luodaan app-kansion sisälle.
- Kansion sisälle luodaan komponentin HTML-tiedosto, CSS- tai SCSS-tiedosto sekä TypeScript-tiedostot logiikkaa ja testausta varten.
- Komentoliittymä päivittää sovelluksen app.module.ts-tiedostoa ja ilmoittaa uuden komponentin olemassaolosta (esimerkkikoodi 10).

```
> ng generate component posts
```

Esimerkkikoodi 9. Uuden komponentin luonti Angular CLI:n avulla.

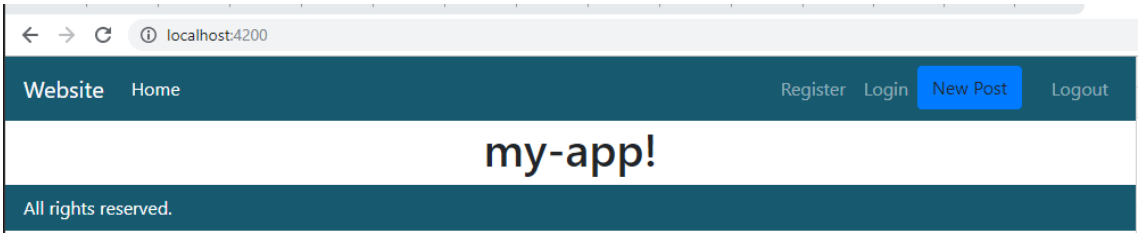
```
@NgModule({
  declarations: [
    AppComponent,
    PostsComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
```

Esimerkkikoodi 10. Angular-sovelluksen NgModule-luokan declarations-osaan listataan kaikki sovelluksessa käytetyt komponentit.

App-komponentti

Angular-sovelluksen kehitys aloitettiin App-komponentista lisäämällä sinne ylätunniste, alatunniste, sekä runko, johon suurin osa sivun muuttuvasta sisällöstä lisättiin (engl. Header, Footer, Main) (kuva 8).

App-komponentti on Angular-projektin luonnin yhteydessä automaattisesti luotu komponentti, joka toimii hierarkiassa ylimpänä juurikomponenttina ja johon viitataan aikaisemminkin mainitulla viittauksella (selector, app-root), joka sijaitsee uusissa projekteissa index.html-tiedostossa. Index.html-tiedosto on paikka, josta kehittäjän kannattaa aina aloittaa Angular-sovellukseen tutustuminen.



Kuva 8. App-komponentin ylätunniste, runko ja alatunniste. Tyyleihin on käytetty Bootstrap-kehystä.

Posts-komponentti

Sovelluksessa oli tarkoituksena esittää taulukon useampi osio etusivulla. Jokaisella osiolla olisi omat tietonsa, ja nämä osiot voisivat olla mitä tahansa, esimerkiksi kirjoja, matkoja, urheilutapahtumia, uutisia, elokuvia, jne. kuvineen ja teksteineen.

Taulukkoa varten tarvittiin objekteja, joissa kaikilla oli samat ominaisuudet, kuten otsikko, kuvaus ja kuvan URL-osoite (kuva 9). Tätä varten luotiin TypeScript-luokka (esimerkkikoodi 11), joka määrittelisi näiden objektien syntaksin ja jota objektien tai entiteettien olisi noudatettava. (24.)

```
export class Post {
    title: string
    description: string
    writer: string
    imageUrl: string
    added_at: Date
}
```

Esimerkkikoodi 11. TypeScriptillä luotu Post-luokka, joka sisältää otsikon, kuvauksen, kirjoittajan, kuvan URL-osoitteen ja lisäyspäivämäärän.

```
dogPost : Post = {
  _id: "36456",
  title: "Valokuva",
  description: "Koira on ihmisen kesyttämistä eläimistä vanhin. Kesyyntymisen alkuaikoina
  writer: "Tester",
  imageUrl: "https://picsum.photos/id/237/400/300",
  added_at: new Date
}

posts : Post[] = [this.travelPost, this.moviePost, this.dogPost]
```

Kuva 9. Post-luokan ominaisuudet sidottuna dogPost-objektiin ja posts-listaan lisättynä useita Post-objekteja.

Osioiden esittäminen sivulla onnistui helposti ngFor-direktiivillä, jonka toiminta perustuu for-silmukkaan. Silmukalle syötetty taulukko iteroidaan läpi, ja jokaisella iteroinnilla tiedot tallennetaan muuttujaan, johon pääsee HTML-mallissa käsiksi käyttäen interpolointia (esimerkkikoodi 12). Lopputuloksena taulukon jäsenistä luotiin omia HTML-elementte-jänsä ja osiansa, jotka esitettiin käyttäjälle.

```
<div class="card" *ngFor="let post of posts">
     <div>
          <img src="{{post.imageUrl}}">
     </div>
</div>
```

Esimerkkikoodi 12. Posts-taulukon iterointi ngFor-direktiivillä.

Muut komponentit

Sovelluksessa tarvittiin myös muita komponentteja, joista joissain käytettiin lomakkeiden validointia sekä kommunikoitiin sovelluksen palvelimen kanssa.

PostMore-komponentti sisälsi toiminnallisuuden yksittäiseen osioon. Käyttäjä tai sivun haltija voi halutessaan navigoida Posts-komponentin listassa olevista osioista yksittäiseen osioon ja katsoa tästä lisätietoja, jotka haettiin tietokannasta sivua ladattaessa. Komponentissa oli myös painikkeet PostForm-komponenttiin, jossa on toiminnallisuus osion muokkaukseen tai poistoon vain sivun ylläpitäjille.

PostForm-komponentissa kirjautuneella ylläpitäjällä oli mahdollisuus luoda uusi osio tai päivittää osion kenttiä lähettämällä lomakkeessa olevat tiedot palvelimen rajapintaan. Lomakkeessa oli jokaiselle osion tiedolle oma kenttänsä, johon käyttäjän tuli lisätä tiedot, jotka validoitiin ennen palvelimelle lähettämistä.

About-komponentti sisälsi tietoa verkkosivusta. Tiedot olivat staattisia ja tarvetta palvelimen kanssa kommunikoimiseen ei ollut. Tietoja olivat esimerkiksi verkkosivun tarkoitus, ylläpitäjän nimi, sähköposti, puhelinnumero ja profiilikuva.

LoginForm-komponentissa oli toiminta käyttäjän kirjautumiseen. Kirjauduttaessa tarkastettiin tietokannasta, että käyttäjä löytyi tietokannasta, sekä lisättiin selaimen varastoon eväste, jotta sovelluksen käyttäjän todentamista vaativat toiminnot oli mahdollista suorittaa. Tällainen toiminto olisi esimerkiksi uuden osion lisääminen ja muokkaus.

RegisterForm-komponenttia käytettiin uuden käyttäjän luontiin. Komponentissa oli lomake, jossa tiedot validoitiin, lähetettiin palvelimelle ja tallennettiin tietokantaan, jos käyttäjää samoilla tiedoilla ei ollut jo olemassa.

4.3 Reititys

Router-reititinkirjasto on yksi Angular-kehyksen ydinominaisuuksista ja se mahdollistaa SPA-sovelluksien rakentamisen useammilla näkymillä sekä navigoimisen niiden välillä. Se on tehokas JavaScript-reititin, jonka on rakentanut ja jota ylläpitää Angularin kehitystiimi.

Kirjasto asennetaan valmiiksi sovelluksen luonnin yhteydessä, mutta sen voi asentaa myös NPM-paketinhallintatyökalun avulla. Se mahdollistaa useamman reititindirektiivin (engl. router-outlet directive) käytön, erilaisia polkujen yhteensovittamisen strategioita, helposti käytettäviä reitittimen parametreja sekä reitittimen suojia suojaamaan komponentteja luvattomalta käytöltä. (25.)

Kirjastosta saatavilla oleva direktiivi (esimerkkikoodi 13) muuttuu dynaamisesti aina komponentille määriteltyä polkua kutsuttaessa. Toisin sanoen reititin sijoittaa näkymään komponentin, jonka polku vastaa annettua URL-osoitetta ja näin ollen komponentin sisältö esitetään käyttäjälle.

```
<router-outlet></router-outlet>
```

Esimerkkikoodi 13. Router-outlet-direktiivi, joka toimii paikanpitäjänä ja muuttuu dynaamisesti reitittimen tilan mukaan.

Direktiivi ei automaattisesti tunnista komponentteja vastaavia URL-osoitteita vaan tätä varten ne määritellään reittiobjekteja sisältävänä taulukkona (esimerkkikoodi 14). Nämä objektit muodostuvat vähintään

- polusta, joka viittaa ainutlaatuiseen URL-osoitteeseen
- Angular-komponentista, joka on yhteydessä polkuun ja joka esitetään näytöllä polkua kutsuttaessa (esim. http://localhost:4200/posts).

```
import { Routes } from "@angular/router"
import { PostsComponent } from './posts/posts.component'

export const routes : Routes = [
    {
        path: 'posts',
        component: PostsComponent
    },
]
```

Esimerkkikoodi 14. Angular-sovelluksen reititin.

Verkkosivuja kehitettäessä saattaa tulla vastaan tilanne, jossa osaksi polkua on liitettävä parametri, jota tarvitaan jotain tiettyä toimintoa varten (esimerkkikoodi 15). Tämmöinen tilanne voi olla esimerkiksi silloin, kun halutaan avata lisätiedot-sivu jostain tietystä osiosta ja sivua ladattaessa tarvitaan osion tunniste (id) erilaisiin tietokantahakuihin. Tämä onnistuu poimimalla aktiivisesta polusta osion parametri (esimerkkikoodi 16).

```
{
      path: 'posts/:id/more',
      component: PostMoreComponent
},
```

Esimerkkikoodi 15. Reitittimeen lisätty polku PostMore-komponentille, jossa parametrinä ”id”.

```
import { ActivatedRoute } from '@angular/router';

export class PostMoreComponent implements OnInit {
      constructor(private route: ActivatedRoute){}

      ngOnInit(){
            let postId = this.route.snapshot.params['id']
            this.postsService.getPost(postId).subscribe(data => {
                  ...
            })
    }
}
```

Esimerkkikoodi 16. PostMore-komponenttia ladattaessa aktiivisena olevasta polusta tallennetaan osion tunnisteen parametri (id) ja tehdään jatkotoimenpiteet.

Polkuja on hyvä suojata luvattomalta käytöltä, ja vastaava tilanne tulee usein vastaan sovelluksissa, joissa on käytössä käyttäjän todentaminen. Tästä on esimerkkinä sovellukseni LoginForm-komponentti, johon ei ollut syytä navigoida sisään kirjautuneena. Polun suojaaminen onnistui käyttämällä kirjaston valmista liitäntää (engl. CanActivate interface), jota käytetään usein käyttäjän todentamisessa. Tämä liitäntä antaa laajennuksen

kautta käyttöön canActivate()-metodin, joka pitää usein sisällään logiikan polun käytön sallimiseksi tai estämiseksi.

Kuva 10 ja esimerkkikoodi 17 havainnollistavat, miten kyseinen metodi estää pääsyn tiettyyn polkuun ja navigoi käyttäjän etusivulle, jos käyttäjä ei ole kirjautuneena.

```
import { UserService } from './user.service';
import { CanActivate, Router } from '@angular/router';
import { Injectable } from "@angular/core";

@Injectable()
export class AuthenticationGuard implements CanActivate {

    constructor(private router: Router){}

    canActivate(): boolean {
        if(UserService.isAuthenticated()) {
            return true
        }
        this.router.navigate(['/posts'])
        return false
    }
}
```

Kuva 10. AuthenticationGuard-palvelu, jota käytetään polun suojaamisessa.

```
{
      path: 'login',
      component: LoginFormComponent, canActivate: [AuthenticationGuard]
},
```

Esimerkkikoodi 17. LoginForm-komponentin polku suojattuna AuthenticationGuard-palvelulla.

Taulukossa komponenteille määritellyt polut ja reitittimen tarjoama dynaamisuus ovat käytettävissä vasta sen jälkeen, kun taulukko on lisätty sovelluksen moduulin jäseneksi (esimerkkikoodi 18).

```
import { routes } from './routes';
import { RouterModule } from '@angular/router';

@NgModule({
  imports: [
    RouterModule.forRoot(routes)
  ],
})
```

Esimerkkikoodi 18. Angular-sovelluksen reitittimen konfigurointi.

4.4 Lomakkeiden validointi

Ennen tiedon lähettämistä palvelimelle oli hyvä varmistaa, että kaikki tarvittavat tiedot olivat täytettynä ja oikeassa muodossa. Tämä helpotti tiedon käsittelyä myöhemmissä vaiheissa, kuten esimerkiksi palvelimen puolella. Käyttäjän puolella tehty validointi on tärkeä ominaisuus ja kuuluu hyvään käyttökokemukseen. Kun käyttäjää ohjeistetaan syöttämään oikeat tiedot, vähenevät virheet palvelimen puolella ja poistetaan turhien palvelinpyyntöjen tarve, mikä tekee sovelluksesta tehokkaamman. (26.) Lomakkeiden validointi helpottui huomattavasti kehyksen tarjoamalla sisäisellä kirjastolla (@angular/forms).

Angular-lomake on tyypillinen HTML-lomake, johon on lisätty uusia toiminnollisuuksia. Lomakkeen jokainen syötettävä kenttä tarvitsee FormControl-luokkaa olevan objektin. Tämä objekti antaa kentän tilasta erilaista tietoa, kuten onko siihen syötetty aiemmin tietoa ja ovatko syötetyt arvot sallittuja. Objekteista koostuu FormGroup-kokoelma, jollaista tarvitaan vähintään yksi jokaisessa lomakkeessa. Objektien luontiin on kaksi eri tapaa:

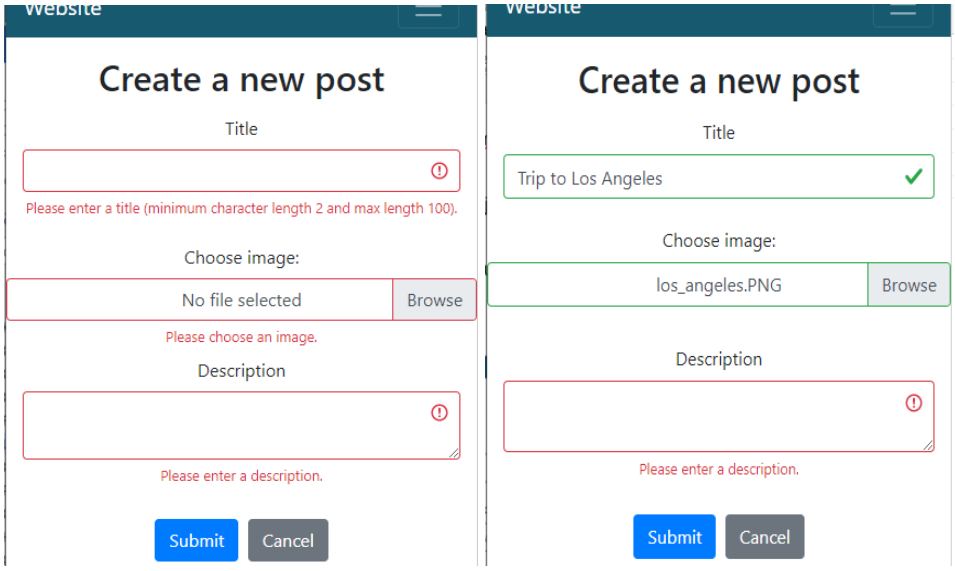
- reaktiivinen, jossa lomakkeen validointiin saa enemmän kontrollia luomalla objekteja manuaalisesti
- mallikäyttöinen, jossa objektit luodaan automaattisesti käyttämällä Angular-direktiivejä (esimerkkikoodi 19). (27.)

Sovelluksessa keskityttiin jälkimmäisenä mainittuun. Esimerkkikoodissa 19 lomake lähetetään parametrina TypeScript-tiedoston onSubmit()-metodiin käsiteltäväksi. Metodi tarkistaa, kelpaako lomake lähetettäväksi, ja tarvittaessa merkitsee syötettävät kentät jo käytetyiksi. Jokaiseen validoitavaan kenttään tulee lisätä direktiivi (ngModel) ja mahdolliset tarvittavat validoitavat kriteerit, kuten syötettävän tiedon tarpeellisuus sekä minimi- ja/tai maksimipituus. Jos lomakkeen kenttiin syötetyt arvot eivät vastaa validoitavia kriteerejä, on käyttäjälle hyvä antaa palautetta siitä, mikä on mennyt vikaan (kuva 11).

```
// HTML-tiedosto
<form #form="ngForm" (ngSubmit)="onSubmit(form)" [ngClass]="{'was-validat-
ed': !form.valid && !form.pristine}">
  <div class="form-group">
    <label for="ptitle">Title</label>
    <div>
      <input type="text" required class="form-control" minlength="2"
        maxlength="100" name="title" id="ptitle" ngModel>
      <div class="invalid-feedback">
      Please enter a title (minimum character length 2 and max length 100).
      </div>
    </div>
  </div>
  <div>
        <button type="submit" class="btn btn-primary">Submit</button>
        <button class="btn" (click)="onCancel()">Cancel</button>
   </div>
</form>

// TypeScript-tiedosto
onSubmit(form: NgForm){
    if(form.valid) {
        // form is valid and can now be submitted
        console.log("submit")
    } else {
        // mark every FormControl object as dirty
        for(let control in form.controls){
            form.controls[control].markAsDirty()
        }
    }
}
```

Esimerkkikoodi 19. Lomakkeen validointi @angular/forms-kirjastolla.



Kuva 11. Käyttäjälle on hyvä antaa lomaketta lähettäessä palautetta syöttökenttiin syötettävistä arvoista ja tietomuodoista.

4.5 Palvelut ja HTTP-pyynnöt

Angularissa palvelu on laaja kategoria, joka sisältää jonkin sovellukselle tarpeellisen toiminnon tai tehtävän. Ne voivat olla esimerkiksi luokkia, joille on annettu jokin tarkka tehtävä ja joita voidaan käyttää useammassa sovelluksen komponentissa helpottamaan tiedonhakua ja -jakoa. Palveluiden käyttö vähentää ylimääräistä koodia ja helpottaa koodin uudelleenkäyttöä. Yksi niiden tärkeistä tehtävistä voi olla esimerkiksi kommunikointi sovelluksen palvelimen kanssa. (28.)

Sovelluksessa tarkoituksena oli lisätä, hakea, muokata ja poistaa tietoa tietokannasta tekemällä HTTP-pyyntöjä Express.js-palvelimen ohjelmointirajapintaan. Itse tietokantamuutokset käsiteltiin palvelimen puolella. Pyyntöjä varten oli luotava uusi palvelu (esimerkkikoodi 20), jonka kautta pyyntöjen tekeminen helpottuisi ja joihin sovelluksen komponentit pääsivät helposti käsiksi.

```
> ng generate service posts
```

Esimerkkikoodi 20. Uuden palvelun luonti Angular CLI:n avulla.

Pyyntöjen lähettämisessä palvelimen rajapintaan tarvitaan Angularin sisäistä moduulia (esimerkkikoodi 21, @angular/common/http), joka tarjoaa toiminnallisuuden palvelimen kanssa kommunikointiin.

```
// app.module.ts
import { HttpClientModule } from '@angular/common/http';
...
@NgModule({
  ...
  imports: [
    BrowserModule,
    RouterModule.forRoot(routes),
    FormsModule,
    HttpClientModule
  ],
  ...
})
export class AppModule { }
```

Esimerkkikoodi 21. HttpClient-moduuli lisättynä sovelluksen NgModule-luokkaan.

Moduuli perustuu RxJS-nimiseen ohjelmointirajapintaan, mikä tarkoittaa, että se palauttaa aina pyynnön tehtyä havaittavan (engl. observable) tiedon, joka pitää tilata (subscribe), jotta vastauksen mukana tulevaan tietoon on mahdollista päästä käsiksi. Ilman tätä tilausta mitään ei tapahdu (29). Esimerkkikoodi 22 kuvastaa, miten pyyntö tehdään palvelimen rajapintaan käyttämällä moduulin GET-metodia ja miten sen jälkeen pyynnössä palautuva tieto tallennetaan taulukkoon.

```
// posts.service.ts
import { HttpClient } from '@angular/common/http'
...
constructor(private http: HttpClient){}
...
getAllPosts(){
      return this.http.get<Post[]>('http://localhost:8080/api/posts', Us-
erService.getHttpHeaders())
}

// posts.component.ts
getPosts(){
      this.postsService.getAllPosts().subscribe( data => this.posts = data)
}
```

Esimerkkikoodi 22. HttpClient-moduulin GET-metodilla lähetetty pyyntö ja sen tilaus.

User-palvelu

Joissain pyynnöissä oli tarpeellista käyttää käyttäjän todennusta, jotta tietokannan väärinkäytöltä vältyttäisiin. Tässä tuli mukaan User-palvelu, jonka tehtävänä oli mm. luoda uusi käyttäjä tietokantaan sekä kirjautua sisään tai ulos jo olemassa olevalta käyttäjältä päivittämällä selaimen evästeitä senhetkisen tilanteen mukaan.

Evästeisiin lisättiin palvelimella luotu ja sieltä palautettu autentikointitunnus käyttäjän kirjautuessa sisään ja poistettiin käyttäjän kirjautuessa ulos (esimerkkikoodi 23, login- ja logout-metodit). Autentikointitunnuksen avulla tarkistettiin, että käyttäjä oli kirjautuneena todentamista vaativia toimenpiteitä tehdessään (esimerkkikoodi 23, isAuthenticated- ja getHttpHeaders-metodit). Näitä toimenpiteitä olivat reitittimen polkujen suojaus sekä tiedon haku, luonti, päivitys ja poisto tietokannasta välittämällä palvelimelle tarvittavat tiedot GET-, POST-, PUT- ja DELETE-pyynnöillä ja HTTP-otsikkotietoja käyttäen.

```
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { map } from 'rxjs/operators'
import { User } from '../models/user';

export class UserService {

  constructor(private http: HttpClient){}

  static getHttpHeaders(){
    return {
      headers: new HttpHeaders({
        'Content-Type': 'application/json',
        'Authorization': `Bearer ${localStorage.getItem('auth-token')}`
      })
    }
  }

  static isAuthenticated(): Boolean {
    return localStorage.getItem('auth-token') !== null
  }

  // creating new user
  register(user: User): any {
    return this.http.post<User>('http://localhost:8080/api/users/register',
user, httpHeaders)
  }

  // login with user data
  login(user : User) : any{
    let res = this.http.post(`http://localhost:8080/api/users/login`, user,
httpHeaders)

    return res.pipe(
      map(data => {
        if (data['token']){
          // store the returned token to browser local storage
          localStorage.setItem('auth-token', data['token'])
          return true // login successful
        } else {
          return false // unable to login
        }
      })
    )
  }

  logout() : void {
    localStorage.removeItem('auth-token')
  }
}
```

Esimerkkikoodi 23. User-palvelu, joka on vastuussa käyttäjän todentamista vaativista toimenpiteistä.

Posts-palvelu

Posts-palvelun tehtäväksi jäi hoitaa kaikki osioita koskeva toiminta (kuva 12). Palvelun avulla komponenttien piti pystyä hakemaan, luomaan, päivittämään ja poistamaan tietoa

tietokannasta lähettämällä pyyntöjä palvelimen ohjelmointirajapintaan. Pyynnöissä käytettiin Angularin sisäistä moduulia, joka mahdollisti GET-, POST-, PUT- ja DELETE-pyyntöjen tekemisen.

Yksittäisen osion hakuun, muokkaukseen ja poistoon käytettiin osion tunnistetta (id), joka liitettiin reittimuuttujana osaksi pyynnön URL-osoitetta. Tarvittavat osion tiedot lähetettiin pyynnön mukana, ja pyynnössä käytetyn metodin ja URL-osoitteen mukaan palvelin suoritti tarvittavat tietokantatoiminnot.

```
export class PostsService {
    constructor(private http: HttpClient){}

    getPosts(){
        return this.http.get<Post[]>('http://localhost:8080/api/posts', UserService.getHttpHeaders())
    }

    getPost(id: string) : Observable<Post> {
        return this.http.get<Post>(`http://localhost:8080/api/posts/${id}`,  UserService.getHttpHeaders())
    }

    createPost(post : Post) : any {
        return this.http.post<Post>('http://localhost:8080/api/posts/create', post,  UserService.getHttpHeaders())
    }

    updatePost(post : Post) : any {
        return this.http.put<Post>(`http://localhost:8080/api/posts/${post._id}/update`, post, UserService.getHttpHeaders())
    }

    deletePost(post: Post) : any {
        return this.http.delete<Post>(`http://localhost:8080/api/posts/${post._id}/delete`, UserService.getHttpHeaders())
    }
}
```

Kuva 12. Posts-palvelu ja sen palvelimelle tekemät HTTP-pyynnöt.

5 Palvelinpuolen toteutus

Tarkoituksena oli kehittää sovelluksen palvelin ja sen ohjelmointirajapinta, joka vastasi käyttöliittymältä lähetettyihin pyyntöihin. Yksi sen tärkeistä tehtävistä oli hoitaa mm. tarvittavat tietokantamuutokset ja -haut, jotka suoritettuaan se palautti näiden lopputuloksen käyttöliittymälle. Käytetyt teknologiat olivat MongoDB tietokantana, Express.js-kehys helpottamassa rajapinnan kehitystä sekä Node.js mahdollistamassa JavaScript-ohjelmointikielen käytön palvelimella.

Palvelimen kehitystyö aloitettiin luomalla projektin juurikansioon app.js-tiedosto (kohta 2.4, esimerkkikoodi 5), joka oli ns. alkupiste ja jonka koodi suoritettiin aina palvelinta

käynnistettäessä. Tämä tiedosto luo Express-sovelluksen objektina ja määrittää sen erilaisilla asetuksilla ja väliohjelmilla. (30.)

Node.js-sovelluksia kehitettäessä myös package.json-tiedosto on välttämätöntä luoda (esimerkkikoodi 24), ja koska kyseessä on full-stack sovellus, tätä tiedostoa ei ole hyvä sekoittaa Angular-projektin samannimiseen tiedostoon tai säilyttää samassa kansiossa sen kanssa.

```
> npm init
```

Esimerkkikoodi 24. Uuden package.json-tiedoston luonti onnistuu NPM-paketinhallintatyökalun avulla.

Nodemon-kirjasto

Kehitysprosessissa huomattiin, että palvelimella tehdyt päivitykset eivät olleet käytettävissä ennen sen uudelleenkäynnistämistä, ja tämä hidasti kehitystä aina, kun se piti ajaa alas ja käynnistää uudelleen testausta varten. Toteutuksen nopeuttamiseksi prosessia päätettiin automatisoida asentamalla Nodemon-kirjasto (esimerkkikoodi 25), joka nopeuttaa Node.js-pohjaisten sovellusten kehitystä käynnistämällä sovelluksen uudestaan aina, kun tiedostomuutoksia on havaittavissa. (31.)

```
> npm install nodemon

> nodemon app.js
```

Esimerkkikoodi 25. Nodemon-kirjasto asennus NPM-paketinhallintatyökalun avulla sekä palvelimen käynnistys.

5.1 Tietokanta

Tarkoituksena oli, että käyttäjät ja ylläpitäjät pystyivät luomaan, hakemaan, päivittämään ja poistamaan käyttäjiin tai osioihin liittyvää tietoa sovelluksessa. Toiminta tapahtui HTTP-protokollan mukaan välittämällä tiedot käyttöliittymältä pyyntöjen mukana ja suorittamalla niiden mukaan tarvittavat tietokantatoiminnot (CRUD).

Tietokannaksi valikoitui hyvin skaalautuva MongoDB, joka jo valmiiksi asennettuna (kohta 3.3) toimi lokaalisti portissa 27017. Asennuksen yhteydessä tullut komentorivi on hyödyllinen työkalu tietokantakyselyjen suorittamisessa, ja sitä voi käyttää mm. uuden tietokannan luontiin, tietojen hakuun ja päivitykseen (esimerkkikoodi 26).

```
> mongo
MongoDB shell version v4.2.1
connecting to: mongodb://127.0.0.1:27017/...
...
use site_db
switched to db site_db
...
db.posts.insertOne({"title": "valokuva", "description": "Koira on ihmisen
kesyttämistä eläimistä vanhin.", "writer" : "Timi Liljeström", "imageUrl":
"https://picsum.photos/id/237/400/300", "addedAt":  ISODate("2019-12-
05T08:27:57.965Z")})
...
db.posts.insertMany([{"title": "valokuva", ...},{"title": "valokuva 2",
...}])
...
db.posts.find().pretty()
...
```

Esimerkkikoodi 26. Mongo-komentorivillä voi helposti testata asennetun MongoDB-palvelimen toiminnallisuutta ja tietokantatoimintoja.

Mongoose-kirjasto

Tietokantatyöskentelyä varten päätettiin asentaa tiedonmallinnuskirjasto (ODM) Mongoose (esimerkkikoodi 27), joka helpotti mm. sovelluksen tietokantakyselyjä sekä yhteyksien muodostamista (esimerkkikoodi 28). Sen tärkeimpinä tehtävinä oli hoitaa tietoyhteyksiä ja tarjota tietokantadokumenttien mallien validointia. Koodissa sitä käytettiin objektien kääntämiseen ja tietokannassa objektien esittämiseen. (32.)

```
> npm install mongoose
```

Esimerkkikoodi 27. Mongoose-kirjaston asennus NPM-paketinhallintatyökalun avulla.

```
let mongoose = require('mongoose')
const server = 'localhost:27017' // tietokannan palvelin
const database = 'site_db'       // tietokannan nimi

// yhteyden muodostaminen Mongo-tietokantaan käyttäen Mongoose-kirjastoa
mongoose.connect(`mongodb://${server}/${database}`, { useNewUrlParser:
true, useUnifiedTopology: true })

const db = mongoose.connection
// tapahtumat
db.on('connected', () => console.log('Connected to database'))
db.on('error', (err) => console.err('Connection error: ' + err))
db.on('disconnected', () => console.log('Disconnected'))
```

Esimerkkikoodi 28. Yhteyden muodostaminen Mongo-tietokantaan Mongoose-kirjaston avulla.

Suunnitellessa oli tärkeää miettiä etukäteen tietokannan dokumenttien mallit ja kaaviot (engl. model ja schema) käyttäjiä ja osioita varten. Niiden avulla oli helpompi saada käsitys siitä, mitä tietoa ja missä muodossa käyttäjistä ja osioista oli tarkoitus tallentaa. Käyttäjällä nämä tiedot olivat käyttäjänimi, sähköpostiosoite, etunimi, sukunimi sekä suojattu salasana. Osion tiedot olivat otsikko, kuvaus, kirjoittaja, kuvan URL-osoite sekä lisäyspäivämäärä. Tiedoista ei erikseen piirretty tietokantakaaviota, mutta tämä olisi ollut hyvä tehdä varsinkin, jos olisi isompi tietokanta kyseessä.

Tietokannan dokumenttien kanssa työskentelyä varten käytettiin kirjaston malleja, joissa jokainen käyttäjä tai osio olivat omia objektejansa. Mallit määriteltiin käyttämällä kirjaston kaavioliitäntää (Schema interface), joka mahdollisti objektin tai dokumentin tietokantaan tallennettavien kenttien määrittelyn validointikriteereineen ja tyyppeineen (esimerkkikoodi 29). (33.)

```
import mongoose from 'mongoose'
const Schema = mongoose.Schema;

// kaavion määrittely
let userSchema = new Schema({
    username: {
        type: String,
        // mahdollistaa uniikin käyttäjänimen
        unique: true,
        // required kertoo, että käyttäjänimi on pakollinen
        required: true,
        // poistaa tyhjät välilyönnit käyttäjänimen alusta ja lopusta
        trim: true
    },
    email: {
        ...
    },
    firstName: {
        ...
    },
    lastName: {
        ...
    },
    // käyttäjän salasanan salaukseen käytetyt arvot
    hash: String, salt: String,
})
```

Esimerkkikoodi 29. Käyttäjädokumenttien kenttien määrittely Mongoose-kirjaston kaavioliitäntää käyttäen.

Määrittelyn jälkeen mallit olivat kääntö- ja käyttövalmiita tietokantaoperaatioita varten (esimerkkikoodi 30). Kirjasto luo automaattisesti tietokantaan jokaisen mallin nimen mukaan dokumenteista koostuvan kokoelman, jossa dokumenttien kentät noudattavat kaavion määrittely- ja validointikriteerejä.

```
// Mallien luominen tapahtuu kääntämällä kaaviot (schema) malleiksi (model)
let Post = mongoose.model("Post", postSchema)
let User = mongoose.model("User", userSchema)

...
   // uuden osion luonti käyttämällä Post-mallia
   new Post({
      title: "Valokuva",
      description: "Koira on ihmisen kesyttämistä eläimistä vanhin.",
      writer: "Timi Liljeström",
      imageUrl: "https://picsum.photos/id/237/400/300",
      added_at: new Date})
      .save(err => {
         if(err){
            res.json({success:false, message:"Unable to save post."})
         } else {
            res.end()
         }
      })
...
   // osion haku käyttäen MongoDB:n osiolle automaattisesti luotua id:tä
   Post.findById(id, (err, post) => {
      // vastataan pyyntöön riippuen haun lopputuloksesta
      if(err) {
         res.json({success: false, message: "Query failed. Post not
found"})
      } else {
         res.write(JSON.stringify(post))
         res.end()
      }
   })
...
```

Esimerkkikoodi 30. Mongoose-mallien luonti sekä uuden Post-dokumentin luonti ja haku käyttäen Post-mallia.

5.2 Ohjelmointirajapinta

Käyttöliittymällä määritellyille (luku 4.5) ja sieltä palvelimen ohjelmointirajapintaan lähetetyille pyynnöille tuli lisätä palvelimelle reitit, joiden avulla suoritettiin erilaisia operaatioita, kuten tietokantamuutoksia ja -hakuja aina ohjelmointirajapinnan URL-osoitteita kutsuttaessa. Operaatiot suoritettiin käyttämällä apuna aikaisemmin luotuja malleja (luku 5.2), ja kun ne oli suoritettu, lopputulos palautettiin käyttöliittymälle, jossa palvelimelta palautunut tieto esitettiin käyttäjälle.

Ohjelmointirajapintaa oli helppo lähteä kehittämään, koska sen reittien tuli vastata käyttöliittymällä aikaisemmin määriteltyjä reittejä. Reittien luontiin on monta eri tapaa, mutta tässä päätettiin käyttää Expressin Router-väliohjelmistoa, joka mahdollisti GET-, POST-, PUT- ja DELETE-metodien yhdistämisen rajapinnan URL-osoitteisiin. Sovellukselle luotiin käyttäjille ja osioille omat kontrollerinsa, joihin reitit kategorisoitiin kullekin kuuluvan toimintansa mukaan.

Rajapintaa luodessa tarkoituksena oli tehdä kutsutuista osoitteista mahdollisimman selkeät ja informatiiviset. Esimerkkikoodissa 31 ovat havainnollistettuna rajapinnan osoitteet, joita kutsumalla haettiin muun muassa kaikkien osioiden tiedot lähettämällä käyttöliittymältä kutsu palvelimen osoitteeseen (”/api/posts”). Yhtä osiota haettaessa liitettiin kutsuun osion tunniste ja joissain tapauksissa, kuten muokkauksessa ja uuden luonnissa, kutsun rungossa (engl. body) lähetettiin tarvittavat lomakkeessa syötetyt tiedot.

```
// posts.controller.js
let router = express.Router()

// kaikkien osioiden haku
router.get('/api/posts', function(req, res, next){
      ...
})

// yhden osion haku tunnisteella
router.get('/api/posts/:id', function(req, res, next){
      ...
})

// uuden osion luonti
router.post('/api/posts/create', function(req, res, next){
      ...
})

// osion päivitys
router.put('/api/posts/:id/update', function(req, res, next){
      ...
})

// osion poisto
router.delete('/api/posts/:id/delete', function(req, res, next){
      ...
})

export {router}

// app.js
import { router as postsController } from '../app/controllers/posts.con-
troller'

app.use('/', postsController)
```

Esimerkkikoodi 31. Palvelimen ohjelmointirajapinnan URL-osoitteisiin yhdistetyt GET, POST-, PUT- ja DELETE-metodit.

6 Yhteenveto

Insinöörityössä perehdyttiin MEAN-ohjelmistopinoon ja verkkosivujen kehitykseen käyttämällä suhteellisen uusia, avoimeen lähdekoodiin perustuvia JavaScript-komponentteja, jotka tekevät ohjelmistopinosta suositun ja helposti lähestyttävän. Ohjelmistopinossa käytetty yksi ohjelmointikieli on suuri etu full-stack-kehittäjille, joilla on aikaisempaa kokemusta JavaScript-ohjelmoinnista. Tämän lisäksi JavaScript kerää koko ajan lisää suosiota ja on varmasti myös tulevaisuudessa hyvin ylläpidetty ja yksi suosituimmista ohjelmointikielistä.

Kiinnostuin aiheesta kuultuani siitä muutamista eri lähteistä, ja itse työtä pääsin toteuttamaan yrityksessä, jossa suoritin aiemmin työharjoitteluni ja nyt myös kyseisen opinnäytetyöni. Ohjelmistopinosta on saatavilla hyvin paljon erilaista tietoa, jota oli mielenkiintoista lukea ja soveltaa mikä helpotti opiskelua, työn edistymistä ja raportointia.

Verkkosivuston kehityksessä käytettiin myös muita ulkoisia kirjastoja ja teknologioita, jotka nopeuttivat kehitystä niiden helppokäyttöisyyden ja hyvän dokumentaation ansiosta. Niiden avulla käyttöön saatiin myös käyttäjäystävällisyyttä parantavia ominaisuuksia.

MEAN-ohjelmistopinolla työskentelyssä nousi esille monia hyviä puolia ja hyödyllisiä tuloksia, joista tärkeimpinä mielestäni nousivat esille seuraavat asiat:

Ohjelmointikieli (JavaScript)

- Ohjelmointikielen suuri suosio helpottaa osaajien löytämistä (yritysten kannalta).
- Yksi ohjelmointikieli rajaa tarvittavaa osaamisaluetta sekä kehitystyötä ja mahdollistaa kehittäjän osallistumisen full-stack-kehitykseen.
- Kehittäjiä ei tarvitse rekrytoida erikseen front- tai back-end kehitykseen.
- Yksi ohjelmointikieli helpottaa yhteistyötä, projektin hallintaa, laatua, käytettyä aikaa ja kustannuksia.

Kehitystä nopeuttava yksinkertaisuus ja helppokäyttöisyys

- Kirjastot ovat valmiiksi sisäänrakennettuina.
- Ulkoiset kirjastot asentuvat helposti NPM-paketinhallintatyökalun avulla.

- Komponenttien hyvä dokumentaatio tukee ja nopeuttaa uuden tiedon oppimista.
- Ohjelmistopinon nykyaikaisuus ja suosio takaavat monipuoliset lähteet ja tekevät siitä helposti lähestyttävän.

Komponenttikohtaiset vahvuudet

- Angular ja Node.js mahdollistavat dynaamisen sovelluksen.
- Arkkitehtuurinsa vuoksi Node.js suorittaa JavaScript koodin erittäin nopeasti.
- MongoDB ei ole sidonnainen johonkin tiettyyn käyttöjärjestelmään.
- Tietokantadokumentit tekevät kyselyistä skaalautuvia ja joustavia.
- Tiedon varastointi JSON-muodossa mahdollistaa tietokantarakenteen muutokset ajan myötä.
- Typescript-oliomaisuus helpottaa koodin uudelleenkäyttöä ja ylläpitoa.
- Express.js yksinkertaistaa rajapinnan kehitysprosessia.
- Angular-kehyksen valmiit kirjastot tukevat kehitystä ja testausta.
- Lähdekoodi on avoin ja käyttö maksutonta.

Lopputuloksena työssä syntyi suhteellisen lyhyessä ajassa toteutettu yksinkertainen verkkosivusto, jossa käyttäjät ja ylläpitäjät pystyivät rekisteröitymään ja lisäämään sivustolle erilaista sisältöä. Sellaisenaan verkkosivusto ei kuitenkaan ole täysin käyttökelpoinen, mutta hyvä lähtökohta ja suuntaa antava mahdolliseen jatkokehitykseen.

Lähteet

1 Beal, Vangie. 2007. Acronym Guide to Web Stacks. Verkkoaineisto. <https://www.webopedia.com/quick_ref/webstack_acronyms.asp>. Luettu 11.1.2020.

2 How to Pick the Right Web Technology Stack for Your Product. 2017. Verkkoaineisto. Hackernoon. <https://hackernoon.com/how-to-pick-the-right-web-technology-stack-for-your-product-f6d94440af2f>. 30.8.2017. Luettu 11.1.2020.

3 2018 Stack Overflow Developer Survey. 2018. Verkkoaineisto. Stack Overflow. <https://insights.stackoverflow.com/survey/2018>. Luettu 12.1.2020.

4 Patel, Ronak. 2019. Why Choose Mean Stack For Your Web & Mobile App Development Projects? Verkkoaineisto. <https://medium.com/@ronak8036/why-mean-stack-ec42aa82818>. 9.8.2019. Luettu 12.1.2020.

5 Dev Shankar, Ganguly. 2019. Most Popular Technology Stack To Choose From Full Stack Vs. MEAN Stack Vs. MERN Stack in 2019. Verkkoaineisto. <https://medium.com/datadriveninvestor/most-popular-technology-stack-to-choose-from-full-stack-vs-mean-stack-vs-mern-stack-in-2019-d12c0a17439a>. 3.3.2019. Luettu 7.10.2019.

6 MEAN Stack: An Essential Guide. 2019. Verkkoaineisto. International Business Machines Corporation. <https://www.ibm.com/cloud/learn/mean-stack-explained>. 9.5.2019. Luettu 25.9.2019.

7 What is Mean Stack Developer? Skills, Salary, Growth. 2019. Verkkoaineisto. Guru99. <https://www.guru99.com/mean-stack-developer.html>. 23.8.2019. Luettu 7.10.2019.

8 Node.js - Introduction. 2019. Verkkoaineisto. Tutorialspoint. <https://www.tutorialspoint.com/nodejs/nodejs_introduction.htm>. Luettu 17.10.2019.

9 Patel, Priyesh. 2018. What exactly is Node.js? Verkkoaineisto. <https://www.freecodecamp.org/news/what-exactly-is-node-js-ae36e97449f5/>. 18.4.2018. Luettu 16.10.2019.

10 Roth, Issac. 2014. What Makes Node.js Faster Than Java? Verkkoaineisto. <https://strongloop.com/strongblog/node-js-is-faster-than-java/>. 30.1.2014. Luettu 17.10.2019.

11 TypeScript - Overview. 2019. Verkkoaineisto. Tutorialspoint. <https://www.tutorialspoint.com/typescript/typescript_overview.htm>. Luettu 14.10.2019.

12 AngularJS vs Angular 2 vs Angular 4: What's the Difference? 2019. Verkkoaineisto. Guru99. <https://www.guru99.com/angularjs-1-vs-2-vs-4-vs-5-difference.html>.30.8.2019. Luettu 8.10.2019.

13 Shah, Miloni. 2017. What exactly is Angular 2? Verkkoaineisto. <https://www.quora.com/What-exactly-is-Angular-2>. 28.6.2017. Luettu 14.10.2019.

14 What is MongoDB? Introduction, Architecture, Features & Example. 2019. Verkkoaineisto. Guru99. <https://www.guru99.com/what-is-mongodb.html>. 3.9.2019. Luettu 8.10.2019.

15 What is MongoDB? 2019. Verkkoaineisto. MongoDB. <https://www.mongodb.com/what-is-mongodb>. Luettu 8.10.2019.

16 MongoDB History. 2019. Verkkoaineisto. Javatpoint. <https://www.javatpoint.com/mongodb-history>. Luettu 8.10.2019.

17 Express – Node.js web application framework. 2019. Verkkoaineisto. Express.js. <https://expressjs.com/>. Luettu 10.10.2019.

18 Express.js. 2019. Verkkoaineisto. Tutorials Teacher. <https://www.tutorialsteacher.com/nodejs/expressjs>. Luettu 10.10.2019.

19 Node.js – Express Framework. 2019. Verkkoaineisto. Tutorialspoint. <https://www.tutorialspoint.com/nodejs/nodejs_express_framework.htm>. Luettu 10.10.2019.

20 ExpressJS - Middleware. 2019. Verkkoaineisto. Tutorialspoint. <https://www.tutorialspoint.com/expressjs/expressjs_middleware.htm>. Luettu 13.10.2019.

21 Writing middleware for use in Express apps. 2019. Verkkoaineisto. Express.js. <https://expressjs.com/en/guide/writing-middleware.html>. Luettu 13.10.2019.

22 Tzur, Dor. 2016. The Unbelievable History of the Express JavaScript Framework. Verkkoaineisto. <https://thefullstack.xyz/history-express-javascript-framework>. 21.3.2016. Luettu 13.10.2019.

23 Lotanna, Nwose. 2018. Building a Simple Angular 7 App with Bootstrap 4 Styling. Verkkoaineisto. <https://codeburst.io/getting-started-with-angular-7-and-bootstrap-4-styling-6011b206080>. 29.11.2018. Luettu: 20.12.2019

24 TypeScript – Interfaces. 2019. Verkkoaineisto. Tutorialspoint. <https://www.tutorialspoint.com/typescript/typescript_interfaces.htm>. Luettu: 21.12.2019.

25 Bouchefra, Ahmed. 2018. A Complete Guide To Routing in Angular. Verkkoaineisto. <https://www.smashingmagazine.com/2018/11/a-complete-guide-to-routing-in-angular/>. 28.11.2018. Luettu: 22.12.2019.

26 Form data validation. 2019. Verkkoaineisto. MDN. <https://developer.mozilla.org/en-US/docs/Learn/HTML/Forms/Form_validation>. Luettu: 17.12.2019.

27 Ansari, Shadab. 2019. Angular Form Validation. Verkkoaineisto. <https://stackabuse.com/angular-form-validation/>. 2.7.2019. Luettu: 9.1.2020.

28 Introduction to services and dependency injection. 2019. Verkkoaineisto. Angular. <https://angular.io/guide/architecture-services>. Luettu: 10.1.2020.

29 Angular HTTP Client – Quickstart Guide. 2019. Verkkoaineisto. Angular University. <https://blog.angular-university.io/angular-http/>. 26.4.2019. Luettu: 14.1.2020.

30 Express Tutorial Part 2: Creating a skeleton website. 2019. MDN web docs. Verkkoaineisto. <https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/skeleton_website>. 13.12.2019. Luettu: 17.1.2020.

31 Nodemon. 2019. Npm. Verkkoaineisto. <https://www.npmjs.com/package/nodemon>. 11.12.2019. Luettu: 22.1.2020.

32 Karnik, Nick. 2018. Introduction to Mongoose for MongoDB. Verkkoaineisto. <https://www.freecodecamp.org/news/introduction-to-mongoose-for-mongodb-d2a7aa593c57/>. 11.2.2018. Luettu: 24.1.2020.

33 Express Tutorial Part 3: Using a Database (with Mongoose). 2020. MDN web docs. Verkkoaineisto. <https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/mongoose>. 28.1.2020. Luettu: 31.1.2020.