Jonne Borgman, Henri Uimonen

Web-sovellus taloyhtiön korjausvelan arvioimiseen

Metropolia Ammattikorkeakoulu
Insinööri (AMK)
Tieto- ja viestintätekniikka
Insinöörityö
1.11.2024

Tiivistelmä

Tekijä:	Jonne Borgman, Henri Uimonen
Otsikko:	Web-sovellus taloyhtiön korjausvelan arvioimiseen
Sivumäärä:	55 sivua
Aika:	1.11.2024
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja Viestintätekniikka
Ammatillinen pääaine:	Ohjelmistotuotanto
Ohjaajat:	Vesa Ollikainen, Lehtori

Insinöörityön tavoitteena oli suunnitella ja toteuttaa MVP-versio (Minimum Viable Product) full stack -verkkosovelluksesta, joka auttaa taloyhtiöiden korjausvelan määrittämisessä. Sovellus on suunnattu asunnon ostajille, sillä insinöörityön aikana markkinoilla oli tarjolla vain yksi työkalu korjausvelan laskemiseen. Kehitimme vaihtoehtoisen palvelun, joka tarjoaa samanlaista palvelua nopeammin ja edullisempaan hintaan.

Toisena tavoitteena oli tutustua Java Spring Boot -kehykseen ja kuvata käyttöönotto- ja kehitysprosessi lukijalle. Perustelimme teknologiavalinnan ja kävimme läpi eri asioita, joita tarvitsee luoda ja ottaa huomioon palvelinta kehittäessä. Käyttöliittymä toteutettiin React Native -teknologialla, ja tietokannaksi valitsimme MongoDB:n sen helppokäyttöisyyden vuoksi.

Loimme useita erilaisia näkymiä, ja pyrimme tekemään mahdollisimman selkeän käyttöliittymän, jotta käyttäjien tekemät virheet minimoitaisiin. Tavoitteena oli, että käyttäjät saisivat nopeasti käsityksen siitä, millaisen raportin he tulevat sovelluksella luomaan. Loimme etusivulle interaktiivisen malliraportin, jonka avulla käyttäjä voi tutustua raportin sisältöön ennen oman raportin luomista. Käyttäjälle generoidun raportin luomisprosessi on jaettu neljään vaiheeseen, jotta käyttäjältä ei kysytä liian paljon tietoja kerralla, ja käyttökokemus pysyy sujuvana.

Julkaisimme sovelluksen Azuren pilvipalveluun Azure Web Servicen alle ja rakensimme jatkuvan integraation putkiston toimimaan GitHub Actioneilla automaattisesti helpottaaksemme jatkokehitystä.

Avainsanat: Full stack, Spring boot, React-Native, MongoDB

Tämän opinnäytetyön alkuperä on tarkastettu Turnitin Originality Check -ohjelmalla.

Abstract

Authors:	Jonne Borgman, Henri Uimonen
Title:	Web Application for Estimation of Condominium Repair Debt
Number of Pages:	55 pages
Date:	1.11.2024
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Professional Major:	Software Development
Supervisor:	Vesa Ollikainen, Senior Lecturer

The goal of the study was to design and implement an MVP (Minimum Viable Product) version of a full-stack web application to help calculate the repair debt of housing companies. The application is targeted at home buyers, as during the time of writing, there was only one tool available on the market for calculating repair debt. We developed an alternative service that offers similar functionality at a faster pace and a more affordable price.

Another objective was to explore the Java Spring Boot framework and document the setup and development process for the reader. We justified the technology choices and discussed various aspects that need to be created and considered when developing a backend server. The frontend was built using React Native, and MongoDB was chosen as the database for its ease of use.

We designed multiple views and aimed to create a user-friendly interface to minimize user errors. The goal was for users to quickly understand the type of report they would generate with the application. To achieve this, we created an interactive sample report on the homepage, allowing users to preview the content of the report before generating their own. The report generation process is divided into four stages to prevent overwhelming the user with too many questions at once, ensuring a smooth user experience.

The application was deployed to the Azure cloud under Azure Web Services, and we built a continuous integration pipeline using GitHub Actions to automate and simplify future development.

Keywords: Full stack, Spring boot, React-Native, MongoDB

Sisällys

Lyhenteet

1 Johdanto 1

2 Sovelluksen määrittely 3

2.1 Mitä korjausvelka on? 4

2.2 Tekninen käyttöikä ja korjausvelka 5

2.3 Annuiteettilaina 5

2.4 Sivuston tavoitteet 6

3 Sovelluksen suunnittelu 6

3.1 Käyttötapaukset ja rakenne 7

3.2 Arkkitehtuurimallit 7

3.2.1 MVC 8

3.2.2 MVVM 9

3.2.3 MVP 11

3.2.4 MVC-, MVP- ja MVVM-arkkitehtuurien eroavaisuudet 13

3.3 Flux 14

3.4 Redux 15

3.5 Arkkitehtuurien vertailu 17

3.6 Valittu arkkitehtuuri 20

3.7 Valittu teknologia 22

4 Projektin toteutus 24

4.1 Taustapalvelu 25

4.1.1 Käyttöönotto 25

4.1.2 Arkkitehtuuri 28

4.1.3 Korjausvelan laskeminen 30

4.1.4 MongoDB 31

4.1.5 Autentikaatio 33

4.2 Sovelluksen käyttöliittymä 36
4.2.1 Sivuston etusivu 37
4.2.2 Hinnoittelu ja erilaiset suunnitelmat 37
4.2.3 Raportin luominen 38
4.2.4 Omat raportit 43
4.2.5 Raporttipohja 43
4.3 Azure 46
4.3.1 Azure Web Service 47
4.3.2 CI/CD 48
5 Jatkokehitys 50
6 Yhteenveto 51
Lähteet 53

Lyhenteet

API	Application programming interface. Komponenttien ja moduulien välinen ohjelmointirajapinta.
AsOy	Asunto-osakeyhtiö. Osakeyhtiö, jonka huoneistoista suurin osa on asumiskäytössä. Yleisimmin tunnettu nimityksellä taloyhtiö.
CRUD	Create, Read, Update ja Delete. Tiedonhallinnan neljä perusoperaatiota, joilla hallitaan dataa tietokannassa.
JRE	Java Runtime Enviroment. Java-ohjelmien suorittamiseen tarvittava ympäristö.
MVC	Model View Controller. Ohjelmistokehityksessä käytetty arkkitehtuurimalli.
MVP	Minimum Viable Product. Pienin toimiva tuote, joka määrittää varhaiset asetetut tavoitteet.
MVP (arkkitehtuurimalli)	Model View Presenter. Ohjelmistokehityksessä käytetty arkkitehtuurimalli.
MVVM	Model View Viewmodel. Ohjelmistokehityksessä käytetty arkkitehtuurimalli.
PTS	Pitkän tähtäimen suunnitelma. Suunnitelma kiinteistön tulevista remonteista ja huoltotoimenpiteistä.
REST	Representational state transfer. Http-protokollaan perustuva, rajapinnan toteuttamiseen käytettävä arkkitehtuurityyli.

1 Johdanto

Taloyhtiö on kiinteistön omistava yhteisö, joka koostuu yhdestä tai useammasta rakennuksesta ja niissä sijaitsevista asunnoista tai liiketiloista. Yleisimmin taloyhtiö on asunto-osakeyhtiö (AsOy), jossa osakkeenomistajat omistavat asuntonsa hallintaan oikeuttavia osakkeita (1).

Taloyhtiöllä on kunnossapitovastuu yhtiön kiinteistöstä, joka säännellään asunto-osakeyhtiölaissa (2). Se vastaa kiinteistön ja sen rakennusten ylläpidosta, korjauksista ja mahdollisista parannustöistä. Asukkaat osallistuvat taloyhtiön päätöksentekoon yhtiökokouksissa, joissa käsitellään muun muassa talousasioita ja tulevia remontteja.

Taloyhtiön hallitus vastaa kiinteistön ylläpidon ja tulevien korjausten suunnittelusta. Suunnittelun työkaluina käytetään kuntoarviota, jonka suorittaa ulkopuolinen ammattilainen. Kuntoarvion perusteella luodaan taloyhtiöön pitkän tähtäimen suunnitelma (PTS), joka on arvio kiinteistön ylläpidon ja korjausten tarpeista tulevien vuosien aikana (3). Se sisältää suunnitelman siitä, millaisia remontteja tai huoltotoimenpiteitä kiinteistössä tarvitaan sekä arvioidut kustannukset ja kattaa vähintään 5 vuoden ajanjakson.

Taloyhtiössä korjausvelka on rahamäärä, joka olisi pitänyt investoida kiinteistön kunnon ylläpitämiseen ja tulevien korjausten kattamiseen. Se kuvaa sitä, kuinka paljon rahaa taloyhtiöltä puuttuu, jotta kaikki kiinteistön osat olisivat hyvässä kunnossa. Lähes jokaisella rakennuksen osalla on oma rajoitettu käyttöikänsä, mikä edellyttää säännöllistä tarkastelua ja ennakoivaa huoltoa. (4.)

Taloyhtiöt yleensä rahoittavat ylläpidon ja korjauksen remontit lainalla. Finanssivalvonta huomasi vuonna 2017 yhtiölainojen kasvun, joka on osaksi noussut korjausrakentamisen voimakkaan kasvun takia. Viime aikoina varsinkin huonokuntoisilla taloyhtiöillä on ollut entistä vaikeampaa saada lainaa. (5.)

Lisäksi taloyhtiöiden konkurssit ovat viime aikoina olleet nousussa. Kun taloyhtiö nostaa lainaa, niin rahoitusvastike kasvaa. Mikäli vastikkeita jää maksamatta, taloyhtiö voi ajautua taloudellisiin vaikeuksiin, mikä voi johtaa konkurssiin ja osakkeiden arvon huomattavaan alenemiseen. (6.)

Useimmilla asunnon ostajilla ei ole tarpeeksi rakennusteknistä asiantuntemusta arvioidakseen itsenäisesti tulevia korjaustarpeita ja niiden kustannuksia. Esimerkiksi putkiremontin tai kylpyhuoneremontin kustannukset ja ajankohtaisuus osataan usein huomioida, mutta rakennuksessa on monia muitakin osia, jotka vaativat kunnossapitoa tai uusimista. (4.)

Insinöörityön tavoitteena on suunnitella ja toteuttaa verkkopalvelu, joka arvioi taloyhtiöiden sekä kiinteistöjen korjausvelkaa. Se on suunnattu yksityishenkilöille, ja erityisesti asunnon ostoa harkitseville. Tavoitteenamme on, että henkilö, joka on esimerkiksi hankkimassa ensiasuntoaan, pystyy vaivattomasti ja nopeasti saamaan tietoa taloyhtiön kunnosta sekä tulevista remonteista. Näiden tietojen avulla asunnon ostaja voi paremmin arvioida mahdollisia tulevia kustannuksia.

Monet ostajat tekevät päätöksiä puutteellisin tiedoin, mikä saattaa johtaa taloudellisiin vaikeuksiin, jos rahoitusvastikkeet yllättäen nousevat remonttien takia (4). Luomamme palvelu pyrkii ratkaisemaan tämän ongelman tarjoamalla yksinkertaisen ja helposti lähestyttävän tavan saada taloyhtiön korjausvelka laskettua, antaa alustavaa kuntoarviota taloyhtiöstä, ja näyttää laskettuna mahdollinen rahoitusvastikkeiden nousu tulevaisuudessa.

Koemme insinöörityön aiheen erittäin ajankohtaiseksi tämänhetkisen maailmantilanteen takia. Viime vuosina nousseet korot ovat lisänneet merkittävästi asuntolainojen ja rahoitusvastikkeiden kustannuksia. Myös rakennuskustannukset ovat nousseet noin 22 % vuodesta 2015, jossa suurin nousu on rakennusmateriaaleissa (7). Asunnon ostajien on entistä tärkeämpää olla tietoisia taloyhtiön mahdollisista tulevista korjauskustannuksista, jotta he voivat arvioida realistisesti omaa taloudellista tilannettaan.

Tämän lisäksi asunto-osakkeita on rakennettu huomattava määrä 70- ja 80-luvulla, joista suuri osuus on kerrostaloja, ja näissä alkaa osa talotekniikasta olemaan elämänsä lopussa (8). Tästä kertyy korjausvelkaa, joka johtaa tarpeeseen suorittaa mittavia ja kalliita remontteja lähitulevaisuudessa. Kun korot ovat korkealla, näiden remonttien rahoittaminen tulee sitä kalliimmaksi ja vaikeammaksi, mitä enemmän korjausvelkaa on.

Toisena tavoitteena on tutustuttaa lukija Java Spring Boot -kehykseen (9) ja sitä käyttäen palvelimen pystyttämiseen sekä kehittämiseen, ja kuvata lukijalle sen kehitysprosessia. Spring Boot on erittäin suosittu palvelimen kehitysalusta, joka tarjoaa erittäin tehokasta palvelintoimintaa ja yksinkertaistaa merkittävästi Java-pohjaisten sovellusten kehitysprosessia. Lisäksi se on tunnettu erinomaisena valintana mikropalvelu- ja pilviratkaisuissa, joka mahdollistaa skaalauksen myöhemmin projektin skaalan kasvaessa. Projektissa luomme tämän yhtenä palvelinkokonaisuutena, mutta tutkimme myös, mitä se vaatisi, jos muuttaisimme palvelun toimimaan mikropalveluina isommassa skaalassa.

Insinöörityö ja raportin laatiminen jakautuivat ryhmän jäsenten kesken tasapuolisesti. Projekti eteni sprinttien mukaisesti, ja haasteet jaettiin pienempiin kokonaisuuksiin, mikä kannusti molempia työskentelemään kaikissa projektin osissa. Tämä antoi meille molemmille mahdollisuuden oppia monipuolisesti uusia teknologioita ja niiden soveltamista käytännössä. Kumpikin vastasi oman osaamisalueensa syventämisestä, mikä mahdollisti myös tasavertaisen osallistumisen raportin kirjoittamiseen.

2 Sovelluksen määrittely

Nykyisessä tilanteessa asunnon ostaja saattaa olla täysin tietämätön taloyhtiön mahdollisesta korjausvelasta, mikä voi johtaa epämiellyttäviin yllätyksiin myöhemmin. Edullinen yhtiövastike saattaa vaikuttaa houkuttelevalta, mutta sen takana voi piillä karu todellisuus: taloyhtiön korjausvelka voi kasvaa merkittävästi. Tämä voi johtaa siihen, että alun perin kohtuulliselta vaikuttanut yhtiövastike nousee myöhemmin satoja euroja, mihin ostaja ei ole ollut varautunut.

Tällä hetkellä ostajilla on hyvin rajalliset mahdollisuudet selvittää korjausvelkaa. Ainoa yleisesti saatavilla oleva työkalu on Korjausvelkalaskuri Oy:n (10) tarjoama laskuri, jossa arvioinnin hinta on tällä hetkellä 60 euroa. Raportti voi olla saatavilla heti, jos kyseisestä kohteesta on jo olemassa aiempi arvio, tai vaihtoehtoisesti sen saaminen kestää 2–3 arkipäivää. Tämä tekee korjausvelan selvittämisestä aikaa vievää ja lisää kustannuksia asunnon ostoon liittyvässä päätöksenteossa.

2.1 Mitä korjausvelka on?

Korjausvelka on kiinteistöön kertyvä huoltotarvetta mittaava summa, joka antaa arviota siitä, kuinka paljon pitäisi käyttää rahaa kiinteistön kunnon palauttamiseen normaaliin tasoon (11). Jokaisessa kiinteistössä on korjausvelkaa, mutta sen suuruus riippuu monesta eri tekijästä, kuten teknisestä käyttöiästä, rakennustavasta tai käytetyistä materiaaleista. Kiinteistö voi näyttää ulkoisesti hyvältä, mutta esim. rakenteet voivat kulua ulospäin näkymättä, tai rakennuksessa voi olla piileviä rakennusvirheitä, jotka esiintyvät huoneistossa esimerkiksi sisäilmaongelmaa aiheuttavana haittana. (4.)

Korjausvelan määrä ei kuitenkaan suoraan anna arviota rakennuksen kunnosta, vaan se on arvio siitä, kuinka paljon rahaa keskimäärin tarvitaan sen peruskunnon palauttamiseen. Kiinteistö voi olla vähemmän kulunut kuin korjausvelka ennustaa, jolloin tarvittavat korjaukset ovat vähäisempiä, ja hoitamatta jätetyt tai viivästyneet remontit voivat aiheuttaa lisävahinkoja, kuten vesivahinkoja, jolloin velka on suurempi kuin korjausvelka arvioi (4).

Korjausvelkaa ei pidä käyttää ainoana työkaluna kiinteistön kunnon arvioimiseen, vaan suuntaa antavana mittarina sen kulumisesta. Taloyhtiön on tärkeää suunnitella tarkasti kiinteistön huoltaminen hyödyntämällä useita eri työkaluja, kuten ammattilaisen tekemää kuntoarviota, jotta rakenteiden todellinen kunto sekä huolto- tai korjaustarve selviävät.

2.2 Tekninen käyttöikä ja korjausvelka

Jokaisella kiinteistön osalla, kuten rakenteilla, järjestelmillä tai laitteilla, on tekninen käyttöikä, joka tarkoittaa aikaa, jonka ne pysyvät toimivina ja täyttävät niille asetetut vaatimukset. Kun tämä käyttöikä on kulunut umpeen, kyseinen osa on järkevää korvata uudella. (12.)

Tekninen käyttöikä perustuu kiinteistön ja sen tilanteen tietoihin, sekä siihen, kuinka kauan kiinteistön eri osat yleensä kestävät käytössä. Se ei ole tarkka, vaan yleistävä arvio. Kun osa, kuten putkisto tai katto, tarvitsee uusimista, usein samassa yhteydessä on taloudellisesti ja teknisesti järkevää uusia myös muita siihen liittyviä rakenteita. Esimerkiksi, jos talon salaojat uusitaan, samalla voi olla tarpeen vaihtaa maaperän täyttökerrokset, perustusten kosteuseristykset ja sadevesijärjestelmä. (12.)

Tekninen käyttöikä on olennainen tekijä sovelluksemme korjausvelan laskennassa, sillä sen avulla määritetään kiinteistön korjausvelka yhdessä neliöpohjaisen kustannusarvion kanssa. Näiden tietojen pohjalta luodaan tekniikan osista aikajana, josta voidaan nähdä, miltä korjausvelka on näyttänyt aiemmin ja miltä se näyttää nykytilanteessa.

2.3 Annuiteettilaina

Korjausvelan vähentämiseksi taloyhtiöiden tarvitsee tehdä korjauksia, ja koska taloyhtiöiltä ei yleensä ole suuria rahasäästöjä, niin nämä rahoitetaan yleensä yhtiölainalla (13). Annuiteettilaina on lainamuoto, jossa lainan alussa jokainen takaisinmaksuerä on yhtä suuri. Lainan maksuerään sisältyy alussa lainan lyhennys sekä korko.

Laina-ajan alussa pääosa maksuerästä koostuu korosta ja toinen osa itse lainan pääomasta. Maksuerien edetessä koron osuus pienenee, ja tällöin itse lainan lyhennyksen osuus kasvaa. Kaava 1 on annuiteettilainan kaava maksuerän selvittämiseksi, missä N on lainasumma, p on korkokanta prosentteina, ja n maksuerien määrä.

$$A = \frac{\left(1 + \frac{p}{100}\right)^n * \frac{p}{100}}{\left(1 + \frac{p}{100}\right)^n - 1} * N$$

Kaava 1. Annuiteettilainan maksuerän kaava.

Laskemme käyttäjälle mahdollisen annuiteettilainan suunniteltujen remonttien pohjalta. Tähän lainaan otetaan mukaan huoneistokohtainen neliömäärä, jotta käyttäjä näkee tarkemman summan kulujen noususta.

2.4 Sivuston tavoitteet

Tavoitteena on luoda yksinkertainen sivusto, jossa käyttäjillä olisi mahdollista luoda taloyhtiöistä raportteja sekä tarkastella edellisiä raportteja. Raportin luominen tapahtuu porrastetusti siten, että käyttäjältä kysytään tietoa taloyhtiöstä. Tiedot käyttäjä on saanut asuntoesittelystä saadusta isännöitsijäntodistuksesta. Kun käyttäjä on syöttänyt sivustolle tiedot, saa hän takaisin heti raportin, jonka voi sitten tallentaa omalle koneelle.

Haluamme luoda graafin, joka visualisoi simuloidun arvion kertyvästä korjausvelasta 10 vuotta eteenpäin. Tämä luku valittiin siksi, että taloyhtiöille suositellaan tekemään PTS, joka suositellaan suunnittelemaan 10 vuoden päähän tekohetkestä (14).

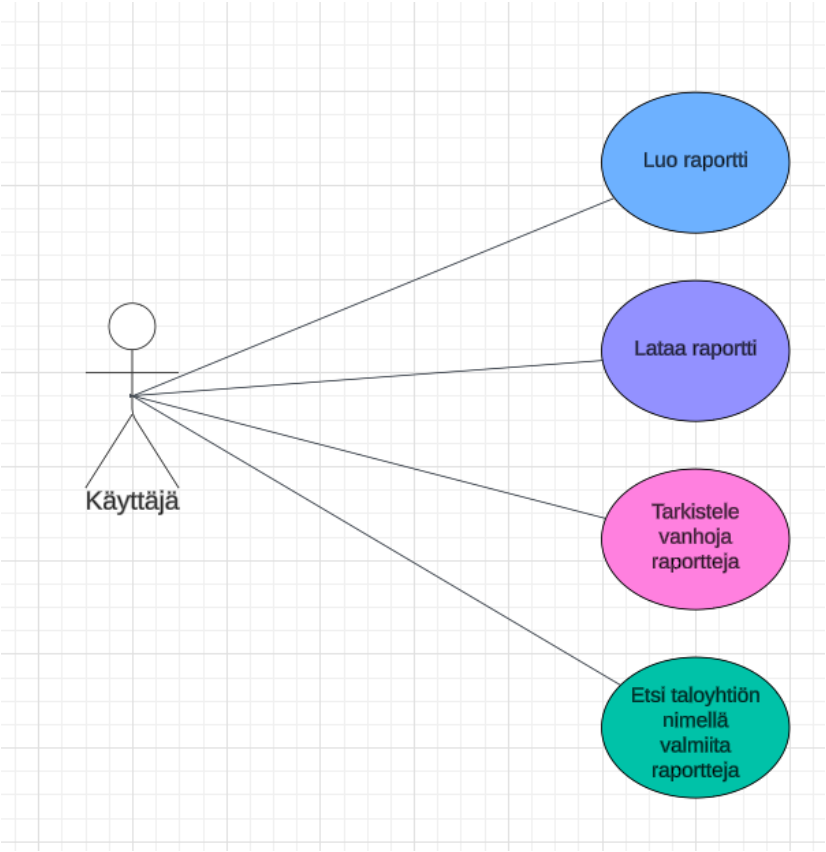
Lisäämme myös visualisointina graafin, joka näyttää käyttäjälle remonteista tulevan arvion vastikkeen noususta, joka johtuu korjausvelan vähenemisestä remonttien tapahtuessa. Tämän on tarkoitus antaa käyttäjälle selkeä kuva siitä, miten asumisen kustannukset mahdollisesti lisääntyvät tulevaisuudessa.

3 Sovelluksen suunnittelu

Ennen varsinaisen sovelluskehityksen aloittamista laadimme huolelliset suunnitelmat, jotka ohjasivat kehitystyötä. Määrittelimme keskeiset käyttötapaukset, arkkitehtuurivaihtoehdot sekä valittavat teknologiat. Seuraavaksi käymme läpi näitä osa-alueita tarkemmin.

3.1 Käyttötapaukset ja rakenne

Käyttäjällä on sovelluksessa neljä erilaista käyttötapausta (Kuva 1). Hän pystyy luomaan raportin, lataamaan raportin, tarkastella vanhoja raportteja sekä hakemaan valmiita raportteja taloyhtiön nimen perusteella. Käyttäjän tulee olla rekisteröitynä palveluun ennen käyttöä. Käyttötapauksia rajoitettaisiin tilin tyypin mukaan.



Kuva 1. Käyttötapausten määritys.

3.2 Arkkitehtuurimallit

Jotta sovellusta olisi helppo lähteä toteuttamaan, on sovellukselle valittava arkkitehtuuri, jonka mukaan sovellus rakennetaan. Ilman selkeää arkkitehtuuria ei sovelluksen kehityksestä tule selkeää ja helposti ylläpidettävää. Hyvän arkkitehtuurin valinta mahdollistaa vahvan perustuksen sovellukselle ja tekee kehittämisestä helpompaa. Seuraavissa kappaleissa tarkastelemme yleisimpiä Full stack

-sovellusten arkkitehtuurimalleja, jotta voimme valita tälle sovellukselle parhaiten soveltuvan arkkitehtuurin.

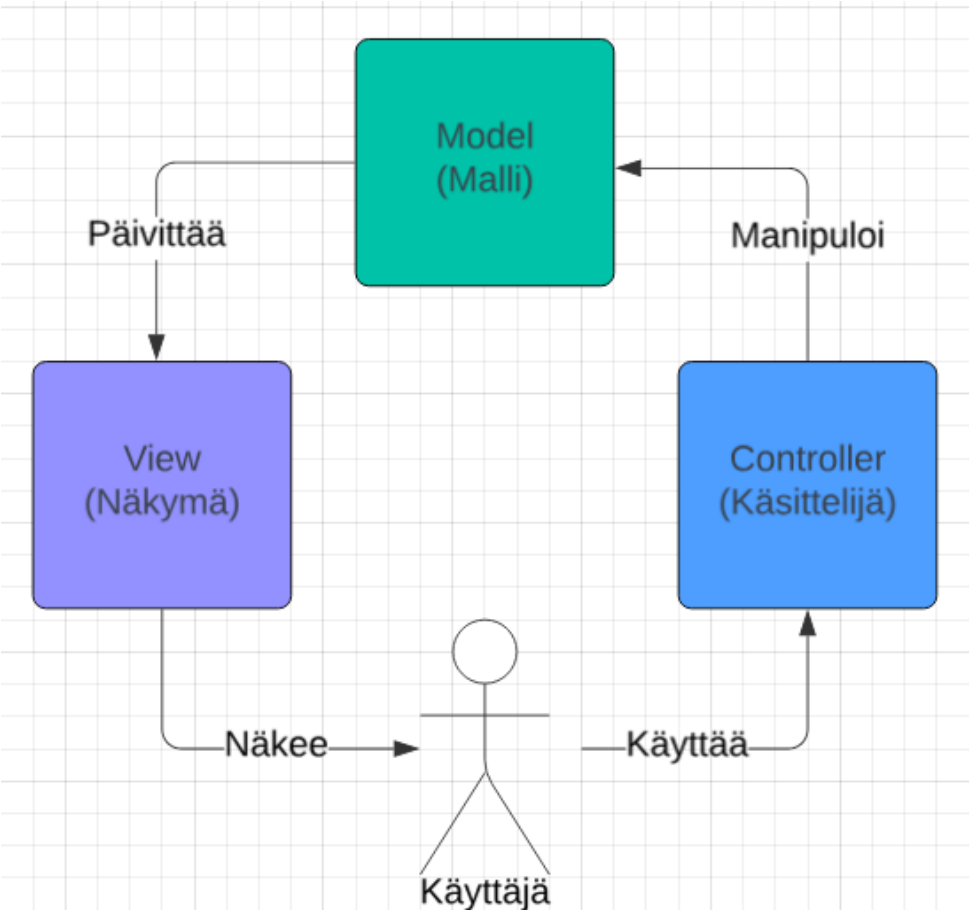
3.2.1 MVC

Model-View-Controller (MVC) on arkkitehtuurimalli, joka jakaa sovelluksen kolmeen pääkomponenttiin: malli, näkymä ja käsittelijä (15). Kuva 2 esittää ja kuvaa käyttäjän interaktiota MVC-mallin komponenttien kanssa. Nämä komponentit hoitavat seuraavat tehtävät:

Malli
Mallikomponentti vastaa sovelluksen datan ja liiketoimintalogiikan hallinnasta. Se käsittelee sovelluksen tietoja, soveltaa liiketoimintasääntöjä ja vastaa muiden komponenttien tietopyyntöihin.

Näkymä
Näkymäkomponentti edustaa sovelluksen käyttöliittymää. Se näyttää käyttäjälle mallikomponentista saapuvan datan ja tulokset. Näkymä keskustelee suoraan käsittelijäkomponentin kanssa, esimerkiksi välittämällä käyttäjän syötteet sille prosessoitavaksi. Näkymä on passiivinen komponentti, eikä se suoraan kommunikoi mallin kanssa.

Käsittelijä
Käsittelijä toimii välittäjänä mallin ja näkymän välillä. Se käsittelee käyttäjän syötteet, päivittää mallikomponentin vastaamaan näitä syötteitä, ja sen jälkeen päivittää näkymän heijastamaan malliin tehtyjä muutoksia. Käsittelijä sisältää sovelluslogiikkaa, kuten syötteen validoinnin ja tietojen muuntamisen tarvittavaan muotoon. (16)



Kuva 2. MVC-arkkitehtuuri.

3.2.2 MVVM

Model-View-ViewModel (MVVM) -arkkitehtuuri kehitettiin osittain ratkaisemaan MVC-mallin tiettyjä rajoituksia, erityisesti monimutkaisissa ja tietointensiivisissä sovelluksissa (17). Kuva 3 esittää komponentit ja käyttäjän interaktion komponenttien kanssa. MVVM-arkkitehtuuria käytetään usein sovelluksissa, joissa on rikas käyttöliittymä. Rikkaalla käyttöliittymällä tarkoitetaan sovellusta, joka on hyvin interaktiivinen, hyvin ja moderointisesti tyylitelty eikä sivustolla liikkuminen aiheuta sivun lataamista uudelleen. Arkkitehtuuria yleensä käytetään Angular (18) -pohjaisissa sivuissa.

MVVM-arkkitehtuuri koostuu seuraavista osista:

Malli
Komponentti edustaa sovelluksen dataa ja liiketoimintalogiikkaa samalla tavalla

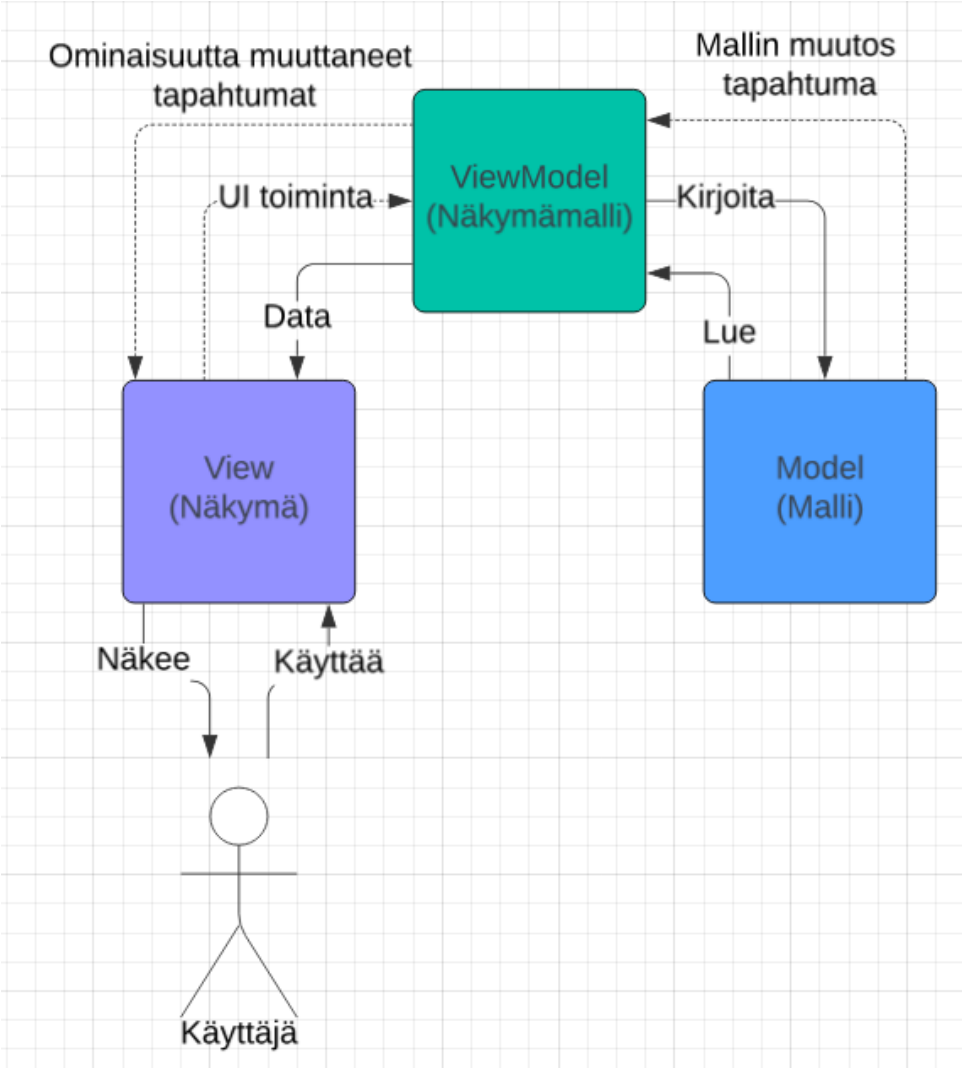
kuin MVC-arkkitehtuurissa. Se huolehtii sovelluksen tiedoista ja niiden käsittelystä.

Näkymä
Komponentti vastaa käyttäjälle esitettävien tietojen näyttämisestä ja käyttäjän vuorovaikutusten, kuten hiiren klikkausten tai tekstinsyötön, vastaanottamisesta. Nämä käyttäjän vuorovaikutukset välitetään näkymämallille tiedon sitomisen avulla. Toimii siltana mallin ja näkymämallin välillä.

Näkymämalli
Komponentti on näkymän abstrakti kerros, joka tarjoaa julkisia ominaisuuksia ja komentoja näkymän käyttöön. Verrattuna MVC-mallin käsittelijäkomponenttiin tai MVP-mallin esittäjäkomponenttiin, MVVM-mallissa on sitomismekanismi (binder), joka automatisoi tiedonsiirron näkymän ja sen sidottujen ominaisuuksien välillä. Tämä automatisointi poistaa tarpeen suoraan koodata logiikkaa näkymämallin ja näkymän synkronoimiseksi.

Suurin ero näkymämallin ja MVP-mallin esittäjän välillä on se, että esittäjällä on aina viittaus näkymäkomponenttiin, kun taas MVVM-mallissa tällaista viittausta ei ole. Sen sijaan näkymä on sidottu näkymämallin ominaisuuksiin, mikä mahdollistaa automaattisen päivitysten lähettämisen ja vastaanottamisen. Tämä vaatii yleensä tiedonsiirtotekniikoita tai sidontakoodin luomista.

Sitoja
Microsoftin Solution-arkkitehtuurissa sitojana toimii XAML-niminen merkintäkieli. Sitoja vapauttaa kehittäjän kirjoittamasta toistuvaa koodia, joka tarvitaan näkymämallin ja näkymän synkronoimiseksi. Kun MVVM-mallia toteutetaan Microsoftin ympäristön ulkopuolella, deklaratiivinen tiedonsidontateknologia on yleensä ainoa tapa mahdollistaa MVVM-mallin käyttö. Ilman sitojakomponenttia käytetään yleensä MVC- tai MVP-mallia välttääkseen ylimääräisen toistuvan koodin kirjoittamisen. (16.)



Kuva 3. MVVM-arkkitehtuuri.

3.2.3 MVP

Model-View-Presenter (MVP) on MVC-mallin johdannainen arkkitehtuurimalli, joka on suunniteltu käsittelemään esityslogiikkaa modulaarisella ja selkeällä tavalla (19). MVP-malli (Kuva 4) koostuu kolmesta pääkomponentista:

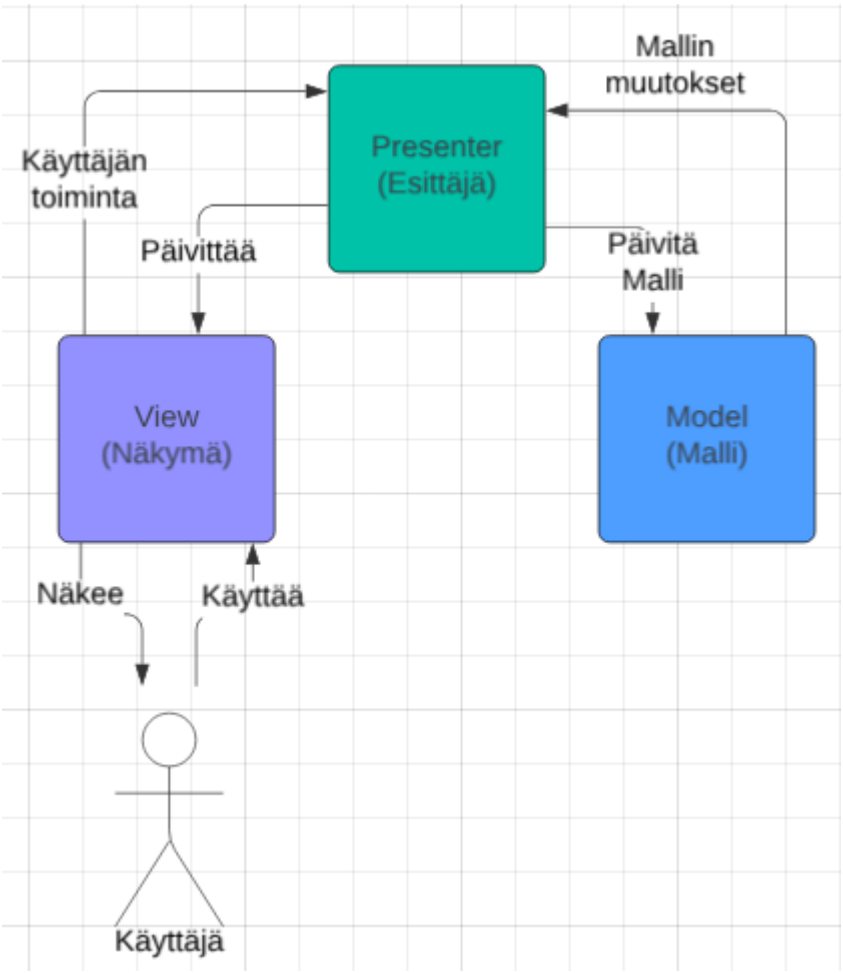
Malli
Komponentti vastaa sovelluksen datasta ja liiketoimintalogiikasta samalla tavalla kuin MVC- ja MVVM-arkkitehtuureissa. Se huolehtii tietojen käsittelystä ja liiketoimintasääntöjen toteuttamisesta.

Näkymä

Komponentti on passiivinen käyttöliittymä, joka näyttää käyttäjälle tuloksia ja reitittää käyttäjän toiminnot, kuten painikkeen painamisen tai tekstikentän täyttämisen esittäjäkomponentille. Näkymä delegoi mahdollisimman paljon vastuuta esittäjälle ja keskittyy vain tiedon esittämiseen.

Esittäjä

Komponentti toimii välittäjänä näkymän ja mallin välillä. Se käsittelee liiketoimintalogiikkaa, vastaanottaa tiedot mallista, ja päivittää näkymän mallista saatujen tulosten mukaisesti. Esittäjä vastaa myös käyttäjän vuorovaikutusten käsittelystä ja niiden perusteella tarvittavien toimintojen suorittamisesta. (16.)



Kuva 4. MVP-arkkitehtuuri.

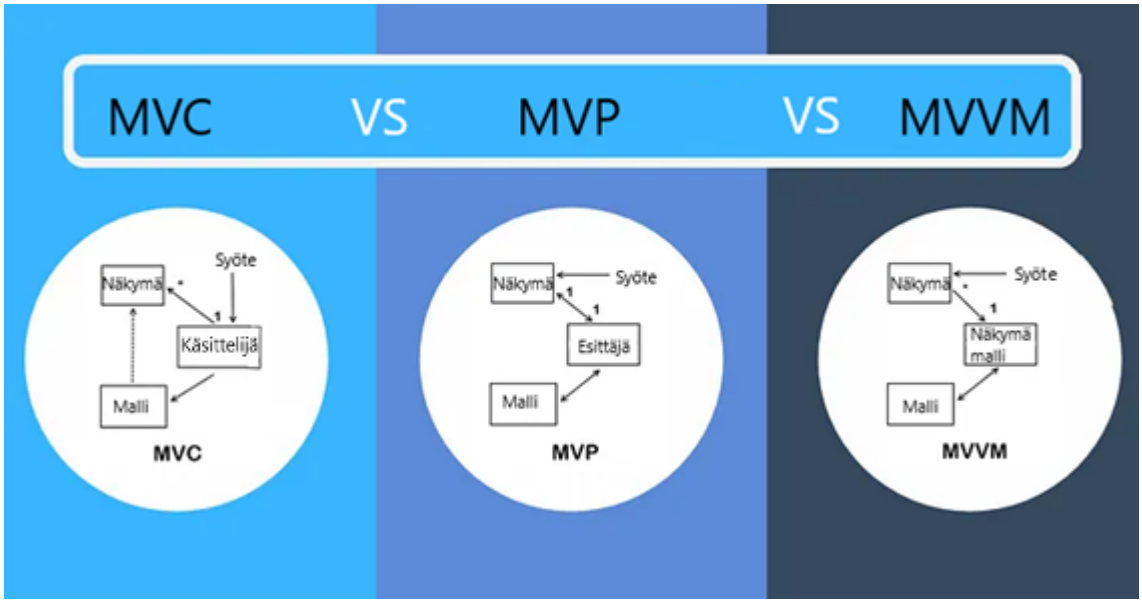
3.2.4 MVC-, MVP- ja MVVM-arkkitehtuurien eroavaisuudet

MVC-arkkitehtuurissa käsittelijä komponentti vastaa käyttäjän syötteen validoinnista, mallikomponentin päivittämisestä ja lopulta näkymän päivityksestä. Käsittelijä toimii siis aktiivisena välittäjänä käyttäjän ja sovelluksen logiikan välillä.

MVP-arkkitehtuurissa esittäjä komponentti hallitsee suurimman osan esityslogiikasta ja kommunikoi sekä näkymän että mallin kanssa, mikä tekee näkymästä mahdollisimman passiivisen. Esittäjä käsittelee kaikki käyttäjän toiminnot ja huolehtii näkymän päivittämisestä.

MVVM-arkkitehtuurissa näkymämallikomponentti paljastaa datan ja komennot näkymälle datasidonnan avulla, mikä mahdollistaa automaattisen päivityksen näkymälle, kun mallikomponentin tiedot muuttuvat. Tämä lähestymistapa vähentää näkymän suoraa vuorovaikutusta mallin kanssa sekä tarvetta luoda erillistä logiikkaa esityslogiikan ja datan synkronointiin. MVVM:n avulla sovelluksen käyttöliittymä ja logiikka pysyvät erillään toisistaan tehokkaammin kuin MVC- tai MVP-malleissa.

Kuva 5 tiivistää pääeroavaisuudet näiden kolmen arkkitehtuurin välillä helposti.



Kuva 5. Arkkitehtuurien erot. Suomennettu lähteestä 16.

3.3 Flux

Flux (20) on Facebookin (nykyisin Meta) kehittämä sovellusarkkitehtuuri, jota käytetään asiakaspuolen verkkosovellusten rakentamiseen. Flux hyödyntää Reactin koostettavia näkymäkomponentteja ja perustuu yksisuuntaiseen tietovirtaan.

Flux-arkkitehtuurissa on neljä pääkomponenttia (Kuva 6):

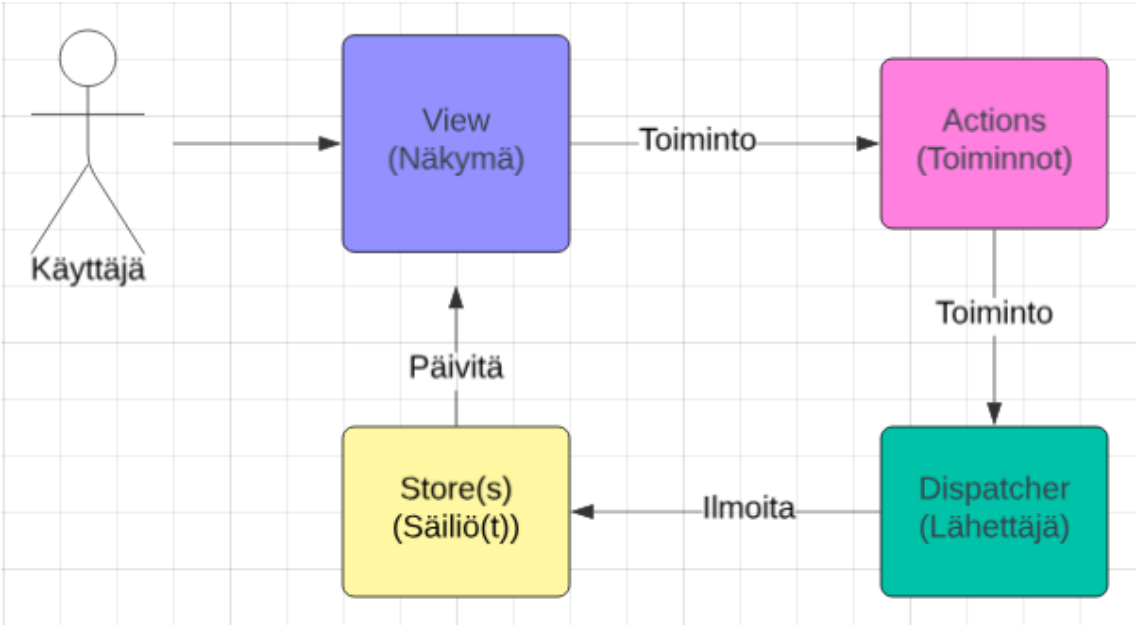
Näkymä
Komponentit ovat käyttöliittymän osia, jotka kuuntelevat säiliö komponentteja ja päivittävät näkymää, kun sovelluksen tila muuttuu. Nämä komponentit ovat yleensä React-komponentteja, jotka tilaavat tietyn säiliön ja reagoivat sen lähettämiin muutostapahtumiin.

Säiliö
Komponentit hallitsevat sovelluksen tilaa ja logiikkaa. Jokainen säiliö vastaa tietystä osasta sovelluksen tilaa ja kuuntelee toimintakutsuja, joiden perusteella se päivittää tilan.

Lähettäjä
Tämä komponentti on keskuskomponentti, joka hallinnoi kaikkia sovelluksen toimintoja. Se vastaa siitä, että kaikki säiliöt vastaanottavat toimintakutsut koordinoidusti, mikä varmistaa yksisuuntaisen tietovirran ja konsistentin tilan päivityksen.

Toiminnot
Nämä komponentit ovat objekteja tai funktioita, jotka kuvaavat sovelluksessa tapahtuvia muutoksia tai käyttäjän toimia. Nämä toiminnot lähetetään lähettäjäkomponentin kautta säiliöihin, jotka sitten päivittävät sovelluksen tilan näiden toimintojen perusteella.

Fluxin yksisuuntainen tietovirta auttaa välttämään monimutkaisia riippuvuuksia ja kilpailevia tilapäivityksiä, mikä tekee sovelluksen käyttäytymisestä ennustettavampaa ja helpottaa virheiden jäljittämistä ja korjaamista.



Kuva 6. Flux-arkkitehtuuri.

3.4 Redux

Redux (21) on avoimen lähdekoodin arkkitehtuurimalli, joka kehitettiin laajentamaan Flux-arkkitehtuuria ja tarjoamaan tehokkaampi tapa hallita JavaScript-sovellusten tilaa. Reduxia käytetään erityisesti React-sovelluksissa, mutta se on riippumaton käyttöliittymäkirjastoista ja soveltuu monenlaisiin JavaScript-pohjaisiin ympäristöihin. Reduxin keskeinen piirre on, että se ylläpitää sovelluksen tilaa yhtenäisessä globaalissa säilössä, jossa tila voidaan ennustettavasti muokata puhtaiden funktioiden avulla.

Redux-arkkitehtuurissa (Kuva 7) on kolme pääkomponenttia ja yksi mahdollinen lisäkomponentti:

Säiliö
Tämä komponentti on keskeinen säiliö, jossa sovelluksen globaali tila sijaitsee.

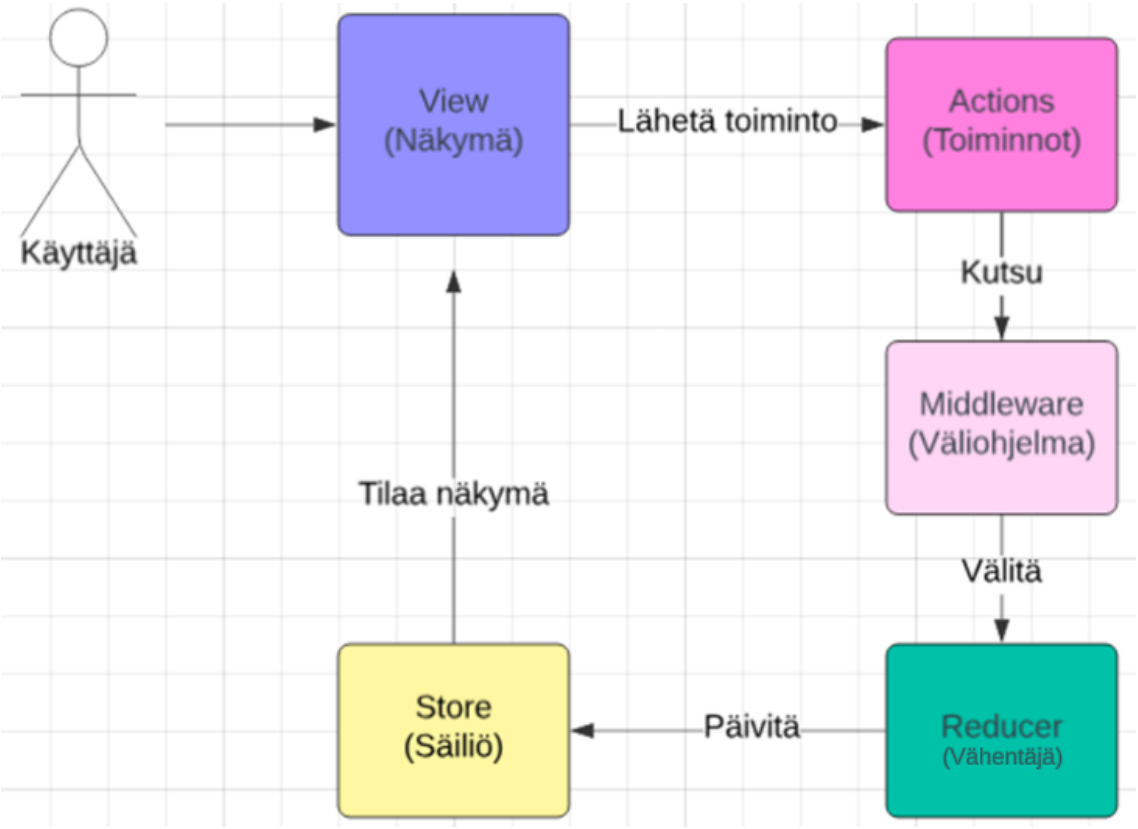
Se on ainoa paikka, jossa sovelluksen tila tallennetaan, mikä tekee tilan hallinnasta keskittynyttä ja ennustettavaa. Säiliöt vastaanottavat toimintoja ja delegoivat nämä vähentäjäfunktioille, jotka päivittävät tilan ja palauttavat uuden tilan.

Toiminnot
Nämä ovat JavaScript-objekteja, jotka kuvaavat sovelluksessa tapahtuvia muutoksia tai käyttäjän toimintoja. Toiminnot sisältävät aina vähintään tyyppitiedon, joka määrittelee, millainen muutos on tapahtumassa. Toiminnot voivat sisältää myös lisätietoja, joita tarvitaan tilan päivittämiseen.

Vähentäjä
Vähentäjät ovat puhtaita funktioita, jotka määrittelevät, miten sovelluksen tila muuttuu toimintojen perusteella. Vähentäjä-funktio ottaa vastaan nykyisen tilan ja toiminnon, ja palauttaa uuden tilan. Vähentäjät eivät koskaan muuta nykyistä tilaa suoraan, vaan luovat aina uuden tilan version, mikä tekee sovelluksen tilanhallinnasta ennustettavaa ja mahdollistaa ajassa taaksepäin siirtymisen.

Väliohjelma (lisäkomponentti)
Väliohjelmat tarjoaa tavan laajentaa Reduxin toimintaa ennen kuin toiminnot saavuttavat vähentäjäfunktiot. Väliohjelmat voivat keskeyttää, muokata tai suorittaa sivuvaikutuksia toimintojen perusteella. Yksi yleinen käyttötapa on asynkronisten toimintojen, kuten API-kutsujen, käsittely ennen tilan päivitystä.

Reduxin arkkitehtuuri perustuu yksisuuntaiseen tietovirtaan ja tilan keskitettyyn hallintaan, mikä tekee sovelluksista ennustettavia ja helpommin testattavia. Reduxin toimintamalli helpottaa myös tilanhallinnan skaalautumista, erityisesti suurissa ja monimutkaisissa sovelluksissa.



Kuva 7. Redux-arkkitehtuuri.

3.5 Arkkitehtuurien vertailu

Virhe. Viitteen lähdettä ei löytynyt. vertailee viittä keskeistä sovellusarkkitehtuuria, jotka ovat MVC, MVVM, MVP, Flux ja Redux. Taulukko tarjoaa kattavan näkymän kunkin arkkitehtuurin pääkomponenteista, tietovirrasta, käyttöliittymän hallinnasta, liiketoimintalogiikasta, tilanhallinnasta, vuorovaikutuksesta, testattavuudesta, skaalautuvuudesta ja asynkronisuuden käsittelystä.

Taulukko 1. Arkkitehtuurien vertailut.

Ominaisuus	**MVC**	**MVVM**	**MVP**	**Flux**	**Redux**

Komponentit	Malli, Näkymä, Käsittelijä	Malli, Näkymä, Näkymämalli	Malli, Näkymä, Esittäjä	Näkymä, Säiliö, Lähettäjä, Toiminto	Säiliö, Toiminto, Vähentäjä, Väliohjelma
Tietovirta	Kaksisuuntainen	Yksi-suuntainen	Kaksi-suuntainen	Yksi-suuntainen	Yksisuuntainen
Käyttöliittymä (UI)	Näkymä päivittyy suoraan käsittelijän kautta	Näkymämalli huolehtii tiedon sitomisesta	Näkymä päivittyy esittäjäkomponentin kautta	Näkymä kuuntelee säiliötä	Näkymä kuuntelee säiliötä
Liiketoiminta-logiikka	Käsittelijä hallitsee sovelluksen logiikkaa	Näkymämalli hallitsee liiketoimintalogiikkaa	Esittäjä hallitsee esitys-logiikkaa	Säiliöt hallitsevat tilaa ja logiikkaa	Vähentäjät hallitsevat tilaa ja logiikkaa
Tilanhallinta	Malli säilyttää tilan	Malli säilyttää tilan	Malli säilyttää tilan	Säiliö säilyttävät tilan	Säiliö säilyttää globaalia tilaa
Vuorovaikutus	Käsittelijä päivittää näkymän ja mallin	Näkymämalli päivittää näkymän ja mallin	Esittäjä päivittää näkymän ja mallin	Lähettäjä koordinoi toimintoja säiliöihin	Väliohjelma käsittelee toimintoja ja vähentäjää
Testattavuus	Keskikokoinen, voi olla haastavaa	Hyvä, koska näkymämalli on eristetty	Hyvä, koska esittäjä on eristetty	Hyvä, koska tilanmuutokset ovat ennustettavia	Erinomainen, tilanmuutokset ovat ennustettavia
Skaalautuvuus	Hyvä, mutta voi tulla mo-	Hyvä, erityisesti suu-	Hyvä, erityisesti sovelluksissa,	Hyvä, yksisuuntainen tietovirta	Erinomainen, yksisuuntainen

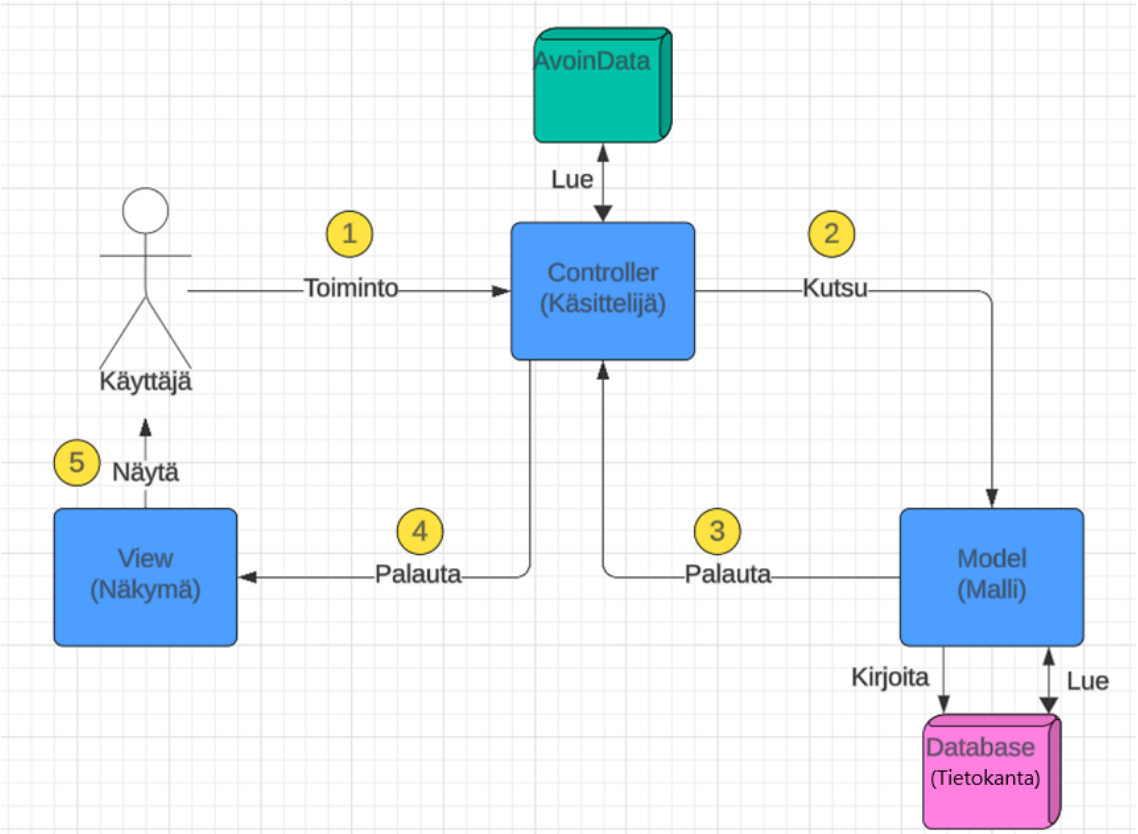
	nimut-kaiseksi suurem-missa so-velluksissa	rissa sovel-luksissa	joissa on monimutkai-nen UI	auttaa hal-linnassa	tietovirta ja keskitetty tila
Asynkroni-suus	Asynkroni-suus voi-daan toteut-taa käsittelijässä	Asynkroni-suus voi-daan toteut-taa näkymä-mallissa	Asynkroni-suus voi-daan toteut-taa esittä-jässä	Lähettäjä ja säiliöt voivat käsitellä a-synkronisia tapahtumia	Väliohjelma käsittelee asynkroni-sia tapahtu-mia
Hyvät puo-let	- Selkeä ra-kenne, jossa roolit ovat hyvin määriteltyjä. - Yleisesti ymmärretty ja käytetty. - Hyvä pie-nissä ja keskikokoi-sissa sovel-luksissa.	- Selkeä erottelu UI ja liiketoi-mintalogii-kan välillä. - Hyvä data-sidonta mahdollis-taa tehok-kaan päivi-tyksen. - Hyvä suu-rissa sovel-luksissa.	- Hyvin sel-keä ra-kenne, jossa esitys-logiikka on eristetyssä esittäjässä. - Hyvä käyt-tää, kun UI on moni-mutkainen. - Hyvä tes-tattavuus.	-Yksi-suun-tainen tieto-virta tekee sovelluksen tilan ennus-tettavaksi. - Hyvin skaa-lautuva. - Selkeä erottelu tilan hallinnan ja näkymän välillä.	-Yksisuun-tainen tieto-virta ja kes-kitetty tila helpottavat tilan hallin-taa. - Erinomai-nen testat-tavuus. - Väliohjelma tar-joaa joustavan tavan käsitellä asynkroni-sia tapahtu-mia.
Huonot puo-let	- Kaksi-suuntainen tietovirta voi tehdä tilan-hallinnasta monimutkai-sempaa. - Käsittelijä voi helposti tulla liian monimut-kaiseksi	-Data si-donta voi olla vaikeaa hallita ja vianetsintä voi olla haastavaa. -Mahdolliset suoritus-ky-kyongelmat suuriin näkymiin.	-Kaksi-suuntainen tietovirta voi johtaa tilan-hallinnan monimutkai-suuteen. - Näkymä ja esittäjä voi-vat olla tiu-kasti sidot-tuja.	- Fluxin ra-kenne voi olla vaikea ymmärtää alkuun. - Lähettäjä voi tulla pul-lonkaulaksi suurissa so-velluksissa. - Koodin or-ganisointi	- Reduxin rakenne voi olla yli-suurta pie-nille sovel-luksille. - Väliohjelma voi li-sätä komp-leksisuutta. - Tarvitsee

	- Vaatii enemmän koodia suurissa sovelluksissa.	-Näkymämallin tilanhallinta voi olla monimutkaista.	- Vaatii lisätyötä testauksen ja ylläpidon kannalta.	voi olla haastavaa.	paljon boilerplate-koodia, mikä voi tehdä kehittämisestä hitaampaa.

3.6 Valittu arkkitehtuuri

Sovelluksen arkkitehtuuriksi valittiin MVC-suunnittelumalli, joka jakaa sovelluksen kolmeen pääkomponenttiin: malliin, näkymään ja käsittelijään. Tämä arkkitehtuurimalli tekee sovelluksesta helposti hallittavan ja ylläpidettävän. MVC-malli mahdollistaa myös komponenttien uudelleenkäytön ja edistää modulaarista lähestymistapaa ohjelmistokehityksessä.

Valitsimme MVC-mallin erityisesti siksi, että se tarjoaa selkeän ja jäsennellyn rakenteen, joka sopii erinomaisesti keskikokoisiin sovelluksiin. MVC-mallin avulla sovelluksen koodi pysyy helposti ymmärrettävänä, mikä parantaa ylläpidettävyyttä ja skaalautuvuutta. Lisäksi mallin selkeä komponenttien erottelu mahdollistaa tehokkaan yhteistyön tiimin jäsenten välillä, sillä kukin komponentti voidaan kehittää ja testata erikseen ilman, että se vaikuttaa muihin osiin. Tämä tekee MVC-mallista tasapainoisen ja käytännöllisen ratkaisun valittuun sovelluskehitykseen. Kuva 8 esittää suunnitellun MVC mallin ylätason arkkitehtuurin, jota käytämme viitteenä sovellusta kehittäessä.



Kuva 8. Korjausvelka sovelluksen ylätason arkkitehtuuri.

Seuraava käyttäjän ja komponenttien vuorovaikutus varmistaa, että jokainen komponentti vastaa tietystä sovelluksen toiminnallisuuden osa-alueesta, mikä johtaa helpommin ylläpidettävään ja skaalautuvampaan arkkitehtuuriin:

- Käyttäjä on vuorovaikutuksessa näkymän kanssa esimerkiksi napsauttamalla painiketta tai kirjoittamalla tekstiä lomakkeeseen. Näkymä vastaanottaa tämän syötteen ja välittää sen käsittelijälle, joka prosessoi sen.
- Käsittelijä tarkistaa syötteen ja validoi sen, esimerkiksi varmistaen, ettei siinä ole virheitä. Jos syötteessä ilmenee ongelmia, käsittelijä palauttaa näkymälle virheilmoituksen. Jos syöte on hyväksytty, käsittelijä päivittää mallikomponenttia käyttäjän toiminnan tai sovelluksen logiikan mukaisesti.
- Malli puolestaan käsittelee pyynnön ja tekee tarvittavat muutokset sovellukseen, kuten tietojen hakemisen, tallentamisen tai poistamisen. Kun muutokset on tehty, malli ilmoittaa niistä käsittelijälle, joka välittää tiedon

takaisin näkymälle. Lopuksi näkymä päivittää käyttöliittymän heijastamaan tehtyjä muutoksia, jolloin käyttäjä näkee ne sovelluksessa.

3.7 Valittu teknologia

Sovelluksen suunnitteluvaiheessa halusimme asettaa itsellemme oppimistavoitteita kehitykseen ja päätimme tehdä osan sovelluksesta eri teknologialla mihin olemme aikaisemmissa projekteissa tottunut.

Emme halunneet keskittyä liikaa projektissa sivuston visuaaliseen kehittämiseen sovelluksen prototyypin kehitysvaiheessa, joten valitsimme React Nativen (22) käyttöliittymäkehykseksi sovellukseen. Olimme jo ennalta kerennyt perehtymään sovelluksien kehittämiseen sillä, joten pystyimme hyödyntämään jo olemassa olevaa osaamista ja työskentelemään tehokkaasti alusta alkaen.

React Native on myös hyvä valinta sen monipuolisuuden ja laajan komponenttien jälleen käytettävyyden vuoksi, mikä nopeuttaa kehitysprosessia ja mahdollistaa selkeän koodirakenteen. Se on myös erittäin skaalautuva, mikä tarkoittaa, että voimme helposti laajentaa sovellusta tulevaisuudessa.

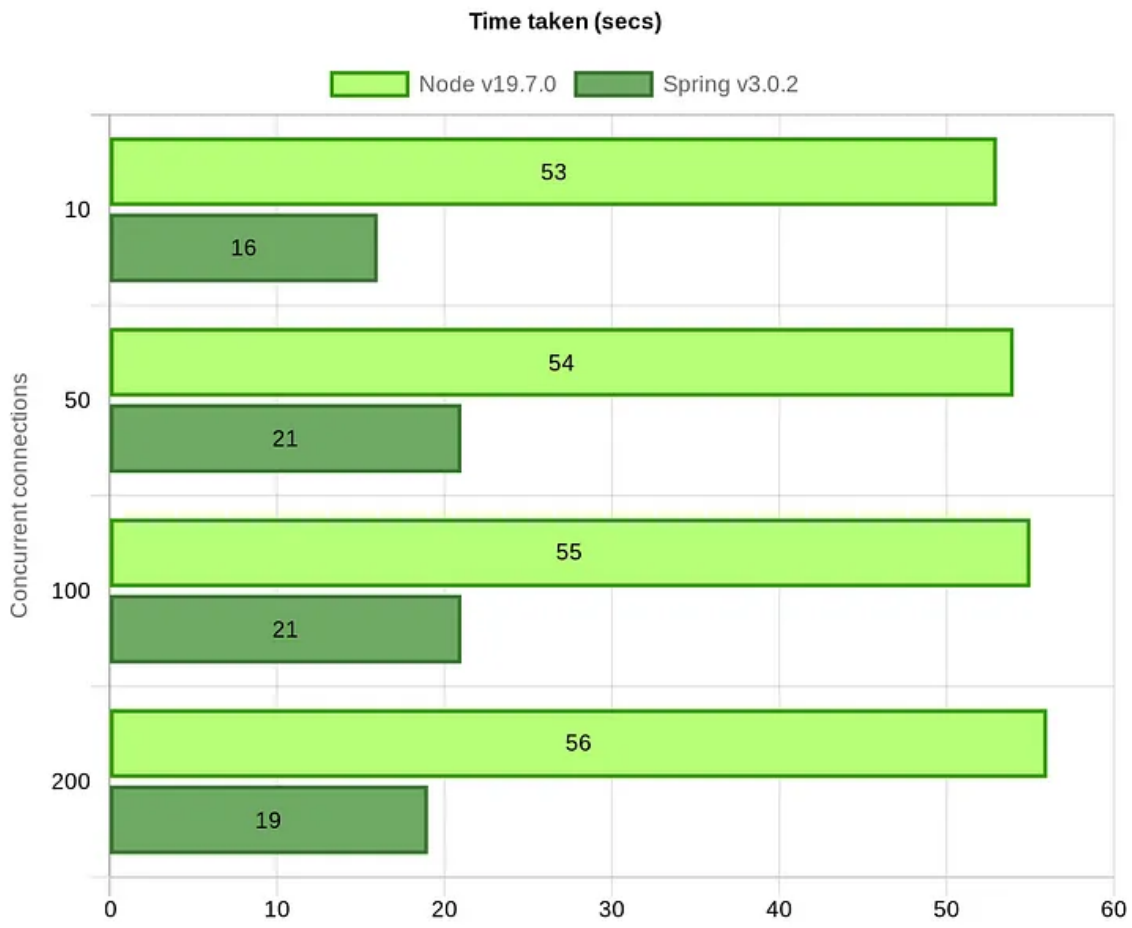
Node.js:ään laajennettu palvelinkehys Express.js (23) oli meille kaikista tutuin palvelinpuolen kehys, ja olimme käyttäneet sitä usein Full Stack -projekteissa. Halusimme kokeilla tästä eri vaihtoehtoa, mutta silti tutulla kielellä, joten Java oli meille erinomainen vaihtoehto. Vaikka Java ohjelmointikielenä oli meille tuttu sovelluskehityksessä ja tietokantayhteyksien luonnissa, valitsimme taustapalvelun kehykseksi Java Spring Bootin, jota kumpikaan meistä ei ollut aikaisemmin käyttänyt. Se on erittäin laajalti käytössä alalla ja on yksi suosituimmista palvelimiin käytetyistä taustapalvelukehyksistä. (24.)

Kun tutkimme Express.js:n sekä Spring Bootin erovaisuuksia, huomasimme suuren eroavaisuuden näiden vasteajoissa. Kuva 9 on tulos vasteajoille, kun samanaikaisia yhteyksiä on 10, 50, 100 ja 200. Testi oli toteutettu käyttäen yksinkertaista http get -kutsua, joka palauttaa käyttäjälle ”Hello World” -tekstin.

Nopeustestissä pyyntöjen määräksi asetettiin miljoona. Spring bootilla aikaa tähän meni noin 20 sekuntia jokaisella testikerralla, kun taas expressillä yli 50 sekuntia.

Testissä huomasimme suurenkin eron näiden kahden teknologian viiveistä. Vaikka testissä yksittäiselle yhteydelle toteutettiin miljoona kutsua, eikä tällainen määrä välttämättä olisi realistinen sovelluksessamme. Saimme hyvän käsityksen näiden kahden nopeuksista.

Halusimme varmistaa, että taustapalvelu kykenee suorittamaan useita laskutoimituksia samanaikaisesti ilman merkittäviä viiveitä käyttäjille. Tuloksista huomasimme, että Spring Boot hyödyntää enemmän prosessointitehoa yhteyksien käsittelyssä kuin Express.js (25). Tämän perusteella päättelimme, että Spring Boot suoriutuisi laskennasta paremmin.



Kuva 9. Vertailu Node.js:n kehyksen (express.js) ja Spring bootin kutsun vasteajalla (25).

Tietokantaratkaisuksi valitsimme MongoDB:n sen joustavuuden ja suorituskyvyn vuoksi, jotka ovat erityisen tärkeitä dynaamisten ja suurten tietomäärien hallinnassa. MongoDB dokumenttipohjainen rakenne sopii erinomaisesti sovelluksemme tarpeisiin, ja sen skaalautuvuus varmistaa, että tietokantamme pystyy käsittelemään kasvavia datamääriä vaivattomasti.

4 Projektin toteutus

Seuraavissa kappaleissa käsittelemme vaiheittain, kuinka toteutetaan Spring Boot -pohjainen taustapalvelu ja miten siihen liitetään tietokanta. Lisäksi tarkastelemme taustapalvelun arkkitehtuuria. Selvitämme myös, miten voimme käyttä-

jän antamien tietojen perusteella määrittää korjausvelan, miten käyttäjän autentikointi on toteutettu ja millaisia käyttöliittymän näkymiä sovellus tarjoaa. Lopuksi kuvaamme, kuinka sovellus saatiin julkaistua Azure-pilvipalveluun.

4.1 Taustapalvelu

Taustapalvelun ohjelmointikieleksi valitsimme Javan sen tuttuuden takia, ja käytämme Spring Boot kehystä taustapalvelimen rakentamiseksi. Spring Boot vähentää tehtävää työtä huomattavasti sen tarjoamien valmiiden toimintojen ja automatisaatioiden avulla. Koska Spring Boot sisältää autoconfiguration-ominaisuuden, monet perinteisesti käsin tehtävät asetukset hoidetaan automaattisesti (26). Automaatio on myös helppo ohittaa silloin, kun on tarve määrittää jotain itse.

Spring Boot tarjoaa myös laajan valikoiman valmiiksi integroitavia kirjastoja ja riippuvuuksia, kuten tietokantayhteydet, REST-palvelut ja turvallisuusratkaisut. Näiden avulla voimme ottaa helposti käyttöön erilaisia toiminnallisuuksia vain lisäämällä tarvittavan riippuvuuden projektin pom.xml -määrittelytiedostoon.

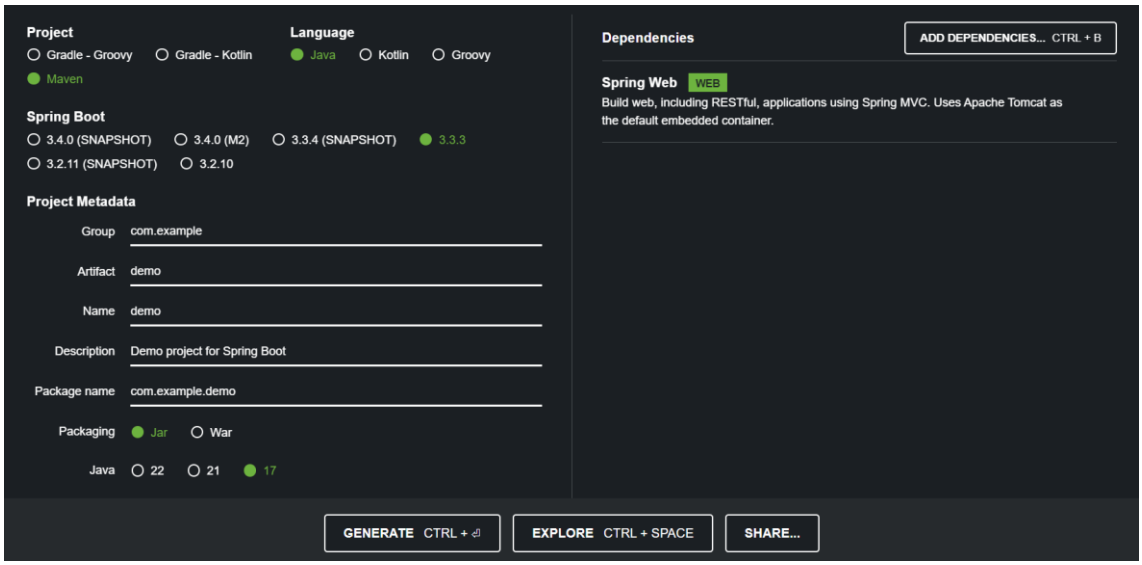
Meillä ei ollut aiempaa kokemusta tämän kehyksen käytöstä, mutta koska sitä on kehitetty yli kahden vuosikymmenen ajan, käyttöohjeet ovat erittäin selkeät ja verkosta löytyy runsaasti ohjeita ja parhaita käytäntöjä. Valintaan vaikutti myös huomattava määrä työpaikkoja, joissa pyydetään tämän kehyksen osaamista.

4.1.1 Käyttöönotto

Otimme Spring Boot -palvelun käyttöön hyödyntämällä sen omaa integroitua generaattoria (27), joka tarjoaa helpon ja käyttäjäystävällisen tavan määrittää projektin asetukset. Kuva 10 näyttää generaattorin, jolla voi vaivattomasti valita projektin kielen, koontityökalun (esimerkiksi Maven tai Gradle), ja lisätä tarvittavat Spring-kirjastot projektiin yhdellä napsautuksella.

Jotta palvelin voisi keskustella käyttöliittymän kanssa, tarvitaan erityinen kirjasto nimeltään Spring Web. Tämä kirjasto mahdollistaa HTTP-pyyntöjen käsittelyn ja

reitityksen REST API:n kautta. Spring Web tarjoaa työkalut sekä GET- että POST-pyyntöjen käsittelyyn, lomakedatan validointiin sekä JSON- tai XML-muodossa palautettaviin vastauksiin.



Kuva 10. Spring Boot -generaattori.

Kun kaikki tarvittavat toimenpiteet on suoritettu, generoidaan sivustosta toimiva taustapalvelu-pohja. Projekti on suunniteltu plug-and-play-periaatteella, joten palvelu voidaan käynnistää välittömästi painamalla "Run"-painiketta (Koodiesimerkki 1).

```
@SpringBootApplication
public class Korjausvelka {
    public static void main(String[] args) {
        SpringApplication.run(Korjausvelka.class, args);
    }
}
```

Koodiesimerkki 1. Palvelun käynnistyskoodi.

Seuraavaksi luomme käsittelijäluokan, joka vastaa pyyntöjen käsittelystä. Jotta pyyntöjä voidaan käsitellä oikein, määrittelemme uuden luokan HelloWorld, joka vastaa halutun pyynnön käsittelystä.

Spring Bootissa käsittelijät luodaan lisäämällä luokkaan “@RestController” annotaatio, joka tekee luokasta http-pyyntöjä vastaanottavan ja niihin vastaavan rajapinnan. Rajapinnan osoitetta voidaan hallita “@RequestMapping(“/**osoite**”) annotaatiolla, jolla saadaan kaikki saman aliosoitteen alle.

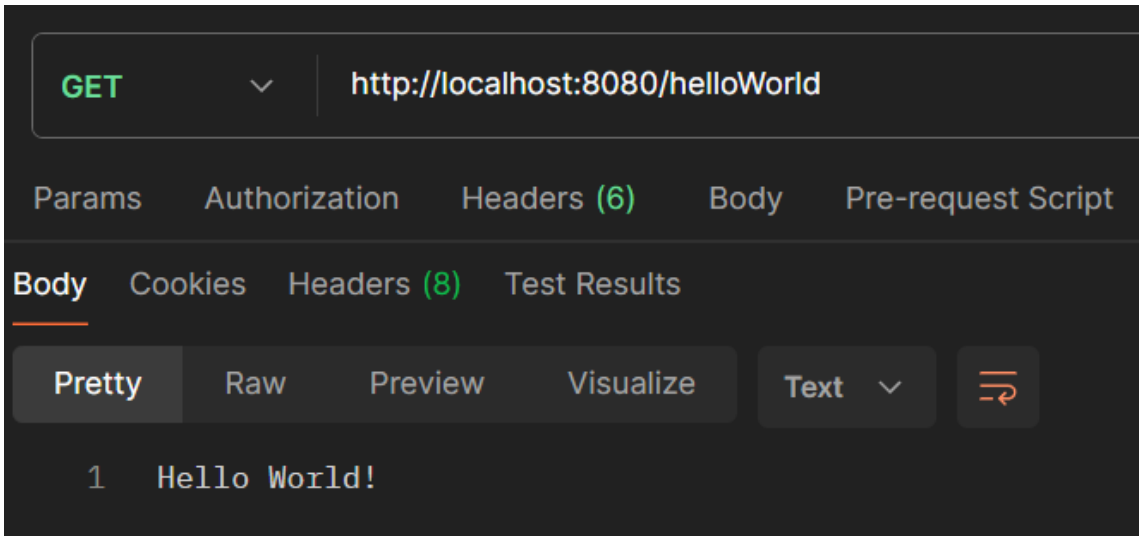
Tämän jälkeen määrittelemme myös, onko kyseessä Post-, Get- vai Delete-operaatio. Esimerkissämme haluamme palauttaa käyttäjälle merkkijonon “Hello World”, joten annamme metodille @GetMapping-annotaation, joka osoittaa, että kyseessä on Get-operaatio. Samalla periaatteella toimivat myös Post- ja Delete-operaatiot. Annotaation jälkeen määritämme polun, joka kertoo, mihin osoitteeseen pyyntö tulee lähettää, jotta palautamme käyttäjälle vastauksen. Seuraavassa tapauksessa (Koodiesimerkki 2) pyyntö osoitteeseen /helloWorld palauttaa merkkijonon ”Hello World!”.

```
@RestController
public class HelloWorld {

    @GetMapping("/helloWorld")
    public String helloWorld() {
        return "Hello World!";
    }
}
```

Koodiesimerkki 2. Esimerkki controller-luokasta.

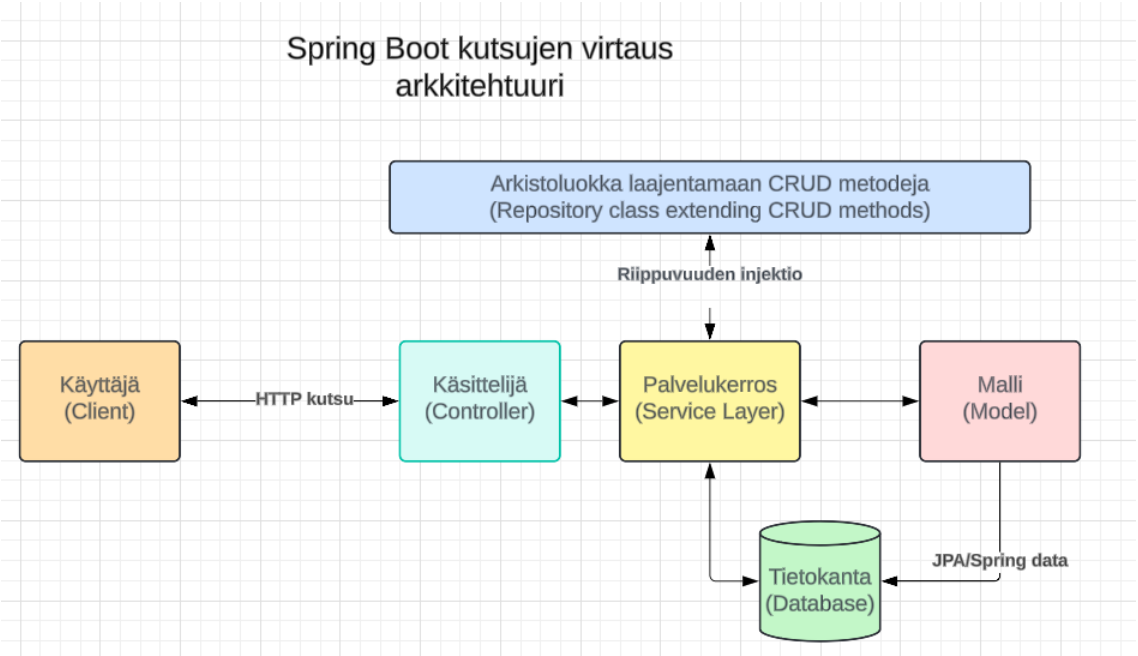
Nyt voimme lähettää Postman-sovelluksella http-pyynnön taustapalvelun osoitteeseen ja vastaanottaa palautuksena merkkijonon. Kuva 11 on esimerkki Postman-pyynnöstä ja sen palautuksesta.



Kuva 11. Postman -esimerkki taustapalvelun kutsumiseen.

4.1.2 Arkkitehtuuri

Kuva 12 esittele suunnitellun taustapalvelun arkkitehtuurin, joka perustuu MVC-malliin. Arkkitehtuuri on Spring Boot -sovellusten keskeinen perusta, joka mahdollistaa selkeän ja loogisen sovellusrakenteen (28).



Kuva 12. Spring Boot -arkkitehtuuri.

Java Spring Boot -arkkitehtuurissa sovellus jaetaan neljään osaan: käsittelijään, palvelukerrokseen, malliin ja arkistoluokkaan.

Käsittelijäluokat vastaavat käyttäjän syötteiden käsittelystä ja validoinnista, esimerkiksi kirjautumispyynnössä käyttäjän tunnuksen ja salasanan tarkistamisesta. Lisäksi ne välittävät kutsun mukana tulleet parametrit palveluluokille jatkokäsittelyä varten. Tämä kerros toimii ikään kuin välittäjänä käyttöliittymän ja sovelluslogiikan välillä, ja sen tarkoitus on varmistaa, että käyttäjän toiminta käsitellään oikeaoppisesti ennen tietojen siirtoa eteenpäin.

Palveluluokka vastaa liiketoimintalogiikan toteuttamisesta. Sen tehtävänä on esimerkiksi tietokantakutsujen suorittaminen sekä tietokannasta palautettujen tietojen validointi ja muokkaaminen ennen niiden palauttamista käsittelijä luokalle. Palvelukerros on keskeinen osa sovelluksen liiketoimintalogiikkaa, ja se pitää huolen, että kaikki sovelluksen toiminnot suoritetaan oikein.

Malli luokka edustaa yksittäistä oliota, joka sisältää olion tiedot ja siihen liittyvän toiminnallisuuden. Malliluokat ovat keskeinen osa olio-ohjelmointia, jossa sovelluksen eri osat ovat kapseloituja olioihin. Olio-ohjelmoinnin periaatteet, kuten kapselointi ja perintä tekevät sovelluksen rakenteesta selkeän ja lisäävät uudelleenkäytettävyyttä.

Arkisto-rajapinta sisältää tietokannan CRUD-metodit. Rajapinta toimii yhteyspisteenä sovelluksen ja tietokannan välillä, mikä mahdollistaa tehokkaan ja turvallisen tavan käsitellä tietoja. Rajapintaan peritään halutun tietokantarepositorion rajapinta, esimerkiksi meidän tapauksessamme MongoRepository-rajapinta, joita arkisto luokka käyttää tietokantaoperaatioiden suorittamiseen ilman SQL-kyselyiden kirjoittamista.

MongoRepository-rajapinnalle annetaan kaksi parametriä: ensinnäkin olio, jota kyseinen rajapinta käyttää, ja toiseksi tietokannassa olevan tunnisteen muoto. Nämä parametrit pitävät huolen, että vain kyseisen olion instanssit voivat kulkea rajapinnan läpi, mikä estää duplikaattien syntymisen tietokantaan.

Arkisto luokan tehtävä on vastata tietokannan skeeman luomisesta, mikä tarkoittaa, että se määrittelee, miten tiedot tallennetaan ja järjestetään tietokannassa. MongoDB kanssa työskennellessä tämä prosessi on erityisen sujuva, sillä MongoDB muuntaa olion automaattisesti omaan binääriseen muotoonsa (BSON) tallennusta varten.

Koodiesimerkki 3 esittelee, kuinka UserRepository määritellään. Tämä rajapinta laajentaa MongoRepository-rajapintaa, joka perii valmiit CRUD-toiminnot.

```
@Repository

public interface UserRepository extends MongoRepository<User, ObjectId>
```

Koodiesimerkki 3. UserRepository niminen tietokantarepositorio-rajapinta.

4.1.3 Korjausvelan laskeminen

Korjausvelan laskennassa käytämme käyttäjän syötteitä, jotka tulevat sovelluksesta pakattuna yhteen REST post -pyyntöön. Se sisältää kaiken tiedon, jota tarvitsemme talon remonttien aikajanan luomiseen, ja se validoidaan sekä lähetettäessä että vastaanotettaessa sen varmistamiseksi, että se sisältää kaiken vaaditun. Käytämme samaa dataa myös annuiteettilainan laskemiseen samassa kutsussa.

Tallennamme aluksi käyttäjän syötteet, koska tarvitsemme sitä raportin luomiseen sekä näyttämiseen myöhemmässä vaiheessa. Jos raportin luominen epäonnistuu, niin voimme tallentaa lokiin merkinnän virheestä ja käydä tutkimassa miksi virhe on tapahtunut. Tallentamisen jälkeen aloitamme laskemisen korjausvelasta, johon on luotu oma metodi, jota post-pyyntö kutsuu. Kun korjausvelka on laskettu, niin teemme saman annuiteettilainan laskentaa varten sen omalla metodillaan.

Koodiesimerkki 4 havainnollistaa, miten käyttäjän autentikointi varmistetaan ennen kuin hän voi lähettää pyyntöä korjausvelan laskemiseksi. Autentikoinnin avulla varmistetaan myös, että oikea raportti toimitetaan oikealle käyttäjälle. Jos käyttäjää ei löydy tietokannasta, raporttia ei luoda.

```
@PostMapping("/calculateDebt")

public ResponseEntity<String> calculateDebt(@RequestHeader("Authorization") String token,

        @RequestBody CalculationRequest calculateDebtRequest) {

    User currentUser = (User) SecurityContextHolder.getContext().getAuthentication().getPrincipal();

    if (currentUser == null) {

        return ResponseEntity.badRequest().body("User not found");

    }
```

Koodiesimerkki 4. Post-pyyntö korjausvelan laskuun ja käyttäjän autentikaation tarkastus.

Kun laskennat on tehty virheettä, tallennamme lopuksi korjausvelan sekä annuiteettilainan laskelmat tietokantaan. Näiden tallentaminen vasta lopussa on tärkeää, koska näin virhetilanteessa ei tarvitse etsiä tehtyjä tallennuksia tietokannasta poistoa varten. Emme myöskään halua käyttäjän näkevän viallisia raportteja listassa. Palautamme raportin id:n sivustolle, joka kertoo lomakesivulle mihin raporttiin sen pitää navigoida, jotta voimme näyttää käyttäjälle luodun raportin. Raportti on myös tämän jälkeen löydettävissä käyttäjän omalta raporttilistasivulta.

4.1.4 MongoDB

Jotta voimme muodostaa yhteyden tietokantaan, tarvitsemme muutamia lisäkirjastoja, jotka mahdollistavat tietokantayhteyden toimimisen. Koodiesimerkki 5 esittelee tietokantayhteyden luomiseen tarvittavat maven-riippuvuudet.

```
<dependency>
        <groupId>org.mongodb</groupId>
        <artifactId>mongodb-driver-sync</artifactId>
        <version>5.1.2</version>
</dependency>
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-mongodb</artifactId>
```

```
</dependency>
```

Koodiesimerkki 5. Maven riippuvuudet tietokantayhteyden luomiseen.

Kun kirjastot on määritelty, luomme luokan, joka vastaa tietokantayhteyden muodostamisesta. Koodiesimerkki 6 luo yhteyden tietokantaan, ja onnistuneen yhteyden saataessa konsoliin tulostuu viesti "You successfully connected to MongoDB".

```
@Configuration
@EnableMongoAuditing
public class MongoConfig {
    @Value("${spring.data.mongodb.uri}")
    private String connectionString;
    @Bean
    public MongoClient mongoClient() {
        ServerApi serverApi = ServerApi.builder()
                .version(ServerApiVersion.V1)
                .build();
        MongoClientSettings settings = MongoClientSettings.builder()
                .applyConnectionString(new ConnectionString(connectionString))
                .applyToConnectionPoolSettings(builder -> builder.maxSize(50)
                        .minSize(10)
                        .maxConnectionIdleTime(30, TimeUnit.SECONDS)
                        .maxConnectionLifeTime(60, TimeUnit.SECONDS))
                .applyToSocketSettings(builder -> builder.connectTimeout(10, TimeUnit.SEC-
ONDS)
                        .readTimeout(10, TimeUnit.SECONDS))
                .applyToServerSettings(builder -> builder.heartbeatFrequency(10, TimeU-
nit.SECONDS)
                        .minHeartbeatFrequency(1, TimeUnit.SECONDS))
                .serverApi(serverApi)
                .build();
        // Create a new client and connect to the server
        MongoClient mongoClient = MongoClients.create(settings);
        try {
            // Send a ping to confirm a successful connection
            MongoDatabase database = mongoClient.getDatabase("Users");
```

```
            database.runCommand(new Document("ping", 1));
            System.out.println("Pinged your deployment. You successfully connected to Mon-
goDB!");
        } catch (MongoException e) {
            e.printStackTrace();
        }
        return mongoClient;
    }
}
```

Koodiesimerkki 6. Tietokantaan yhdistävä pala ohjelmaa.

Ennen palvelun käynnistämistä määritämme vielä application.properties-tiedostoon MongoDB yhteysparametrit (Koodiesimerkki 7). Tarvitsemme tietokannan osoitteen, tietokannan nimen sekä käyttäjätunnuksen ja salasanan. Näiden määritysten jälkeen palvelu voidaan käynnistää ja yhteys tietokantaan on valmis.

```
spring.data.mongodb.uri=mongodb+srv:url
spring.data.mongodb.database=tietokanta
spring.data.mongodb.username="käyttäjä"
spring.data.mongodb.password="salasana"
```

Koodiesimerkki 7. Spring data yhteysparametrien esimerkki.

4.1.5 Autentikaatio

Spring Boot tarjoaa tehokkaan ja selkeän tavan suojata sovelluksen eri polkuja. Jokainen taustapalveluun saapuva HTTP-pyyntö kulkee ensin sovelluksen turvallisuussuodattimen läpi. Tämä suodatin tarkistaa, millaisia käyttöoikeuksia käyttäjältä vaaditaan pyynnön kohteena olevan polun perusteella.

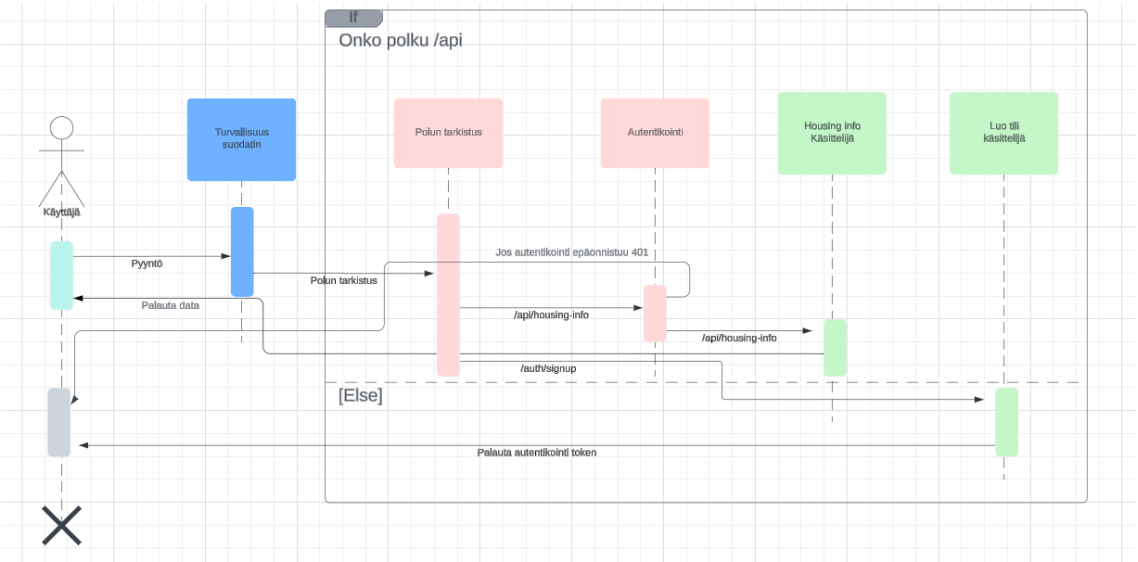
Mikäli käyttäjän pyyntö kohdistuu polkuun, joka alkaa "/auth", käyttäjältä ei vaadita minkäänlaista autentikointitunnusta eli tokenia. Näin ollen esimerkiksi kirjautumis- ja rekisteröitymisprosessit ovat vapaasti käytettävissä ilman ennakkotunnistautumista. Kaikki muut pyynnöt sen sijaan edellyttävät, että käyttäjällä on voimassa oleva autentikointitokeni. Tämä token tarkistetaan jokaisen pyynnön

yhteydessä ennen kuin pyyntö välitetään eteenpäin oikealle käsittelijälle. Kuva 13 esittää sekvenssikaaviolla turvallisuussuodattimen toimintaa tilanteessa, jossa käyttäjä on jo kirjautunut sisään, sekä tilanteessa, jossa uusi käyttäjä rekisteröityy.

Koodiesimerkki 8 kuvaa, kuinka tämä turvallisuussuodatin määritetään Spring Bootissa:

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http.csrf()
         .disable()
         .authorizeHttpRequests()
         .requestMatchers(HttpMethod.OPTIONS, "/**").permitAll()
         .requestMatchers("/auth/**")
         .permitAll()
         .anyRequest()
         .authenticated()
         .and()
         .sessionManagement()
         .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
         .and()
         .authenticationProvider(authenticationProvider)
         .addFilterBefore(jwtAuthenticationFilter, UsernamePasswordAuthenticationFil-
ter.class);
        return http.build();
    }
```

Koodiesimerkki 8. Turvallisuussuodattimen koodi.



Kuva 13. Sekvenssikaavio turvallisuussuodattimesta.

Suodattimen määrittelyn ansiosta Spring Boot käsittelee pyyntöjen reitityksen automaattisesti, joten kehittäjän ei tarvitse manuaalisesti määrittää jokaisen polun suojausta. Kun käyttäjän autentikaatio-token on tarkistettu ja pyyntö on todettu hyväksytyksi, se välitetään oikealle käsittelijäluokalle.

Koodiesimerkki 9 esittää miten käyttäjän pyyntö käsitellään, kun hän haluaa hakea kaikki omat raporttinsa:

```
@RequestMapping("/api/housing-info")

@RestController

public class HousingInfoController

    @GetMapping

    public List<HousingInfo> getAllHousingInfo() {

        return housingInfoService.getAllHousingInfo();

    }
```

Koodiesimerkki 9 Käsittelijäkerros

Esimerkissä (Koodiesimerkki 9) annotaatio, joka toimii syntaktisena metatietona, ohjaa automaattisesti kaikki "/api/housing-info"-polkuun kohdistuvat pyynnöt suoraan oikealle käsittelijäluokalle. Käsittelijä puolestaan ohjaa pyynnön edelleen Palvelukerrokselle (Koodiesimerkki 10).

```
public List<HousingInfo> getAllHousingInfo() {

    return housingInfoRepository.findAll();

}
```

Koodiesimerkki 10. Palvelukerros.

Palvelukerros lähettää pyynnön arkistoluokalle, joka on vastuussa tietokantakyselyjen suorittamisesta (Koodiesimerkki 11). Määrittämällä @Repository-annotaation ja laajentamalla MongoRepository-luokkaa, voimme helposti hyödyntää valmiita metodeja ilman, että meidän tarvitsee itse toteuttaa niitä. Tarvittaessa voimme kuitenkin myös kirjoittaa omia toteutuksiamme tietojen käsittelemiseksi.

```
@Repository

public interface HousingInfoRepository extends MongoRepository<HousingInfo, ObjectId>
```

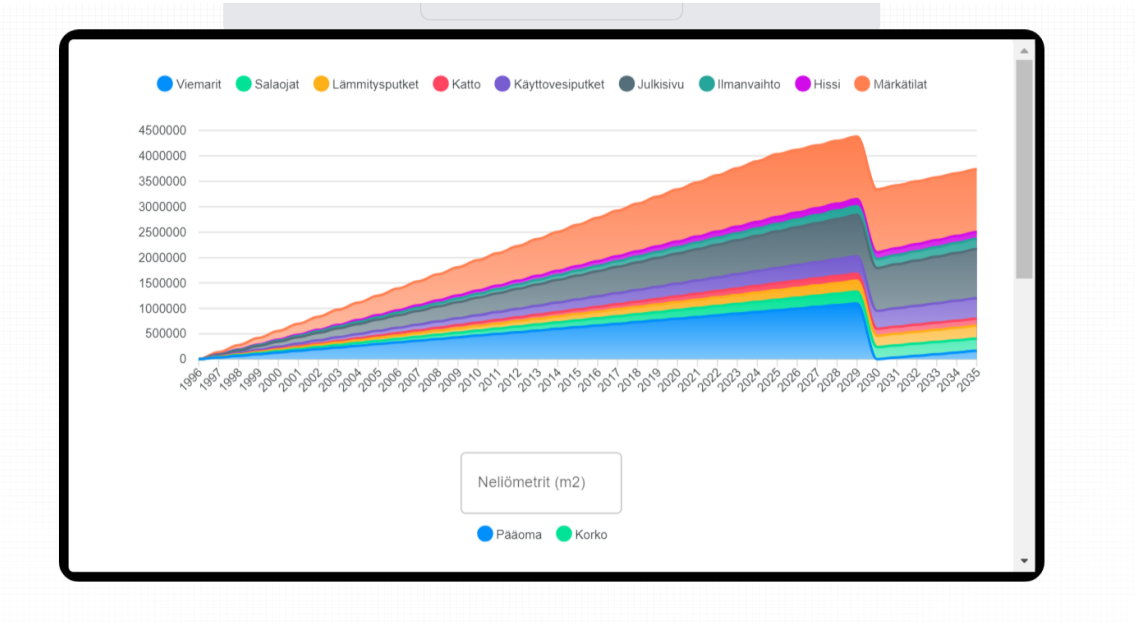
Koodiesimerkki 11. Arkistoluokkakerros.

4.2 Sovelluksen käyttöliittymä

Sivuston etusivusta halusimme tehdä mahdollisimman selkeän ja käyttäjäystävällisen. Tavoitteemme oli tarjota käyttäjälle heti ensisilmäyksellä selkeä kuva siitä, millainen raportti sovelluksesta tulee, ja samalla esitellä sovelluksen toimintaperiaatteet.

4.2.1 Sivuston etusivu

Etusivulle loimme interaktiivisen tietokoneen, jonka avulla käyttäjä voi tarkastella malliraporttia omatoimisesti (Kuva 14). Tämä malli antaa käyttäjälle mahdollisuuden tutustua raportin sisältöön ja toiminnallisuuksiin jo ennen varsinaisen raportin tekoa.



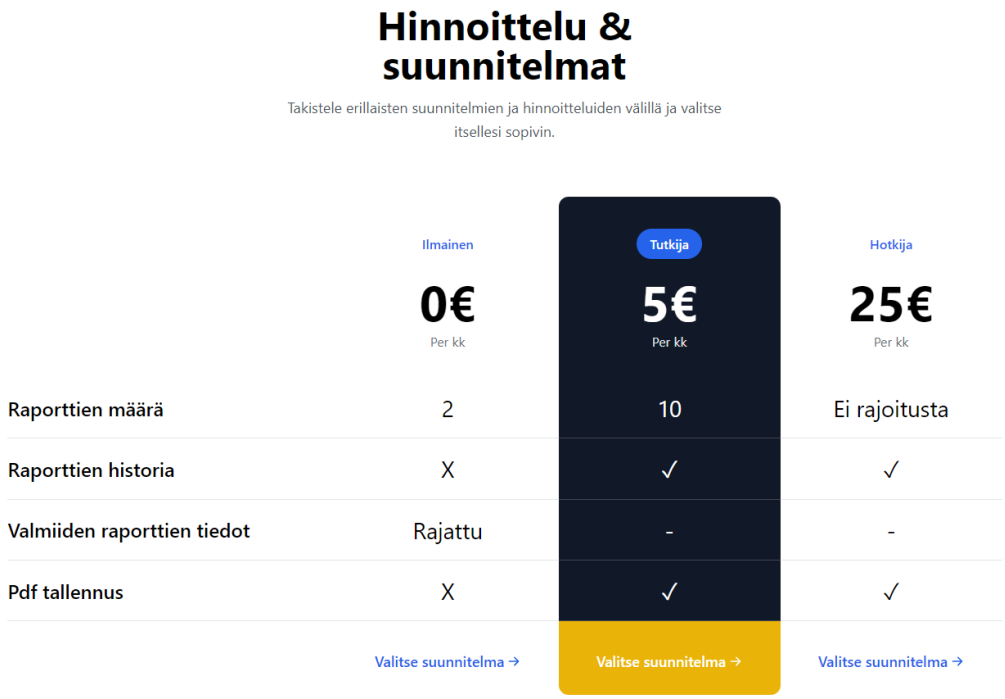
Kuva 14. Etusivun interaktiivinen malliraportti.

Malliraportti perustuu kuvitteelliseen, vuonna 1996 rakennettuun taloon. Käyttäjä voi interaktiivisesti tarkastella talon korjausvelkaa ja piilottaa tai näyttää eri korjauskohteita hiiren avulla. Kuvassa 14 näkyy vuoden 2030 kohdalla korjausvelan putoaminen, joka aiheutuu viemäriremontin ajankohdasta. Raportin alareunassa on lisäksi annuiteetti kaavio, johon käyttäjä voi syöttää leikkimielisesti huoneiston neliöpinta-alan. Kaavio näyttää, miten huoneiston omistajan omavastuu kasvaisi taloyhtiön remonttien yhteydessä.

4.2.2 Hinnoittelu ja erilaiset suunnitelmat

Halusimme tarjota käyttäjille mahdollisuuden valita kolmen erilaisen suunnitelman välillä. Sivuston tarkoituksena on selkeästi esittää käyttäjälle, millaisia vaihtoehtoja hänellä on, mitä ominaisuuksia kukin suunnitelma sisältää ja kuinka paljon kukin suunnitelma maksaa. Kuva 15 esittää sovelluksen tämänhetkisen

hinnoittelumallin. Suosituin suunnitelma on kehitetty saatujen palautteiden perusteella, joka perustuu lähipiiriltä ja ystäviltä saatuihin palautteisiin.



Kuva 15. Hinnoittelusivu.

4.2.3 Raportin luominen

Jotta voimme laskea korjausvelan, tarvitsemme lisätietoa taloyhtiöstä ja sen tilanteesta. Tällä hetkellä avoin data taloyhtiöistä on rajallista, joten keräämme tarvittavat tiedot käyttäjiltä lomakkeella. Korjausvelan laskeminen rajoittuu ainoastaan olemassa oleviin AsOy:hin. Hyödynnämme tässä prosessissa kaupparekisterin avointa dataa (29), johon haemme tietoja rajoitetusti hakuehtojen perusteella, joko yrityksen nimen tai y-tunnuksen avulla. Tämä menettely varmistaa, että käyttäjät syöttävät oikean AsOy:n tiedot, mutta ei takaa, että syötetyt tiedot ovat oikeellisia tai kuuluvat kyseiselle AsOy:lle.

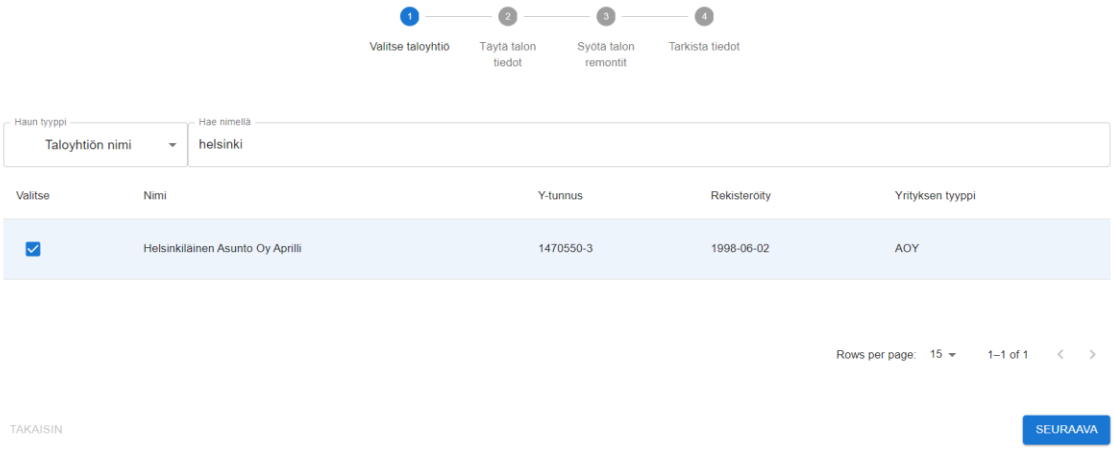
Haemme kaupparekisterin avoimen datan suoraan heidän tarjoamasta REST-API rajapinnasta. Pystymme rajoittamaan haun rajapinnasta vain rekisteröityihin

yrityksiin antamalla haussa parametreja. Yritysrekisteri päivittää palveluun tietoja kerran päivässä.

Kysymme käyttäjältä korjausvelan laskemiseen liittyvät tiedot vaiheittain, jotta käyttäjä voi keskittyä yhteen asiaan kerrallaan. Tämä lähestymistapa myös parantaa sivuston ulkoasua, koska näytöllä ei ole liikaa tietoa samanaikaisesti. Käytämme Material UI-kirjaston (30) React Stepper -komponenttia, joka tarjoaa helpon tavan luoda vaiheittaisia näkymiä. Stepper näyttää yläreunassa myös tämänhetkisen vaiheen, jonka avulla käyttäjä tietää kuinka pitkällä prosessissa ollaan.

Ensimmäisenä vaiheena on yrityshaku, jossa annamme kaupparekisterin API:n kautta joko nimellä tai y-tunnuksella haetun tuloksen AsOy hauista. Hakumäärät on rajoitettu pieneksi nopeuttaaksemme datan hakua API:sta. Y-tunnuksella haettaessa haku antaa vain yhden tuloksen. Tämä johtuu API:n rajoitteesta, jossa osittaishakua ei ole sallittua y-tunnuksella hakiessa.

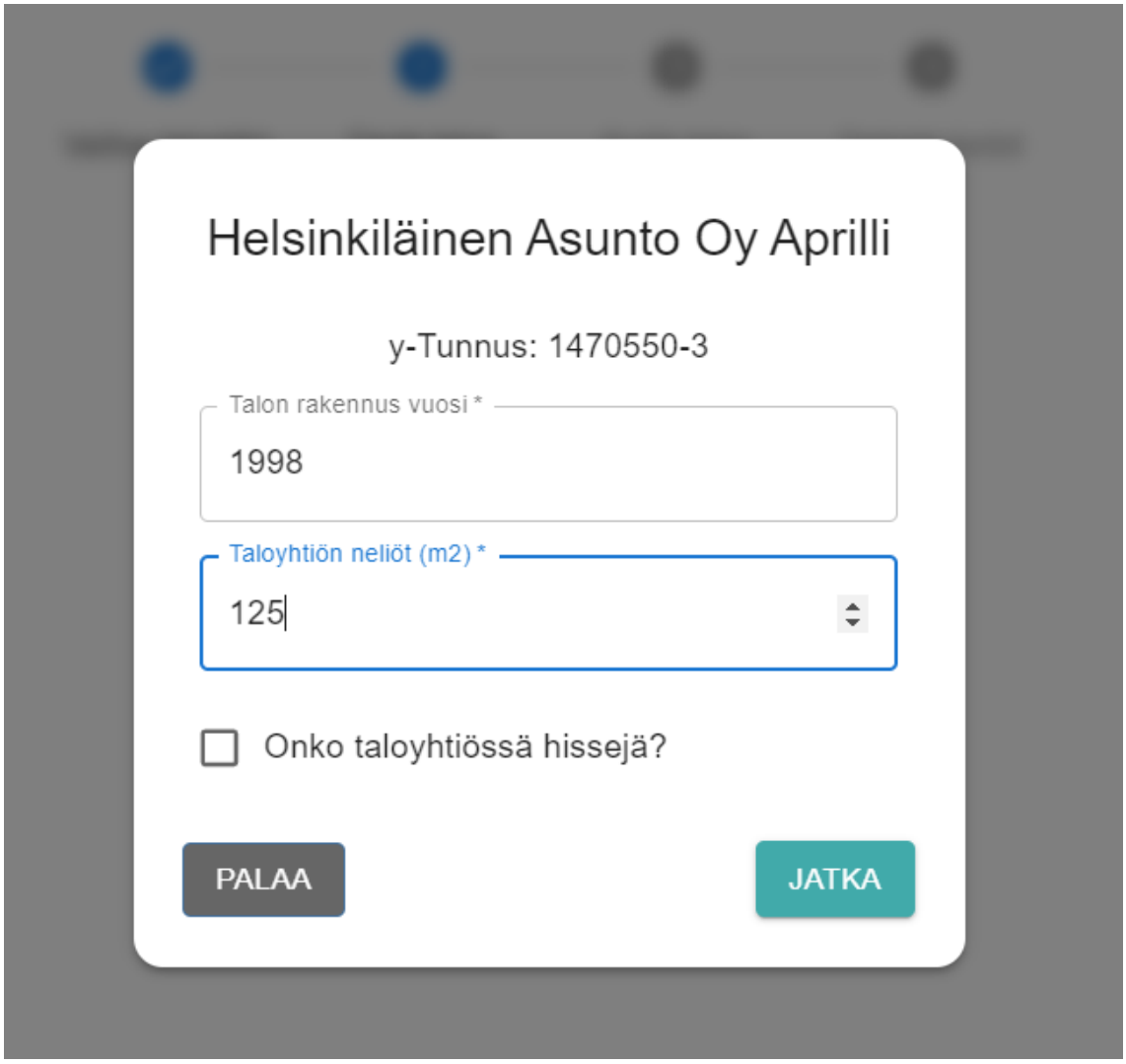
Onnistunut haku näytetään listana tuloksista, jossa näkyy yrityksen y-tunnus sekä milloin kohde on rekisteröity patentti- ja rekisterihallituksen rekisteriin (31). Käyttäjän tulee valita listasta taloyhtiö ennen kuin hän pääsee raportin teossa eteenpäin. Kun käyttäjä on valinnut taloyhtiön, ”seuraava” painike tulee aktiiviseksi. Kuvassa 16 näkyy ensimmäisen vaiheen näkymä, kun käyttäjä on hakenut ja valinnut haluamansa taloyhtiön.



Kuva 16. Korjausvelka sovelluksen raportin ensimmäinen vaihe.

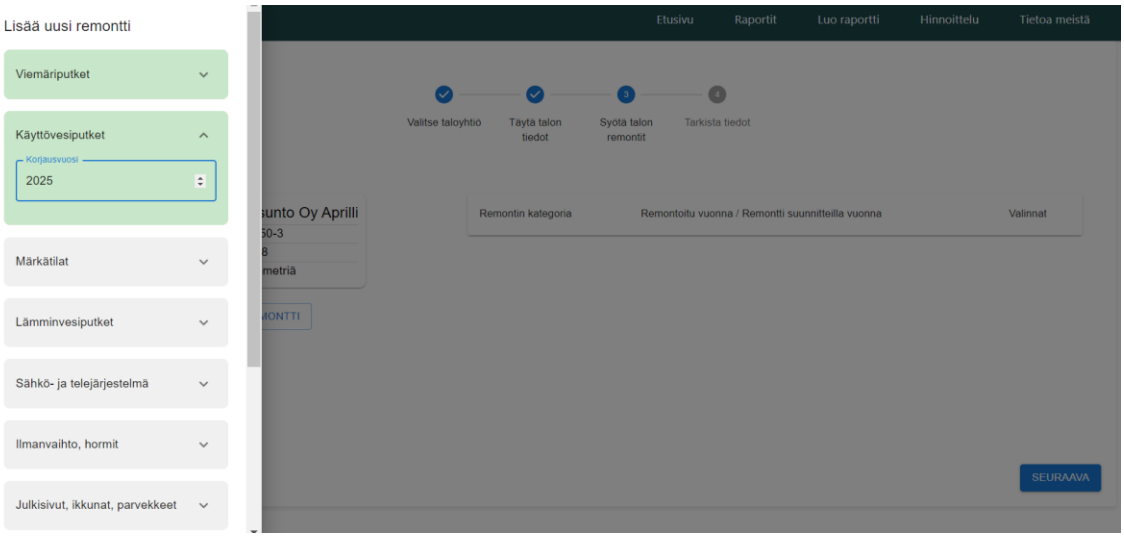
Toisessa vaiheessa keräämme taloyhtiöstä lisätietoja, kuten rakennusvuoden, pinta-alan ja hissin olemassaolon, jotta voimme arvioida korjausvelan tarkasti. Nämä tiedot ovat oleellisia, sillä ne vaikuttavat merkittävästi korjausvelan suuruuteen. Esimerkiksi vanhempien talojen korjaustarpeet ovat usein suurempia, ja hissin lisääminen jälkikäteen tuo suuria lisäkustannuksia.

Rakennusvuoden syötekenttä on rajoitettu realistiseen aikahaarukkaan, mikä vähentää virheellisten tietojen syöttämisen riskiä. Nämä tiedot ovat tärkeitä korjausvelan laskemisessa sekä kuntoarvion teossa. Kuvassa 17 on näkymä, jossa käyttäjä on syöttänyt tarpeelliset tiedot. Tietojen antamisen jälkeen käyttäjä pystyy jatkamaan eteenpäin.



Kuva 17. Toisen vaiheen näkymä.

Kolmannessa vaiheessa käyttäjä syöttää mitä remontteja taloyhtiössä on ollut ja mitä on suunnitteilla. Kuvassa 18 käyttäjä on syöttänyt remontin viemäriputkille sekä käyttövesiputkille, ja saa näkymään vihreän taustan kuvastamaan sitä, että kyseiseen kohteeseen on syötetty remontti.



Kuva 18. Kolmannen vaiheen remonttien syöttönäkymä.

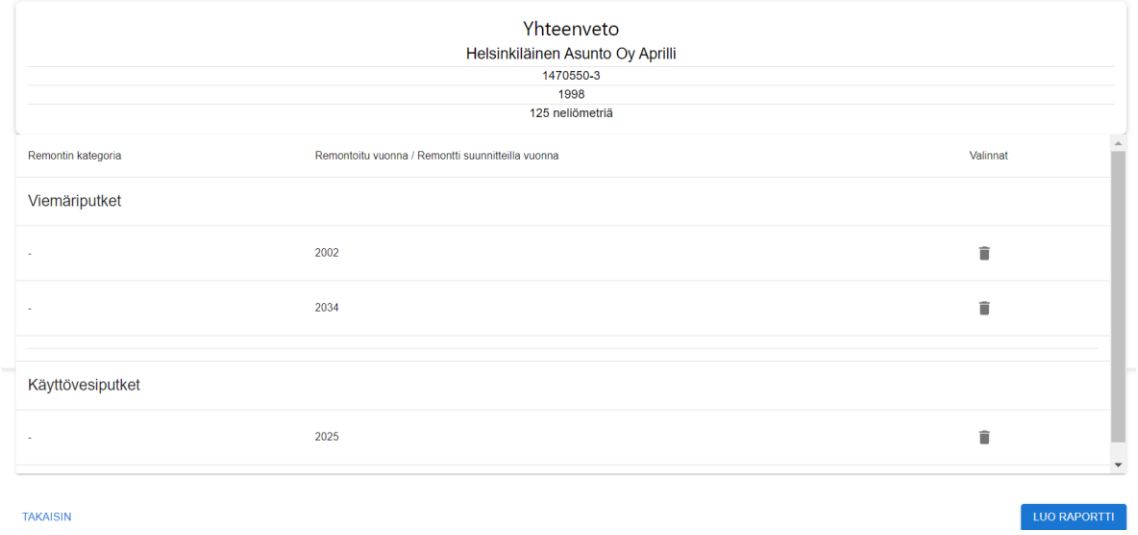
Taloyhtiöön tehdyt remontit eivät ole avointa dataa ainakaan vielä, joten tietojen oikeellisuuden suhteen olemme täysin käyttäjän syötteiden armoilla.

Kun käyttäjä tallentaa syötetyt remontit, siirtyvät remontit näkymään listaan. Kuvassa 19 näkyy komponenttilista, josta käyttäjä pystyy näkemään kaikki hänen lisäämänsä remonttinsa, sekä tarpeen vaatiessa poistamaan yksittäisen remontin.

Kuva 19. Kolmannen vaiheen näkymä, jossa käyttäjän lisäämät remontit listassa.

Neljännessä loppukatsausvaiheessa käyttäjä näkee kaikki syötteet, mitkä on valittu ja annettu, ja hän pystyy ostoskorimaisesti poistamaan listasta esimerkiksi väärin syötettyjä remontteja. Sivun navigaatiolla pääsee myös aikaisempiin näkymiin lisäämään tai muuttamaan asioita, jotka näkyvät loppukatsauksessa. Navigaation ”seuraava”-painike muuttuu tässä näkymässä ”Luo raportti” -painikkeeksi, joka painaessa lähettää syötetyt tiedot palvelimelle käsiteltäviksi.

Kuvassa 20 näkyy viimeisen vaiheen näkymä, josta käyttäjä voi vielä viimeisen kerran tarkistaa syöttämänsä tiedot, ennen kuin hänelle luodaan raportti. Virheen huomatessaan käyttäjä voi vielä palata edelliseen näkymään muokkaamaan tietoja. Kun käyttäjä on tarkistanut, että tiedot pitävät paikkaansa, voi hän painaa ”Luo raportti” -painiketta, joka luo ja ohjaa käyttäjän valmiin raportin sivulle.



Kuva 20. Viimeisen askeleen näkymä.

4.2.4 Omat raportit

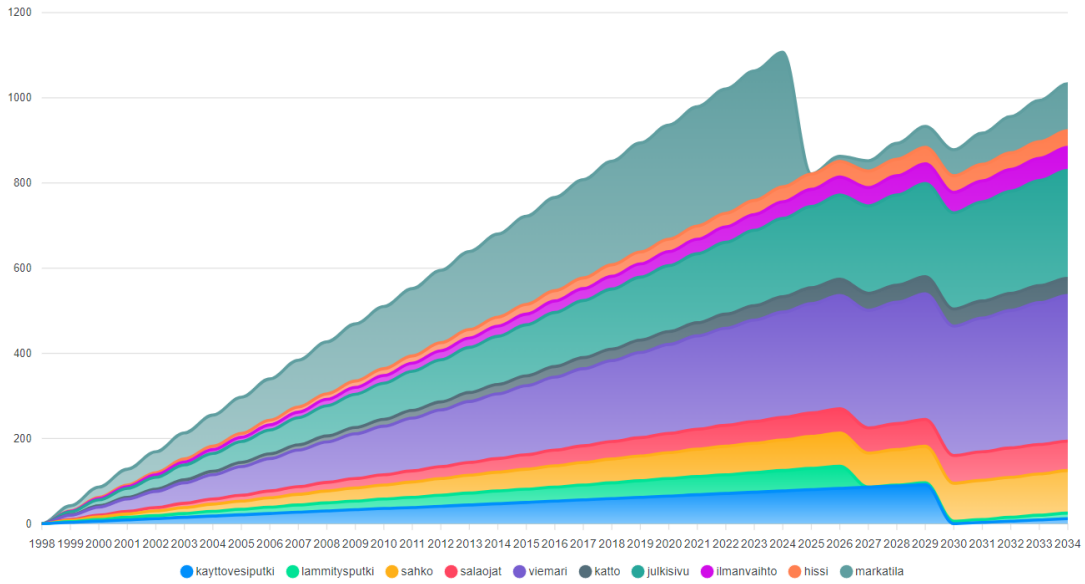
Sivustolla käyttäjä näkee listan kaikista luoduistaan raporteista taloyhtiön perusteella. Tunnisteena raportista käyttäjä näkee, mistä taloyhtiöstä raportti on luotu. Kun käyttäjä napauttaa “Tarkastele”-painiketta, hänet ohjataan sivulle, joka lataa ja näyttää kyseisen raportin tiedot. Sivun avautuessa sovellus hakee raportin tietokannasta ja lataa sen käyttäjän nähtäväksi. Kuva 21 on näkymä sivusta.

Kuva 21. Omat raportit -sivu.

4.2.5 Raporttipohja

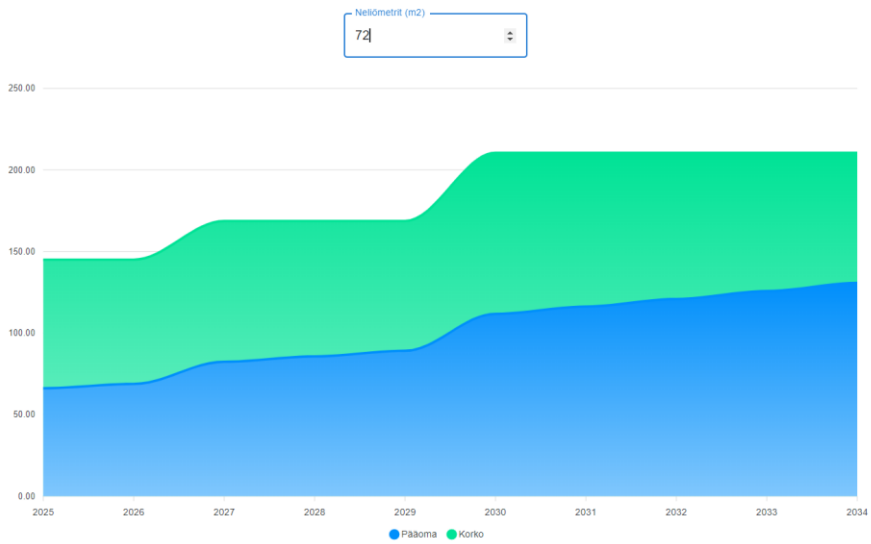
Raportti koostuu kahdesta kaaviosta ja taulukosta. Ensimmäinen kaavio esittää taloyhtiön korjausvelan rakennusvuodesta 10 vuoden suunnitelman mukaisesti.

Kaaviosta (Kuva 22) käyttäjä näkee selkeästi, milloin korjausvelka on suurimmillaan ja missä vaiheessa remontteja on tehty. Esimerkkikaaviosta näkee, että märkätilojen remontti sijoittuu vuodelle 2025, ja siitä syystä kaaviossa on lasku, joka merkitsee, että korjausvelka on laskenut remontin verran.



Kuva 22. Raportin ensimmäinen kaavio taloyhtiön korjausvelasta.

Kaavio (Kuva 23) esittää annuiteettikaavion, jossa käyttäjä voi määrittää syöttökenttään huoneiston pinta-alan, jota hän haluaa tutkia. Kaavio kuvaa raportissa kerrotun, aikaisemman kaavion (Kuva 22) kuvaaman, korjausvelan vaikutusta lainaan, jonka taloyhtiön pitäisi ottaa, jotta korjausvelka saataisiin katettua. Tämän jälkeen käyttäjä saa suuntaa antavan kaavion, joka näyttää, miten yhtiövastike nousee remonttien myötä ja mistä hinnankorotus koostuu. Käyttäjä voi helposti tämän perusteella nähdä esimerkiksi, miten eri huoneistojen koko vaikuttaisi hänen yhtiövastikkeensa.



Kuva 23. Kaavio annuiteettilainasta 72 neliön asunnossa.

Raportin viimeisessä osassa on taulukko (Kuva 24), joka näyttää taloyhtiön kunnon. Kunto on merkitty kolmella valolla: vihreä, keltainen ja punainen. Vihreä valo osoittaa, että yksittäisen kohteen elinikä on yli 75 %, keltainen yli 25 % ja punainen alle tämän. Taulukossa esitetään käyttäjän syöttämät remontit sekä korjauskohteiden hinnat eriteltynä neliömetriä kohden.

	Kategoria	Rakennettu / Remontoitu	Suunniteltu korjausvuosi	Korjausvelka €/m2
🟡	kayttovesiputki	1998	2030	74
🟡	lammitysputki	1998	2027	46
🟡	sahko	1998	-	69
🟡	salaojat	1998	-	51
🟡	viemari	1998	-	238
🔴	katto	1998	-	34
🟡	julkisivu	1998	-	177
🟡	ilmanvaihto	1998	-	37
🔴	hissi	1998	-	33
🔴	markatila	1998	2025	304

Kuva 24. Taulukko taloyhtiön eri osioiden kunnosta.

Käyttäjällä on mahdollisuus tallentaa raportti PDF-muodossa. Tämä on toteutettu käyttämällä react-pdf/renderer-kirjastoa. Kaaviot tallennetaan ensin PNG-muodossa, jonka jälkeen ne muunnetaan Base64-muotoon ja lähetetään PDF-komponentille. Kirjasto tukee vain Base64-muotoisia kuvia.

Koodiesimerkki 12 kuvaa, miten PDF-tiedoston ensimmäinen sivu on rakennettu. Sivu koostuu kahdesta kaaviosta.

```
<Page size="A4" style={styles.page}>
    <View style={styles.section}>
      <Text style={styles.header}>
        Taloyhtiön korjausvelka ja annuiteetti
      </Text>
      <Image style={styles.chartImage} src={debtChartImage1} />
      <Image style={styles.chartImage} src={debtChartImage2} />
    </View>
  </Page>
```

Koodiesimerkki 12. Yhden sivun luominen react-pdf/renderer kirjastolla.

4.3 Azure

Hyödynnämme projektissa Microsoftin Azure (32) pilvipalveluita niiden laajan suosion ja työmarkkinoilla arvostetun aseman vuoksi. Valintaan vaikutti myös se, että Microsoft tarjoaa opiskelijoille ilmaisia palveluita, sekä 100 euron edestä Azure-krediittejä, joiden avulla voi tutustua maksullisiin palveluihin ja saamme mahdollisuuden kokeilla niitä projektiin ilman ylimääräisiä kustannuksia.

Azure on Microsoftin kattava pilviteknologia-alusta, joka mahdollistaa erilaisten sovellusten ja palveluiden kehittämisen, hallinnan ja käytön pilvessä. Alustaan kuuluu laaja valikoima palveluita, kuten virtuaalikoneet, tietokannat, tallennustila, tekoälyratkaisut, analytiikkatyökalut sekä tietoturva- ja verkkoratkaisut. (33.)

Azuren tarjoama tietokantaratkaisu on verrattavissa MongoDB:hen ja toimii samalla tietokanta-ajurilla, jota projekti tällä hetkellä hyödyntää. Emme kuitenkaan näe vielä tarvetta siirtyä tähän vaihtoehtoon. Molemmat tietokannat ovat kehitysvaiheessa ilmaisia ja sisältävät samankaltaisia ominaisuuksia, jotka tukevat tehokasta kehittämistä. Koska eroavaisuudet eivät ole merkittäviä kummankaan ratkaisun välillä, emme koe tarpeelliseksi tehdä pysyvää valintaa tässä vaiheessa.

Käytämme sovellukseen aluksi vain ilmaisia palveluita, jotka on tarkoitettu sovelluksen kehitykseen, jotta emme käytä Azuren tarjoamia krediittejä. Säästämme Azure-krediitit projektin jälkeisiin kehitysvaiheisiin, kuten sovelluksen skaalautuvuuden testaamiseen sekä jatkokehityksen ja tuotantoon siirtymisen tueksi.

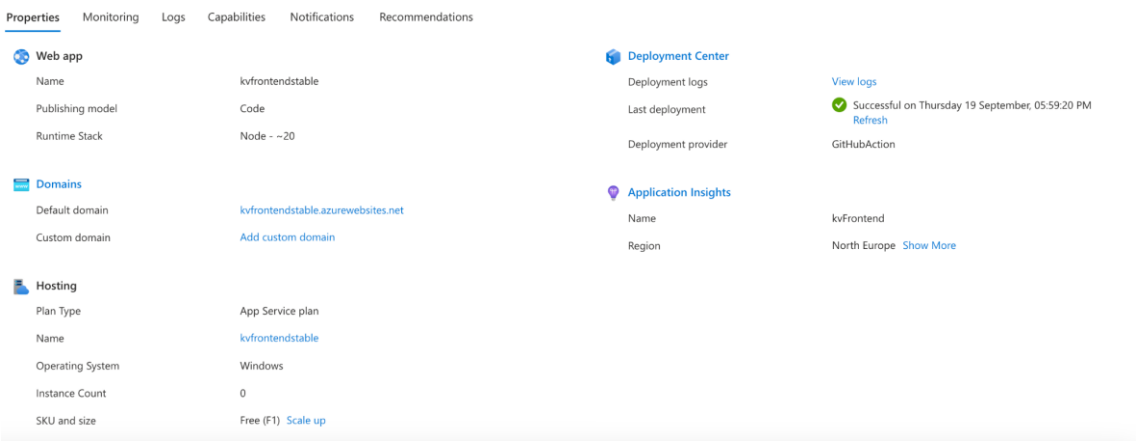
4.3.1 Azure Web Service

Julkaisemme sovelluksen Azure Web Service -palvelussa, joka on osa Azure App Service -alustaa, mikä mahdollistaa sovellusten helpon ja skaalautuvan ylläpidon ilman tarvetta hallita omaa palvelininfrastruktuuria (34). Palvelu sisältää myös Azuren tarjoaman verkkotunnuksen kehittämistä varten, jossa yhteys verkkosivuun on https-suojauksen takana ja sisältää SSL-varmenteen. Kuva 25 näyttää Azuren tarjoaman hallintapaneelin julkaistulle käyttöliittymälle, josta näkee nopeasti palvelun tiedot.

Palveluun julkaistu sovellus kontitetaan, eli se paketoidaan ja ajetaan virtuaalisessa ympäristössä, kontissa. Kontti sisältää kaikki tarvittavat riippuvuudet, kirjastot ja asetukset, jotta sovellus voi toimia eristetyssä ympäristössä, riippumatta siitä, missä alustassa tai palvelimessa sitä suoritetaan. Tämä tekee sovelluksesta helposti siirrettävän ja toistettavasti käynnistettävän missä tahansa pilvipalvelussa tai kehitysympäristössä.

Kun sovellus kontitetaan, se käynnistetään määritetyn ohjelmointikielen, arkkitehtuurin ja version perusteella. Esimerkiksi, Java-sovelluksen kohdalla kontti sisältää Java Runtime Environmentin (JRE), joka suorittaa käännetyn *.jar-tiedoston. Vastaavasti, käyttöliittymä-kehysten, kuten Reactin, kanssa konttiin

asetetaan build-kansion sisältö, jonka avulla sovellus ajetaan virtuaalipalvelimena. Näin sovellus voidaan julkaista, ja se näkyy ulospäin valmiina palvelemaan käyttäjiä.



Kuva 25. Web-sovelluksen määritysten pikanäkymä.

Palvelun kontitukset ja skaalautuminen ovat täysin automatisoituja, ja niitä hallitaan Azuren hallintaportaalin kautta. Voimme määrittää itse ympäristöasetukset sekä instanssien enimmäismäärän, minkä jälkeen Azure tekee skaalautumisen automaattisesti. Tämä sopii hyvin esimerkiksi pienempiin sovelluksiin tai palveluihin, joissa ei ole monimutkaista logiikkaa tai montaa palvelua.

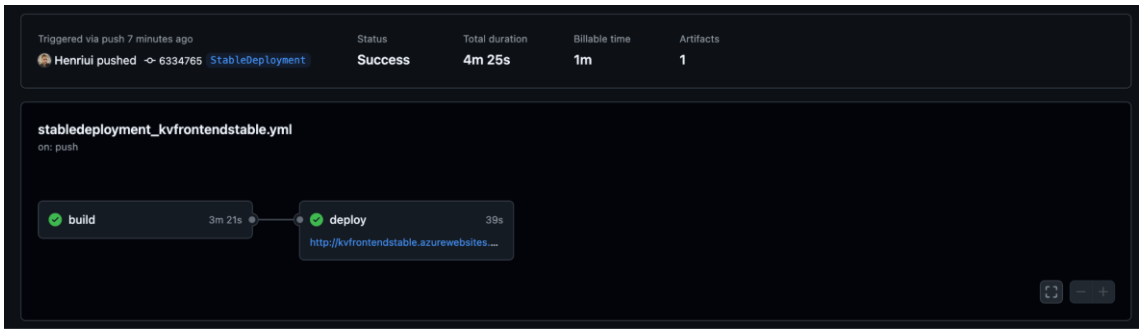
Skaalan kasvaessa tai sovelluksen muuttuessa mikropalvelupohjaiseksi tämä ei välttämättä riitä, koska emme pysty itse määrittämään docker määritystiedostoja kontituksia varten. Tämä automaattinen skaalaus riittää tämänhetkiseen sovellukseen, mutta Azure tarjoaa myös palveluita, joissa on enemmän kykyä hallita julkaisua esimerkiksi lataamalla omia docker-määritystiedostoja, ja tekemällä oman skaalaussuunnitelman.

4.3.2 CI/CD

Azure tukee jatkuvaa integraatiota ja julkaisua (CI/CD) monin eri tavoin, ja yksi tehokkaimmista tavoista hyödyntää näitä ominaisuuksia on automaattisten työnkulkujen käyttöönotto GitHub Actionsin (35) avulla. Azure Pipelinesin ohella GitHub Actions tarjoaa Azure-palveluiden integroinnin suoraan versionhallinnan

kanssa, jolla automatisoimme sovellusten rakentamisen, testaamisen ja julkaisun helposti (36).

Yhdistimme GitHub-repositorion Azureen, joka generoi automaattisesti GitHub Actions -työnkulun tiedoston, joka sisälsi alustavat vaiheet koodin julkaisuprosessiin. Lisäsimme polut kansioihin, koska meillä on koko projekti saman gitin alla, sekä ympäristömuuttujat, joita sovellukset käyttävät yhteyksien luomiseen, salauksiin ja muihin asioihin, joita emme halua näyttää lähdekoodissa. Kuva 26 näyttää onnistuneen julkaisun sivustolla.



Kuva 26. GitHub Actions -näkymä projektin julkaisuautomaatiosta.

Julkaisuprosessi menee seuraavasti:

1. Sovelluksen rakentaminen ja paketoiminen:
    1. Actions hakee repositoriosta lähdekoodin.
    2. Ympäristömuuttujien lisääminen GitHubista ympäristöön.
    3. Sovellus käännetään määritetyllä komennolla.
    4. Build kansio pakataan zip tiedostoksi.
    5. Zip tiedosto ladataan artefaktiksi, joka tallentuu pipelineen.
2. Sovelluksen julkaisu:
    1. Sovelluksen artefakti haetaan nimellä, joka määritettiin paketointivaiheessa.
    2. zip tiedosto puretaan ja tuhotaan.
    3. Actions kirjautuu Azureen, ja alkaa lataamaan tiedostoa Azure Pipelinesiin, josta se julkaistaan määritettyyn Deploymenttiin.
    4. Azure ilmoittaa sovelluksen tilan ja palauttaa linkin sivulle.

Tällä hetkellä ainoastaan StableDeployment-niminen julkaisu git-haara käännetään ja testataan GitHub Actionsin avulla. Suunnittelemme laajentavamme automaatioprosessin koskemaan kaikkia haaroja, mutta tämä edellyttää muutoksia, kuten julkaisuvaiheen poistamista muista haaroista. Myöhemmin, kun siirrymme käyttämään Azuren korkeampaa palvelutasoa, voimme luoda erillisiä julkaisuversioita testausta varten. Tällä hetkellä ilmainen palvelutaso tarjoaa kuitenkin vain yhden julkaisuympäristön.

5 Jatkokehitys

Ensimmäisenä jatkokehitysideana haluaisimme muuttaa tiedonkeruun ensimmäisen vaiheen näyttämään koosteen taloyhtiön kunnosta, ja listan taloyhtiöstä luoduista raporteista. Tähän vaiheeseen haluamme hyödyntää koneoppimista, jotta voimme näyttää käyttäjälle vain mahdollisimman relevantit ja luotettavat raportit, ja koosteen niiden tiedoista. Tämä parantaisi käyttäjäkokemusta vähentämällä väärin syötetyn datan määrää, ja auttaisi käyttäjiä vertailemaan myös muita taloyhtiöitä nopeasti ilman, että joutuu käymään useita raportteja läpi.

Toisena kehityskohteena haluamme ottaa käyttöön maksumuurin, joka rajoittaa ilmaistilien toimintoja. Tavoitteena olisi estää ilmaistilien käyttäjiä näkemästä kaikkia tietoja valmiista raporteista. Rajoittaisimme myös käyttäjien mahdollisuutta luoda omia raportteja asettamalla kuukausittaisen rajoituksen sekä raportin sisällön rajoituksen. Maksullisten tilien käyttäjät voivat sen sijaan luoda esimerkiksi useita kymmeniä raportteja kuukaudessa ja saisivat pääsyn myös valmiiden raporttien koko tietoihin sekä pdf tallennukseen.

Viimeisenä tasona olisi tehokäyttäjille suunnattu paketti, jossa ei ole rajoituksia raporttien määrälle. Tämä taso on tarkoitettu henkilöille, jotka käyttävät sovellusta paljon ja säännöllisesti. Tämä olisi erityisen hyödyllinen esimerkiksi kiinteistövälittäjille, sillä he voisivat tehdä raportin jokaisesta kohteesta ja esitellä sen asunnosta kiinnostuneille henkilöille. Tämä voisi edistää myyntiä, sillä ostajille voisi selkeästi osoittaa kustannuksia seuraavan 10 vuoden ajalta.

Viimeisenä jatkokehitysideana haluamme rakentaa sivusta modernimman näköisen ja responsiivisemman. Haluamme tarjota käyttäjille mahdollisimman

helppokäyttöisen sivuston, jotta jokainen käyttäjäryhmä voisi käyttää sovellusta ilman minkäänlaista osaamista. Esimerkiksi erilaiset auttavat ikkunat, jossa käyttäjä voisi katsoa, mistä vaikkapa remontin tieto pitäisi löytyä isännöitsijätodistuksesta.

6 Yhteenveto

Insinöörityössä suunnittelimme ja toteutimme MVP-version web-sovelluksesta korjausvelan määrittämiseen, huoneistokohtaisen annuiteettilainan selvittämiseen. Projektin aikana opimme paljon Spring Boot -palvelinpuolen kehityksestä, kuten autentikointi- ja autorisointipalveluiden luomisesta, REST-kutsujen rakentamisesta, suodatus- ja polkusuojausten toteuttamisesta sekä API-suunnittelun parhaista käytännöistä.

Tällä hetkellä asunnon ostajat saattavat olla täysin tietämättömiä taloyhtiöön tulevista remonteista. Tunnetuin remontti suomalaisille on putkiremontti, joten ensiasunnon ostajat saattavat olla täysin tietämättömiä muista pakollisista remonteista. Tietämätön asunnon ostaja saattaa saada ikävän yllätyksen, kun yhtiövastike nouseekin yllättäen. Tästä syystä halusimme luoda hyvin yksinkertaisen ja helposti lähestyttävän ensiaskeleen asunnon ostajalle, jotta käyttäjät saisivat mahdollisimman paljon tietoa taloyhtiön kunnosta, sekä kuinka paljon eri huoneistojen yhtiövastike tulisi nousemaan remonttien sattuessa.

Halusimme, että sivustolla pystyisi helposti luomaan raportteja, tarkastelemaan niitä ja lataamaan ne helposti tietokoneelle. Sivuston arkkitehtuuriksi vertailimme monen erilaisen arkkitehtuurin välillä, ja saimme arvokasta tietoa erilaisista vaihtoehdoista. Sovelluksen käyttöliittymän arkkitehtuuriksi valitsimme MVC:n, jonka valinta mahdollisti koko sovellukselle yhtenäisen arkkitehtuurin.

Sovelluksen teknologia pinoksi halusimme asettaa itsellemme uuden oppimistavoitteen, ja päädyimme rakentamaan taustapalvelun meille uudella teknologialla, Spring bootilla. Spring boot mahdollisti sovellukselle myös nopeamman tavan REST API -kutsujen ja taloyhtiön teknisten kohteiden korjausvelka laskujen käsittelyyn kuin esimerkiksi Express. Pystyimme asettamaan helposti erilaisille

poluille suojauksen käyttäen http-kutsuille filtteriä, joka automaattisesti tarkisti määrätyille poluille autentikaation.

Tietokantaratkaisuksi valittu MongoDB soveltui hyvin sekä käyttäjätietojen että heidän luomiensa raporttien tallentamiseen. MongoDB:n helppo integraatio teki siitä hyvän vaihtoehdon tälle projektille, ja pidämme toistaiseksi datan sinne tallennettuna.

Projekti vastasi kokonaisuudessaan odotuksiamme. Vaikkei projektin pääpainopiste ollut käyttöliittymän kehityksessä, onnistuimme luomaan tarvittavat ominaisuudet sekä näkymät. Esimerkiksi halusimme luoda yleisnäkymään käyttäjälle interaktiivisen raporttin, jotta käyttäjä pystyy helposti katsomaan, mistä raportti koostuu, ennen kuin hänen tarvitsee rekisteröityä sivustoon.

Vaikka kaikkia suunniteltuja ominaisuuksia ei ehditty toteuttaa projektin rajallisen aikataulun puitteissa, onnistuimme toteuttamaan yksittäiselle käyttäjälle kaikki haluamamme toiminnot.

Sovelluksen kehittäminen jatkuu insinöörityön jälkeen, painopisteenä verkkosivu-puolen kehitys, jotta sivustosta saisi modernin näköisen ja helposti lähestyttävän. Haluamme, että kaiken ikäiset henkilöt pystyisivät käyttämään sivustoa ilman ongelmia. Tätä varten olemme suunnitelleet erilaisia käytettävyystestauksia, joita voisimme pitää. Testauksen ansiosta voisimme saada enemmän varmuutta, että sivustolla kulkeminen, sekä raportin luominen on mahdollisimman helppoa.

Tulevaa kehittämistä varten Azure-kehitysympäristö ja tätä varten luotu automatisoitu CI/CD-putki auttaa lähdekoodin muutosten julkaisu- ja testausprosesseissa, mikä nopeuttaa kehitystä ja vähentää manuaalisia virheitä. Tällä hetkellä testaus tehdään vain julkaisuhaaraan, mutta tämä voidaan myöhemmin laajentaa jokaiseen git-haaraan niin, että sitä ei voi liittää päähaaraan ilman, että testit menevät läpi.

Lähteet

1. Kiinteistölehti. Asukkaana taloyhtiössä. Verkkoaineisto. 2017. https://www.kiinteistolehti.fi/asukkaana-taloyhtiossa. Luettu 15.8.2024.
2. Tilander J. Kunnossapito ja vastuunjako asunto-osakeyhtiössä. Verkkoaineisto. 2020. https://kiinteistolakimies.fi/artikkelit/kunnossapito-ja-vastuunjako-asunto-osakeyhtiossa/. Luettu 1.6.2024.
3. Sustera Finland. Mikä on PTS. Verkkoaineisto. 2023. https://sustera.fi/ajankohtaista/asumisvinkit/pts-ja-usein-kysytyt-kysymykset/. Luettu 29.5.2024.
4. Herkulex. Korjausvelka ja sen merkitys asunto- ja kiinteistökaupassa. Verkkoaineisto. https://herkulex.fi/korjausvelka-ja-sen-merkitys-asunto-ja-kiinteistokaupassa/. Luettu 6.6.2024.
5. Kiinteistölehti. Taloyhtiöiden on aiempaa vaikeampi saada lainaa. Verkkoaineisto. 2021. https://www.kiinteistolehti.fi/taloyhtioiden-on-aiempaa-vaikeampi-saada-lainaa. Luettu 15.6.2024.
6. Simola U. Konkurssi hiipii taloyhtiöön. Verkkoaineisto. 2023. https://www.taloustaito.fi/koti/konkurssi-hiipii-taloyhtioon--mita-osakkeille-asukkaille-ja-talolle-tapahtuu/. Luettu 18.6.2024.
7. Tilastokeskus. Rakennuskustannusindeksi kustannuslajeittain, kuukausitiedot. Verkkoaineisto. https://pxdata.stat.fi/PxWeb/pxweb/fi/StatFin/StatFin__rki/statfin_rki_pxt_118p.px/. Luettu 1.7.2024.
8. Tilastokeskus. Asunto-osakeyhtiöiden lukumäärät rakennusvuosittain ja suuralueittain, 2009-2023. Verkkoaineisto. https://pxdata.stat.fi/PxWeb/pxweb/fi/StatFin/StatFin__asyta/statfin_asyta_pxt_13dj.px/. Luettu 1.7.2024.
9. Broadcom. Spring Boot pääsivu. Verkkoaineisto. https://spring.io/. Luettu 29.5.2024.
10. Korjausvelkalaskuri Oy. Korjausvelkalaskuri. Verkkoaineisto. https://korjausvelkalaskuri.fi/. Luettu 1.6.2024.
11. Arvonosturi. Korjausvelka alentaa kiinteistön arvoa. Verkkoaineisto. 2020. https://www.arvonosturi.fi/korjausvelka-alentaa-kiinteiston-arvoa/. Luettu 1.6.2024.

12. Rakennustietosäätiö. Kiinteistöjen tekniset käyttöiät ja kunnossapitojaksot. PDF. 2024. https://uutiset.rakennustieto.fi/wp-content/uploads/2024/03/RTS-24-2_KIINTEISTON-TEKNISET-KAYTTOIAT-JA-KUNNOSSAPITOJAKSOT.pdf. Luettu 13.7.2024.
13. Taimitarha O. Taloyhtiölaina omaksi? Verkkoaineisto. https://www.aktia.fi/fi/yhtiolaina-omaksi. Luettu 1.8.2024.
14. Insinööritoimisto suunnittelutekniikka Oy. PTS pitkän tähtäimen suunnitelma. Verkkoaineisto. https://www.sutek.fi/pts-pitkan-tahtaimen-suunnitelma. Luettu 1.8.2024.
15. Mozilla corporation. MVC. Verkkoaineisto. https://developer.mozilla.org/en-US/docs/Glossary/MVC. Luettu 1.6.2024.
16. N G. MVC, MVP, MVVM-arkkitehtuurien vertailu. Verkkoaineisto. 2023. https://blog.stackademic.com/understanding-mvc-mvvm-and-mvp-a-comprehensive-comparison-324fd6e3c730. Luettu 28.7.2024.
17. geeksforgeeks. Introduction to MVVM. Verkkoaineisto. 2023. https://www.geeksforgeeks.org/introduction-to-model-view-view-model-mvvm/. Luettu 1.8.2024.
18. Google. Angular pääsivu. Verkkoaineisto. https://angular.dev/. Luettu 1.7.2024.
19. Vyas A. Model View Presenter. Verkkoaineisto. 2018. https://anshulvyas380.medium.com/model-view-presenter-b7ece803203c. Luettu 1.7.2024.
20. Meta. Flux repositorio. Verkkoaineisto. https://github.com/facebookarchive/flux. Luettu 15.6.2024.
21. Abramov D. Redux dokumentaatio. Verkkoaineisto. https://redux.js.org/. Luettu 15.9.2024.
22. Meta Platforms Inc. React Native etusivu. Verkkoaineisto. https://reactnative.dev/. Luettu 29.5.2024.
23. OpenJS Foundation. ExpressJs pääsivu. Verkkoaineisto. https://expressjs.com/. Luettu 25.5.2024.
24. B R. Top 10 Backend Frameworks in 2024. Verkkoaineisto. 2024. https://medium.com/@talktorahul.b/top-10-backend-frameworks-in-2024-33da4269da39. Luettu 1.9.2024.

25. C M. Node.js vs Spring Boot vertailu. Verkkoaineisto. 2023. https://medium.com/deno-the-complete-reference/express-vs-springboot-hello-world-performance-comparison-dd066bf53858. Luettu 15.6.2024.
26. Broadcom. Using spring boot autoconfiguration. Verkkoaineisto. https://docs.spring.io/spring-boot/docs/2.0.x/reference/html/using-boot-auto-configuration.html. Luettu 18.7.2024.
27. Broadcom. Spring Boot generaattori. Verkkoaineisto. https://start.spring.io/. Luettu 1.8.2024.
28. JavaTPoint. Spring Boot Architecture. Verkkoaineisto. https://www.javatpoint.com/spring-boot-architecture. Luettu 5.8.2024.
29. Patentti- ja rekisterihallitus. Yritys- ja yhteisötietojärjestelmä rekisteri. Verkkoaineisto. https://avoindata.prh.fi/. Luettu 3.6.2024.
30. Material UI SAS. Material UI kirjasto pääsivu. Verkkoaineisto. 2024. https://mui.com/. Luettu 1.6.2024.
31. Patentti- ja rekisterihallitus. Patentti- ja rekisterihallitus pääsivu. Verkkoaineisto. 2024. https://www.prh.fi/fi/index.html. Luettu 28.7.2024.
32. Microsoft. Azure etusivu. Verkkoaineisto. https://azure.microsoft.com/. Luettu 1.7.2024.
33. Bergius K. Mikä se azure oikein on? Verkkoaineisto. 2023. https://sulava.com/pilvi-infrastruktuuri/mika-se-azure-oikein/. Luettu 1.8.2024.
34. Microsoft. Azure web app service. Verkkoaineisto. 2024. https://azure.microsoft.com/en-us/products/app-service/web. Luettu 1.8.2024.
35. Microsoft. Github Actions dokumentaatio. Verkkoaineisto. https://docs.github.com/en/actions. Luettu 1.8.2024.
36. Microsoft. Github Actions Azure. Verkkoaineisto. 2024. https://learn.microsoft.com/en-us/azure/developer/github/github-actions. Luettu 15.7.2024.
37. Ciclum. Kuinka tärkeää arkkitehtuuri on sovelluskehityksessä. Verkkoaineisto. 2024. https://www.ciklum.com/resources/blog/expert-advice-how-important-is-architecture-in-software-development. Luettu 25.7.2024.