Jaku Lahtinen & Jetro Puranen

FULL STACK-SOVELLUSKEHITYS .NET & REACT-TEKNOLOGIOILLA

Clean Architecture -mallin mukainen sovelluksen rakentaminen ja julkaisu Azure-ympäristössä

Opinnäytetyö

Tekniikan ammattikorkeakoulututkinto

Ohjelmistotekniikan koulutus

2024

Kaakkois-Suomen ammattikorkeakoulu

Kaakkois-Suomen ammattikorkeakoulu

Tutkintonimike	Insinööri (AMK)
Tekijä/Tekijät	Jaku Lahtinen ja Jetro Puranen
Työn nimi	Full-Stack-sovelluskehitys .NET- ja React-teknologioilla – Clean Architecture -mallin mukainen sovelluksen rakentaminen ja julkaisu Azure-ympäristössä
Toimeksiantaja	Kaakkois-Suomen ammattikorkeakoulu
Vuosi	2024
Sivut	90 sivua
Työn ohjaaja(t)	Tuomas Reijonen

TIIVISTELMÄ

Opinnäytetyön aiheena on Full-Stack-sovellus X-Sähkö, joka on rakennettu .Net- ja React- teknologioilla hyödyntäen Clean Architecture -mallia. Opinnäytetyön tavoitteena on kertoa, millä teknologioilla ja miten X-Sähkö-palvelu on kehitetty ja julkaistu kaikkien saatavaksi. Opinnäytetyössä käsitellään sovelluksen kehityksessä käytettyjä teknologioita, ohjelmistokehityksen arkkitehtuurimalleja sekä sovelluksen julkaisuprosessiin liittyviä ratkaisuja. Opinnäytetyössä halutaan korostaa versionhallinnan, laadunvarmistuksen sekä erilaisten arkkitehtuurimallien tärkeyttä ohjelmistokehityksessä.

Modernit teknologiat mahdollistavat nykyään helpon ja tehokkaan tavan sovelluksen testaamiseen, laadunvarmistamiseen ja julkaisuun. Työssä esitellään X-Sähkö-sovelluksessa käytettyjä kehitysympäristöjä, kuten .NET- ja React-teknologioita. Entity Framework Core ja ORM- tekniikat mahdollistavat helposti rakennettavan ja ylläpidettävän tietokantatoteutuksen C#-ohjelmointikieliluokkien avulla. Sovelluksen ylläpito ja testattavuus varmistetaan yksikkötestien sekä arkkitehtuurimallien, kuten Clean Architecture -mallin avulla. Selkeä arkkitehtuuri jakaa sovelluksen omiin osa-alueisiin, mikä helpottaa ominaisuuksien lisäämistä tai poistamista. Git-Actions-työkalun avulla voidaan luoda esimerkiksi testaus- ja julkaisuautomaatioita, jotka parantavat laadunvarmistusta sekä testaamista. Microsoftin omistama Azure-pilvipalvelu ja sen ratkaisut mahdollistavat helpon sovelluskokonaisuuden julkaisun ja hallittavuuden sekä tietokantatoteutuksen.

Teoriassa esitettyjen teknologioiden käyttöä kuvaillaan toteutus osiossa. Toteutusosiossa käydään läpi sovelluksen käyttöliittymän toteutusta sekä palvelinpuolen arkkitehtuuria ja laskentalogiikkaa. Tietokantatoteutus ja jatkuva uusien sähkönhintatietojen hakeminen ja tallentaminen tietokantaan esitetään toteutusosiossa. Toteutusosio myös sisältää kaavioita ja kuvaavaa tekstiä siitä, miten Clean Architecture -mallissa data käsitellään ja palautetaan käyttäjälle, kun palvelinpuoli ottaa sen vastaan. Toteutusosio sisältää myös ohjeita siitä, miten käyttöliittymä ja palvelinpuolen toteutus voidaan julkaista Azure-ympäristöön. Laadunvarmistusta kuvaillaan toteutus osiossa esittämällä yksikkötestejä sekä putkilinjoja, jotka mahdollistavat automaattisen sovelluksen julkaisun Azure-ympäristöön ja ohjelmakoodin laadun varmistamisen sekä virheiden kartoittamisen hyvissä ajoin.

Asiasanat: ohjelmistokehitys, ohjelmistoarkkitehtuuri, pilvipalvelut, ohjelmistotestaus

South-Eastern Finland University of Applied Sciences

Degree title	Bachelor of Engineering
Author (authors)	Jaku Lahtinen & Jetro Puranen
Thesis title	Full-Stack application development with .NET and React technologies – Building and deploying an application following the Clean Architecture model in an Azure environment
Commissioned by	South-Eastern Finland University of Applied Sciences
Time	2024
Pages	90 pages
Supervisor	Tuomas Reijonen

ABSTRACT

The thesis topic is a Full-Stack application X-Sähkö, built using .NET and React technologies, utilizing the Clean Architecture model. The goal of the thesis is to explain the technologies and methods used to develop and publish the X-Sähkö service for public access. The thesis discusses the technologies used in the application's development, software development architecture models, and the solutions related to the application's publishing process. The thesis aims to emphasize the importance of version control, quality assurance, and various architectural models in software development.

Modern technologies today facilitate efficient application testing, quality assurance, and publishing. This thesis explores the X-Sähkö application's development environment, leveraging .NET, React, and Entity Framework Core for maintainable database implementation via C#. Clean Architecture ensures modularity, making features easier to manage, while unit tests enhance maintainability. GitHub Actions streamlines testing and merging code safely. Microsoft Azure simplifies application deployment, management, and database integration

The usage of the technologies presented in theory is described in the implementation section. The implementation section goes through the implementation of the application's user interface, the server-side architecture, and the computational logic. The database implementation and the continuous retrieval and storage of new electricity price data in the database are presented in the implementation section. The implementation section also includes diagrams and descriptive text on how data is processed and returned to the user in the Clean Architecture model when the server side receives it. The implementation section also contains instructions on how to publish the user interface and server-side implementation section to the Azure environment. Quality assurance is described in the implementation section by presenting unit tests and pipelines that enable the automated publishing of the application to the Azure environment, as well as ensuring code quality and identifying errors in good time

Keywords: software development, software architecture, cloud services, software testing

SISÄLLYS

1 JOHDANTO

Tämän opinnäytetyön aiheena on kertoa X-Sähkö-palvelun suunnittelusta, rakentamisesta ja toteuttamisesta ensin pohjustamalla sovelluksessa käytettyjä tekniikoita teoriapohjalta, jonka jälkeen kertomalla toteutuksesta. Sovellus rakennettiin syventävän harjoittelujakson aikana. Opinnäytetyön tarkoituksena on kertoa, mitä nykyaikaisia moderneja teknologioita voidaan käyttää rakentaessa sovellusta. Lopussa kerromme sovelluksen toteutuksesta, julkaisuprosessista, sekä laadunvarmistuksesta. Kehitetyn sovelluksen avulla saadaan myös kehitystehtäviä, sekä oppimateriaalia tuleville opiskelijoille.

Teoriaosuus käsittelee sovelluksessa käytettyjä tekniikoita ja teknologioita luvuissa 3–11. Tässä osiossa käydään läpi teoriassa sovelluksen kehitysympäristöä, versionhallintaa, arkkitehtuurimalleja, testausta ja laadunvarmistusta sekä pilvipalveluita, jotka mahdollistavat sovelluksen julkaisun. Toteutusosuudessa kerrotaan sovelluksen kehityksen alkuvaiheista, rakentamisesta ja lopullisesta tuloksesta. Luvussa 12.1 kerrotaan sovelluksen back-end logiikan rakentamisesta ja siitä, kuinka se on toteutettu, kun taas luku 12.2 sisältää käyttöliittymäkoodin toteutusta. Luvussa12.3 käsitellään sovelluksen julkaisemista Azure-ympäristössä ja viimeisenä kerrotaan valmiista sovelluksesta sekä siitä, kuinka se toimii ja palvelee käyttäjiä.

2 OHJELMISTON JA TYÖN IDEA

Ohjelmiston tavoitteena ja ideana on ollut rakentaa tavallisille kuluttajille työkalu, jonka avulla voidaan helposti selvittää, minkälainen sähkösopimustyyppi olisi sopiva vaihtoehto kuluttajalle itselleen. Tarkoituksena on ollut myös rakentaa esimerkkisovellus, jota voidaan hyödyntää opetuskäytössä mm. ominaisuuksien lisäämisellä. Sovellus sai alkunsa, kun harjoittelun sekä opinnäytetyön ohjaaja Tuomas Reijonen tutki mediassa olleita artikkeleita, joissa käsiteltiin mikä olisi paras sähkösopimustyyppi kuluttajalle.

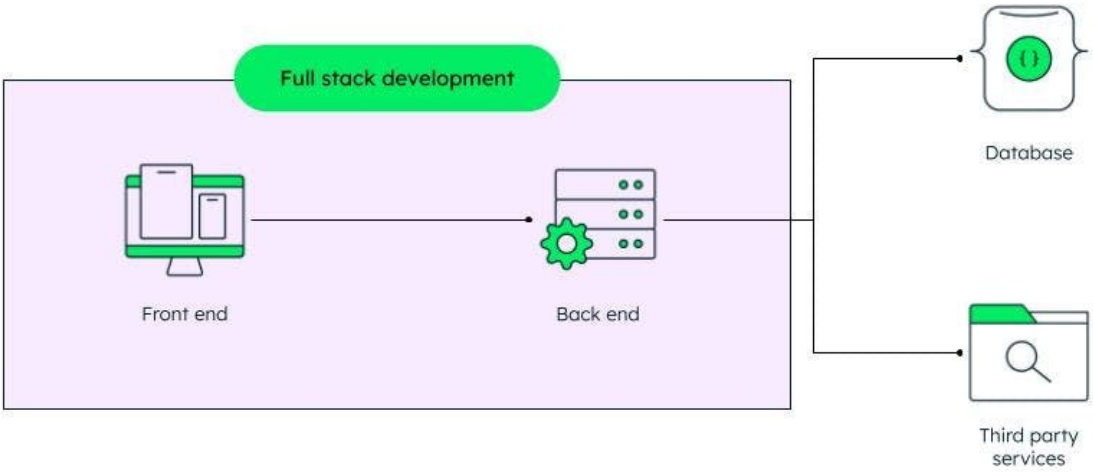
X-Sähkö-palvelussa käyttäjä voi ladata oman sähkönkulutustiedoston palveluun, joka voidaan ladata Fingridin avoimesta dataportaalista. Jokainen, joka omistaa Suomessa sähkösopimuksen pystyy lataamaan omat kulutustietonsa. Tällä työkalulla käyttäjä voi tarkastella omaa sähkönkulutustaan ja verrata sitä

valitsemaansa kiinteään hintaan sekä pörssisähkön hintaan ajanjaksolle, jolta tiedosto on ladattu. Laskurilla voidaan helposti ja nopeasti selvittää, kumpi sähkösopimustyyppi olisi parempi vaihtoehto, pörssisähkö vai kiinteä hinta.

X-Sähkö-palvelun arviointityökalulla voidaan selvittää arvioitu vuosittainen sähkönkulutus sekä kiinteän ja pörssisähkön hinta, jos käyttäjä haluaa tehdä vertailua sähkösopimustyyppien välillä ilman oman kulutustiedoston lataamista palveluun. Tässä työkalussa on otettu huomioon nykyaikaisia sähkönkulutukseen ja säästöön liittyviä asioita, kuten sähköautoilu ja aurinkopaneelit.

3 FULL STACK -OHJELMISTOKEHITYS

Full Stack -ohjelmistokehitys tarkoittaa front-endin (käyttäjälle näkyvän osan) ja back-endin (taustatoimintojen, kuten tietokantojen ja sovelluslogiikan) kehittämistä. Jokainen sovellus sisältää front-end-logiikan, joka sisältää käyttöliittymän ja siihen liittyvän koodin, sekä back-end-logiikan, joka sisältää sovelluksen toiminnan vaativan ohjelmakoodin mukaan lukien tietojärjestelmien integroinnin, datan käsittelyn ja kommunikoinnin muiden sovellusten kanssa. (Amazon s.a.)



Kuva 1. Full-Stack ohjelmistokehityksen rakenne (MongoDB s.a.)

Sanalla ”stack” tarkoitetaan teknologiapinoa, joka on joukko eri teknologioita, joita käytetään sovelluksen rakentamiseen. ”Stack” tai pino sisältää yhden tai useamman ohjelmointikielen, koodikirjastot, palvelimet, tietokannat, kehykset ja kehittäjä työkalut. (Steele 2020.) Kuvassa 1 esitetään Full Stack -ohjelmistokehityksen rakenteesta esimerkkikaavio.

3.1 Front-end

Verkkosivun tai sovelluksen ”Front-end” eli käyttöliittymä viittaa kaikkeen siihen, mitä verkkosivun tai sovelluksen käyttäjä voi nähdä palvelussa. Front-end-kehittäjien pääasialliset työtehtävät voidaan jakaa kahteen kategoriaan, UI (User Interface) - ja UX (User Experience) -tehtäviin. Kehittäjät työskentelevät suunnittelutiimien kanssa luodakseen helppokäyttöisen ja esteettisesti miellyttävän käyttöliittymän (UI) tai graafisen käyttöliittymän (GUI). (Coursera 2024a.)

Codeacademyn (2021) mukaan front-end-logiikan toimivuuden taustalla on monia siihen vaikuttavia asioita, kuten tietokanta-arkkitehtuuri, kehykset ja skaalautuvuusratkaisut. Front-end sisältää seuraavat asiat:

Tyylit: Tyylit sisältävät esimerkiksi painikkeet, tekstikentät, tekstit ja kuvat. Se on verkkosivuston ulkoasu.

Saavutettavuus: Front-end sisältää saavutettavuusominaisuudet, kuten puheentunnistuksen, tekstistä puheeksi -toiminnot. Saavutettavuus tekee verkkosivusta helppokäyttöisemmän ja se parantaa käyttökokemusta

Nopeus: Mitä nopeammin verkkosivu toimii, sitä parempi. Usein verkkosivun latautumista ei jäädä odottamaan pitkäksi aikaa.

Front-end ohjelmistokehittäjien tehtävä on pitää yhteyttä esimerkiksi asiakkaiden ja projektipäälliköiden kanssa, joilla on käsitys tai visio siitä, miltä sovelluksen käyttöliittymän tulisi näyttää.

Courseran (2024a) mukaan yleisimpiä front-end-kehityksessä käytettäviä skripti-, ohjelmointi- ja merkkauskieliä ovat:

- HTML

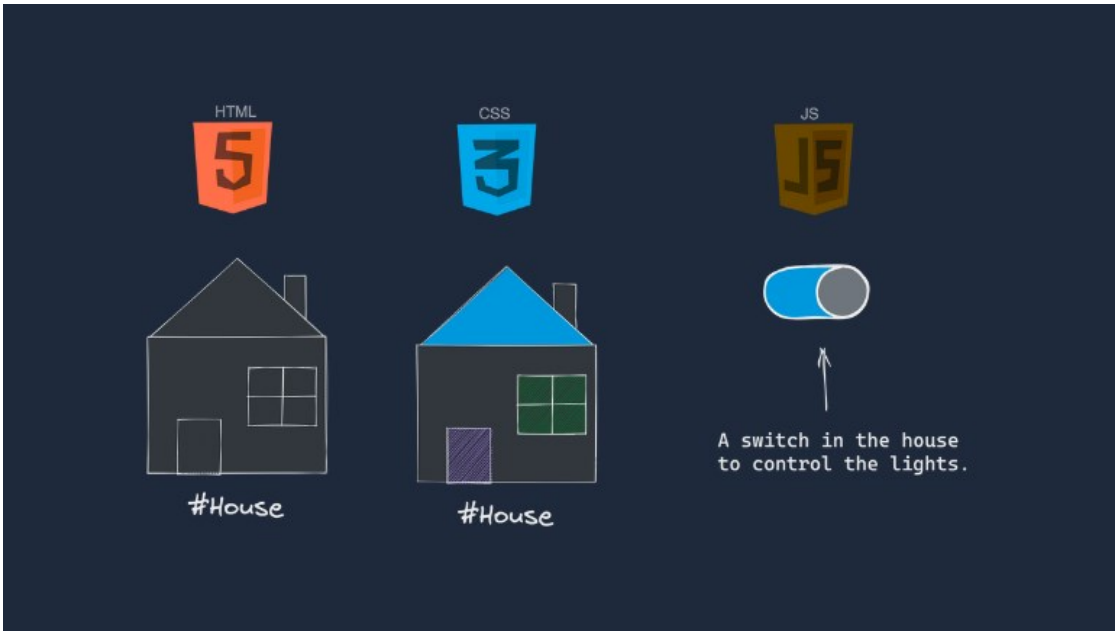
HTML eli Hypertext Markup Language on teknologia, joka luo rakenteen verkkosivulle. HTML sisältää tekstiä, kuvia ja hypertekstiä ja se mahdollistaa siirtymisen sivulta toiselle tarjoten käyttäjäystävällisen alustan asiakirjojen muotoiluun.

- CSS

CSS eli Cascading Style Sheets on kieli, jota käytetään verkkosivun ulkoasun muokkaamiseen. CSS voidaan käyttää esimerkiksi fonttien, sivun värien ja tekstien asetteluun.

- JavaScript

JavaScript on ohjelmointikieli, joka mahdollistaa interaktiivisien elementtien ja efektien luomisen sivulle.



Kuva 2. Esimerkki HTML, CSS ja JavaScript kielien rooleista (Pokhrel 2024)

Courseran (2024a) mukaan jokaista projektia kehittäjien ei tarvitse aloittaa alusta front-end-kehysten avulla. Kehys on ennalta kirjoitettu koodikirjasto, joka helpottaa yleisissä kehitys tehtävissä ja yleistason visuaalisissa elementeissä. Näihin kehyksiin kuuluu muun muassa:

- React JS
- Ember JS
- Sass
- Angular JS

- jQUery
- Flutter
- Semantic-UI
- Foundation
- Backbone.js

Nämä kehykset ja kirjastot tarjoavat kehittäjille työkaluja ja komponentteja, jotka helpottavat sovelluksen rakennusta ja joita voidaan käyttää front-end-kehityksessä.

3.2 Back-end

Back-end tarkoittaa ohjelmakoodia, joka suoritetaan palvelimella. Sen tehtävä on vastaanottaa pyynnöt asiakkailta (Clients) ja suorittaa sovelluslogiikkaa, joka määrittelee sen, minkälaista dataa tai tietoa asiakkaalle palautetaan. Back-end yleensä myös sisältää tietokannan, jonka tehtävä on tallentaa sovelluksen tietoja. Back-end sisältää kaiken teknologian, jota tarvitaan pyyntöjen vastaanottamiseen, vastausten tuottamiseen ja vastausten lähettämiseen takaisin asiakkaalle. Se koostuu tyypillisesti kolmesta osasta: (Codeacademy s.a.)

- Palvelin: Tietokone, joka vastaanottaa pyynnöt.
- Sovellus, joka sijaitsee palvelimella. Sovellus kuuntelee pyyntöjä, hakee dataa tietokannasta ja lähettää vastauksen takaisin asiakkaalle.
- Tietokanta: Tietokannat tallentavat ja käsittelevät dataa.

Courseran (2024b) mukaan Back-end kehityksessä yleisiä teknologioita ovat:

Ohjelmointikielet:

- Python
- PHP
- JavaScript
- Ruby
- Java
- C#

Kehykset:

- Node.js
- Django
- Spring
- Ruby on Rails
- Meteor

- Laravel

Tietokannat:

- MySQL
- Oracle
- MongoDB

Palvelimet:

- NGINX
- Apache
- Lighttpd
- Microsoft IIS

Mitä asiakkaat ovat?

Asiakkaat (Clients) ovat kaikkea sitä, jotka lähettävät pyyntöjä back-end palvelimelle. Usein asiakas on henkilön käyttämä selain. Muita asiakkaita voivat olla esimerkiksi mobiilisovellukset ja toisella palvelimella sijaitsevat sovellukset. (Codeacademy s.a.)

Mikä on tietokanta?

Tietokantoja käytetään sovellusten back-end logiikassa. Tietokannat tarjoavat rajapinnan tietojen tallentamista varten. Tietojen tallentaminen tietokantaan vähentää esimerkiksi palvelimen keskusmuistin kuormittamista, ja se mahdollistaa tiedon palauttamisen. Asiakas voi pyytää tietoja, jotka ovat tietokannassa, tai asiakas lähettää dataa pyynnön mukana, joka tallennetaan tietokantaan. (Codeacademy s.a.)

Mitä ovat back-end logiikan ydintoiminnot?

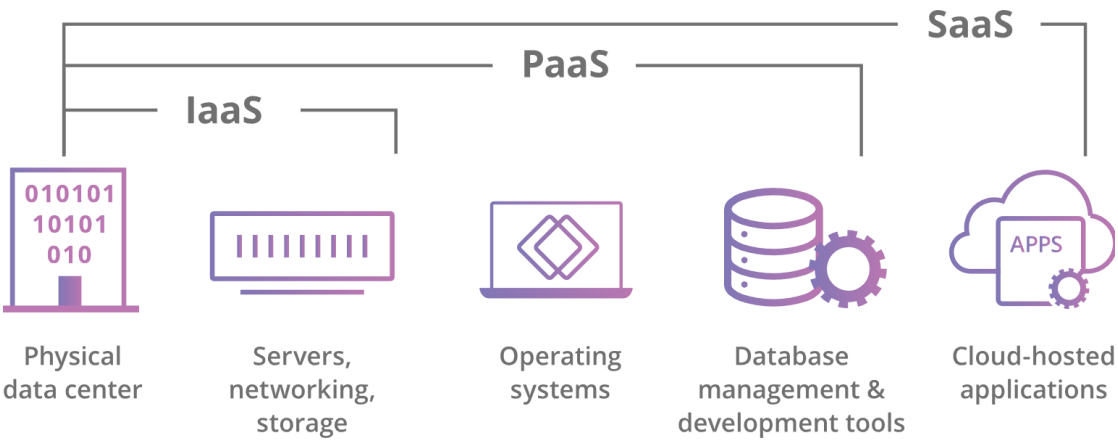
Palvelimella ajettava sovellus sisältää logiikan eri pyyntöjen vastaanottamiseksi HTTP-verbin ja yhtenäisen resurssitunnuksen URI:n (Uniform Recource Identifier) avulla. HTTP-verbiä ja URI yhdistelmää kutsutaan reitiksi ja niiden yhdistämistä pyyntöön tarkoitetaan reitityksekseksi. (Codeacademy s.a.)

Osa logiikan käsittelytoiminnoista ovat väliohjelmistoa (middleware). Väliohjelmisto on ohjelmakoodia, joka suoritetaan sen jälkeen, kun palvelin on saanut pyynnön ja ennen kuin palvelin palauttaa pyynnön. Väliohjelmisto voi esimerkiksi muokata pyyntöobjektia ja tehdä kyselyitä tietokantaan. Lopulta väliohjelmisto palauttaa HTTP vastauksen asiakkaalle. (Codeacademy s.a.)

Palvelin voi lähettää dataa asiakkaille eri muodoissa. Se voi tarjota esimerkiksi HTML-tiedoston, palauttaa JSON dataa, tai palauttaa vain HTTP-tilakoodin. Yksi esimerkki tilakoodista on ”404 – Not Found” tilakoodi, joka tarkoittaa sitä, että käyttäjä on yrittänyt siirtyä URI:iin, jota ei ole olemassa. On kuitenkin olemassa muitakin tilakoodeja, jotka kertovat mitä tapahtui palvelimen vastaanottaessa pyynnön. (Codeacademy s.a.)

3.2.1 Pilvipalvelut ja pilvilaskenta

”Pilvellä” tarkoitetaan palvelimia, joita käytetään internet-yhteyden kautta sekä niillä toimivia tietokantoja ja ohjelmistoja. Nämä palvelimet sijaitsevat eri datakeskuksissa ympäri maailmaa. Pilvilaskennan (Cloud Computing) avulla yritysten ja käyttäjien ei tarvitse hallita fyysisiä palvelimia itse tai suorittaa ohjelmistosovelluksia omilla laitteillaan. Pilvi mahdollistaa esimerkiksi sen, että käyttäjillä on pääsy samoihin tiedostoihin ja sovelluksiin miltä tahansa laitteelta, koska datan tallennus ja laskenta tapahtuu datakeskuksen palvelimilla. Pilvessä sijaitsevia resursseja kutustaan ”palveluiksi” ja palvelut voidaan jakaa erilaisiin kategorioihin eli palvelumalleihin. Pilvitoimittaja hallinnoi niitä aktiivisesti, ja niihin kuuluvat esimerkiksi infrastruktuuri, sovellukset, kehitystyökalut ja tietovarastot. (Cloudflare s.a.)



Kuva 3. Mitkä ovat pilvilaskennan pääpalvelumallit (Cloudflare s.a.)

Ohjelmisto palveluna (SaaS) tarkoittaa kaikkia sovelluksia, jotka on isännöity pilvipalvelimilla ja joihin käyttäjät pääsevät käsiksi internet yhteyden kautta. SaaS palveluita voi verrata asunnon vuokraamiseen, vuokranantaja omistaa talon ja ylläpitää sitä, mutta vuokralainen saa pääasiassa käyttää sitä niin kuin

omistaisi sen. Alusta palveluna (PaaS) -mallissa yritykset eivät maksa isännöidyistä sovelluksista, vaan sen sijaan yritykset maksavat resursseista, joita he tarvitsevat sovelluksien rakentamiseen. PaaS-toimittajat, kuten Microsoft Azure, tarjoavat kaikki mahdolliset resurssit, mitä tarvitaan sovelluksen valmistamiseen, mukaan lukien kehitystyökalut, infrastruktuurin ja käyttöjärjestelmät. Infrastruktuuri palveluna (IaaS) -mallissa yritys voi vuokrata tarvitsemaansa palvelin ja tallennustilaa pilvipalveluntarjoajalta. (Cloudflare s.a.)

4 OHJELMISTON KEHITYSYMPÄRISTÖ

4.1 .NET

.NET on ilmainen avoimen lähdekoodin sovellusalusta, jota Microsoft tukee. Se tarjoaa kevyen kehitysmallin ja joustavuuden eri työkalujen käyttämiseen, sekä eri käyttöjärjestelmäalustoihin. Kokonaisuudessaan .NET viittaa useisiin erilaisiin teknologioihin, kuten .NET Core, ASP.NET Core ja Entity Framework Core. (Microsoft 2024.)

Tämä sovellusalusta käyttää monia .NET-yhteensopivia ohjelmointikieliä, mutta yleisimmin C#-ohjelmointikieltä, joka on moderni avoimen lähdekoodin monialustainen olio-ohjelmointikieli (Microsoft 2024). C#-ohjelmointikieleen on integroitu samanaikaisuus, mikä tarkoittaa, että useita tehtäviä voidaan suorittaa rinnakkain tai yhtä aikaa, mikä on helpottaa kehittäjien työtä, kun halutaan tehdä tehokasta koodia (Sakovich 2024). C#-ohjelmointikielen vahvuuksia ovat myös automaattinen muistinhallinta, jolla se automaattisesti vapauttaa muistia silloin, kun ohjelmaan jää tarpeettomaksi käynyttä koodia.

Prasadin (2024) mukaan .NET, .NET Framework ja .NET Core aiheuttavatkin paljon hämmennystä, koska nimet sekoittuvat helposti keskenään ja koska jokainen nimi sisältää ison määrän teknologiaa ja tietoa.

4.2 .NET Core

.NET Core on monialustainen ja avoimen lähdekoodin ohjelmistokehityksen kehys, joka on erittäin yleinen kehitettäessä sovelluksia riippumatta niiden

alustasta. .NET Corea voidaan käyttää monenlaisiin sovelluksiin, palvelinsovellusten luomiseen tai suurempien sovelluskokonaisuuksien muuntamiseen mikropalveluiksi. (Stackify 2024.)

Esimerkkejä ohjelmistoista (Smith 2023), joita voidaan rakentaa .NET Corea käyttämällä:

- **Työpöytäsovellukset**: .NET Core tukee työpöytäsovellusten luomista ja Visual Studio Code on parhaimpia esimerkkejä .NET Corea käyttävistä työpöytäsovelluksista.

- **Konsolisovellukset**: .NET Core sopii hyvin kevyiden, yksinkertaisten konsolisovellusten rakentamiseen, jotka ovat hyödyllisiä automaatiotehtävissä tai käsittelytyökaluina.

- **IoT ja Sulautetut järjestelmät:** .NET Corea voidaan käyttää IoT-laitteilla ja sulautetuissa järjestelmissä, joissa keveys ja suorituskyky ovat tärkeitä erityisesti, koska .NET Core on alustariippumaton.

- **Web-sovellukset ja -palvelut:** .NET Corea käytetään ASP.NET Core -kehitysalustan kanssa web-sovellusten, API-rajapintojen ja web-palveluiden kehittämiseen. ASP.NET Core hyödyntää .NET Coren alustariippumattomuutta ja suorituskykyä.

- **Pelit:** C#-ohjelmointikieltä voidaan käyttää myös pelien rakentamiseen mobiililaitteille, työpöydälle, konsoleille, televisioille, sekä VR:lle (Virtual Reality).

Milloin kannattaa käyttää?

.NET Corea kannattaa käyttää silloin, kun halutaan tehdä alustasta riippumatonta koodia ja halutaan varmistua jatkuvasta tuesta. Jos sovelluksessa käytetään mikropalveluarkkitehtuuria, jossa suurempi sovelluskokonaisuus rakennetaan pienemmistä palveluista. .NET Core mahdollistaa eri tekniikoiden yhdistelmän ja skaalautuu mikropalveluille (Stackify 2024). Kun sovelluksen täytyy toimia useilla eri alustoilla, kuten Windows, Linux ja macOS.

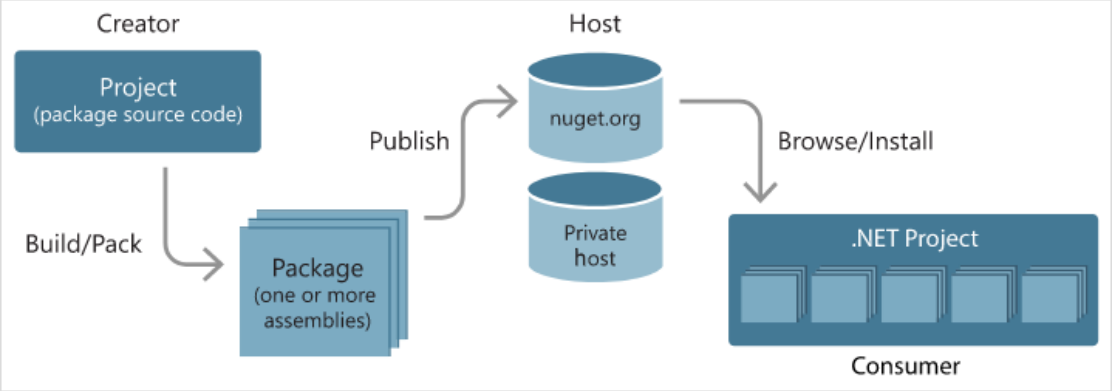
Milloin ei kannata käyttää

.NET Coressa ei ole kaikkia .NET-ominaisuuksia, eikä siten tukea kaikille kirjastoille ja laajennuksille. Jos yrityksen sovelluksissa käytetään paljon Windowsille tarkoitettua ominaisuuksia ja teknologioita, kuten Windows Forms. (Stackify 2024.)

4.3 NuGet-paketit

NuGet on paketinhallintaohjelma, joka on suunniteltu erityisesti .NET-kehystä varten ja sen avulla voidaan luoda, jakaa ja käyttää ohjelmapaketteja. Esimerkiksi järjestelmänä .NET Corea ei asenneta samalla tavalla kuin .NET Frameworkia, koska se koostuu useista NuGet-paketeista (Lukasiewicz 2021). NuGet on Microsoftin tukema koodinjakamismekanismi, ja se tarjoaa valmiita kirjastoja sekä työkaluja, joita kehittäjät voivat ladata ja hyödyntää projekteissaan (Microsoft 2022).

NuGetin avulla kehittäjät voivat nopeasti lisätä toiminnallisuuksia projektiinsa, koska NuGetin avulla koodit ovat kasattu paketteihin valmiiksi. Paketit sisältävät käännetyn koodin dynaamisena linkkikirjastona, jotka sisältävät koodia ja dataa, ja jota ohjelmat voivat jakaa ja käyttää Windows-ympäristössä. (Microsoft 2022.)



Kuva 4. Kuvaus NuGet-paketin luomisesta ja sen päätymisestä kuluttajalle (Microsoft 2022)

Kuvassa 4 havainnollistetaan NuGet-pakettien syntyminen, jossa tekijä luo NuGet-paketin ja sitten valitsee, julkaiseeko sen julkiseksi vai rajatulle käyttäjäkunnalle. Lopuksi julkaisun jälkeen kuluttaja voi ladata ja sisällyttää paketin projektiinsa, jonka jälkeen pakettien API:t ovat muun projektikoodin käytettävissä. (Microsoft 2022.)

4.4 EF Core – Entity Framework Core (ORM)

Entity Framework Core (EF Core) on Microsoftin kehittämä laajennettava, avoimen lähdekoodin ja monialustainen versio Entity Framework -tiedonkäyttötekniikasta. Tällä tekniikalla pystytään mahdollistamaan työskentely tietokantojen kanssa käyttäen .NET-objekteja. EF-Core-tekniikalla tietoihin päästään käsiksi mallin avulla, joka koostuu entiteettiluokista ja kontekstiobjekteista, jotka mahdollistavat tietojen kyselyn sekä tallentamisen. (Entity Framework Core 2024.)

Object Relational Mapper (ORM) on ohjelmistokehityksen tekniikka, jota käytetään helpottamaan tietokantojen käsittelyä. Tällä tekniikalla voidaan käyttää olio-ohjelmointia tietokannan kanssa ilman, että tarvitsee kirjoittaa SQL-kyselyitä. EF Core on yksi ORM-työkaluista ja sen avulla tietojen käyttö tapahtuu mallin avulla, joka koostuu entiteettiluokista ja kontekstiobjekteista (Entity Framework Core 2024).

Habr (2023) mukaan EF Core yksinkertaistaa tietokannan käyttöä siten, että kehittäjät voivat kirjoittaa tietokantakyselyitä SQL:n sijasta käyttämällä LINQ (Language-Integrated-Query) -syntaksia, jota on helppo ymmärtää ja lukea. EF Core tarjoaa myös mahdollisuuden käyttää ”Raw SQL” -tekniikkaa, joka viittaa SQL-kyselyihin ja jotka kirjoitetaan suoraan koodiin sen sijaan, että käytettäisiin ORM-kehystä kuten EF Core. Tämän tekniikan käyttö voi nopeuttaa kehitystä ja tuottavuutta huomattavasti, koska se sisältää paljon automaatiota, jonka avulla voidaan keskittyä rakentamaan sovelluslogiikkaa hitaampien menetelmien sijaan.

4.5 React & TypeScript

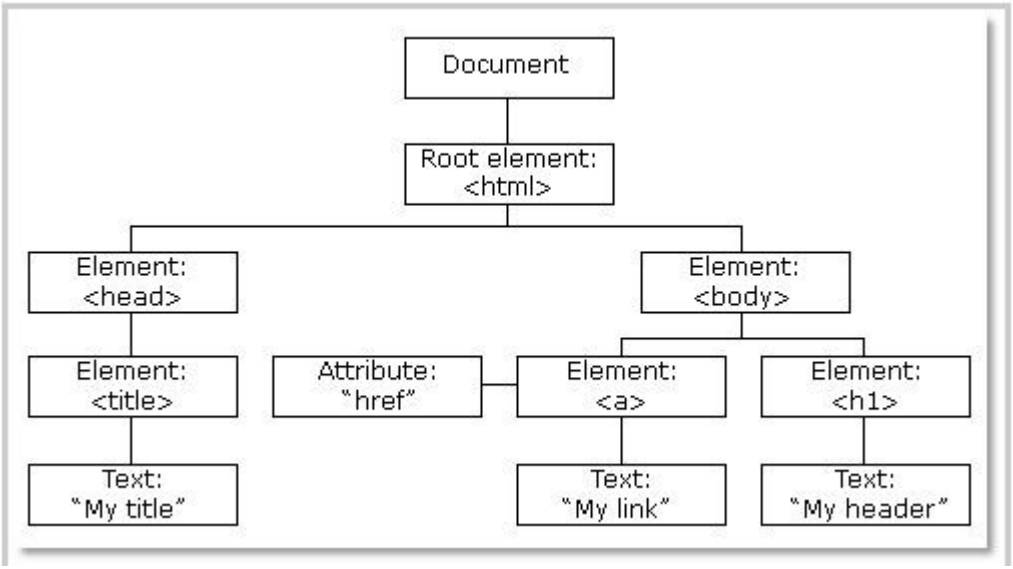
React (myös React.js tai ReactJS) on avoimen lähdekoodin JavaScript-kirjasto, jonka alun perin Facebook on luonut vuonna 2011 yksinkertaistamaan monimutkaisten käyttöliittymien rakentamisprosessia (Herbert 2023). Reactin avulla käyttöliittymiä rakennetaan luomalla komponentteja, jotka ovat yksittäisiä pieniä palasia ja sitten yhdistämällä ne kokonaisuudeksi, josta muodostuu sovelluksen käyttöliittymä.

Kun yleensä haetaan verkkosivua kirjoittamalla sen osoite selaimeen, silloin selain lähettää pyynnön palvelimelle, jolloin sivu latautuu. Jos klikataan linkkiä sivulla, uusi pyyntö lähetetään ja uusi sivu latautuu jälleen. Tämä toistuu aina, kun siirrytään sivuston eri osiin. Tämä tapa toimii kyllä, mutta sivustoilla, jotka sisältävät suuria määriä dataa, se voi hidastaa sivun toimintaa ja käyttökokemusta. (Herbert 2023.)

React toimii eri tavalla ja mahdollistaa siten yksisivuisen sovelluksen (SPA, single-page application), jossa vain ensimmäinen HTML-sivu ladataan. Kun käyttäjä klikkaa linkkiä sivulla, joka on toteutettu React-teknologialla niin, että vain tarvittavat osiot sivustolta päivitetään ilman sivun uudelleenlataamista. Tätä tapaa kutsutaan asiakaspuolen reitityksekseksi, mikä parantaa suorituskykyä huomattavasti. (Herbert 2023.)

Vaikka React ei annakaan tiukkoja sääntöjä koodikäytännöille tai tiedostojen järjestämiselle ja periaatteessa kehittäjät ja tiimit voisivat vapaasti määrittää sopivat käytännöt itselleen, niin monet kehittäjät noudattavat tiettyjä vakiintuneita konventioita. Tällä tarkoitetaan esim. kansiorakenteen järjestys komponenttien, palvelujen ja tyylien mukaan, sekä tietyt nimeämiskäytännöt, jotka parantavat koodin luettavuutta ja ylläpidettävyyttä. (Herbert 2023.).

React käyttää JavaScript-rakennetta, joka tunnetaan nimellä virtuaalinen DOM (Document Object Model). DOM on tietorakennemalli, jonka avulla selain käsittelee verkkosivua, ja jota selaimet ja JavaScript käyttävät verkkosivun sisällön ja ulkoasun muokkaamiseen. Käyttämällä virtuaalista DOM:ia ohjelmien ja sovellusten nopeus kasvaa huomattavasti, koska virtuaalinen DOM päivittää objektin muuttuessa ainoastaan kyseisen objektin tilan, eikä kaikkia mahdollisia objekteja. (Deshpande 2024.)



Kuva 5. Esimerkki verkkosivun DOM-rakenteesta (Simplilearn 2024)

DOM käsittelee XML- tai HTML-dokumentteja puurakenteena, jossa jokainen solmukohta on objekti, joka edustaa dokumentin osaa (kuva 4). Kun jonkun yksittäisen objektin tila muuttuu React-sovelluksessa, niin virtuaalinen DOM päivitetään. Tämän jälkeen se vertaa nykyistä tilaansa aiempaan tilaan ja päivittää vain ne objektit, joihin muutos on tehty. (Deshpande 2024.)

Komponentit ovat React-sovelluksien tärkeimpiä rakennuspalikoita, ja tyypillisessä React-sovelluksessa komponentteja on useita. Tarkemmin sanottuna komponentti on JavaScript-luokka tai funktio, joka hyväksyy ominaisuudet (props) ja palauttaa sitten React-elementin, joka kuvastaa käyttöliittymää. (Kagga 2018.)

Vaikka usein Reactin yhteydessä käytetään JavaScriptiä, voivat kehittäjät käyttää myös TypeScriptiä, joka on Microsoftin kehittämä korkean luokan ohjelmointikieli. Kun JavaScriptistä alkoi muuttua monimutkaisempaa ja haastavampaan, TypeScript toi ratkaisun siihen, koska sillä voidaan kirjoittaa selkeämpää ja paremmin ylläpidettävää koodia korjaamalla virheet jo kehitysvaiheessa. TypeScriptin tärkein hyöty on muuttaa JavaScript -ohjelmointikieli vahvasti tyypitetyksi kieleksi ja sitä suositellaan käytettäväksi JavaScriptin sijasta, jos TypeScript on kehitystiimille tutumpi. (Emizentech 2024.)

TypeScript -ohjelmointikielen hyötyjä:

- parempi koodin laatu
- ominaisuudet kuten IntelliSense

- parempi koodin luettavuus ja ymmärrettävyys
- skaalautuvuus, pystyy hallitsemaan monimutkaisia koodikantoja
- ylläpidettävyys ja muutettavuus (Emizentech 2024).

Yhteenvetona voidaan todeta, että React ja TypeScript yhdessä tarjoavat tukevuutta ja skaalautuvuutta verkkokehitykseen. Tämä yhdistelmä on erityisen hyödyllinen suurikokoisissa sovelluksissa, joissa tietorakenteet ovat monimutkaisia ja haastavia (Sharma 2024). TypeScript siis parantaa React-kehitystä tehden siitä tehokkaampaa ja virheettömämpää, kun otetaan huomioon koodin laadun parannus, virheenkorjaus ja skaalautuvampi arkkitehtuuri.

5 SOVELLUKSEN VERSIONHALLINTA

Versionhallinnalla tarkoitetaan palvelua ja ohjelmistokehityksen käytäntöä, jonka avulla kehittäjät voivat hallita koodin muutoksia, sekä tallentaa koodia keskitettyyn arkistoon (Versionhallinta ja Git s.a.). Tällä pyritään siis säilyttämään varmuuskopioita sekä nykyisestä, että aiemmista versioista. Näin pystytään tekemään helposti yhteistyötä muiden tiimiläisten tai ulkopuolisten kanssa, sekä ylläpitämään koodipohjaan tehtyjen muutosten historiaa.

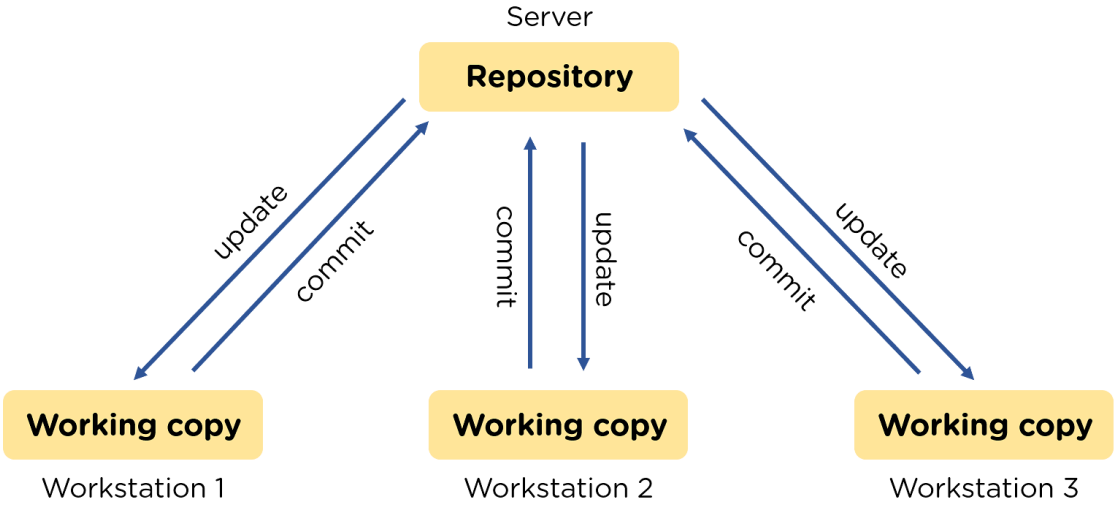
Versionhallintajärjestelmien avulla voidaan luoda haaroja tai oksia, jotka ovat erillisiä kopioita uusimmasta tai tietystä koodipohjan versiosta ja joita voidaan muokata itsenäisesti. Tällä tavalla jokainen kehittäjä voi työskennellä saman projektin äärellä uusien eri ominaisuuksien parissa samaan aikaan tai korjata virheitä häiritsemättä toisia tiimiläisiä (Geeksforgeeks 2022.) Jos, ja kun virheitä sattuu, voidaan palata aina takaisin koodin edelliseen versioon ja palauttaa tehdyt muutokset. Tämä säästää valtavasti aikaa ja vaivaa verrattuna siihen, että virheelliset koodimuutokset kumottaisiin manuaalisesti. Versionhallintajärjestelmät tarjoavat myös historiatiedot koodipohjaan tehdyistä muutoksista, joten voidaan analysoida koodin kehitystä ja etsiä mahdollisia tehtyjä virheitä kauempaakin (Geeksforgeeks 2022).

Käyttämällä versionhallintaa parannetaan huomattavasti projektin kehitysnopeutta tarjoamalla mahdollisuus laajaan yhteistyöhön ja tiimistyöskentelyn helpottumiseen. Virheiden ja ristiriitojen mahdollisuus vähenee, kun ominaisuuksia lisätään haara kerrallaan, ja koska jokaisesta muutoksesta jää jälki, jota voidaan tarkastella ja muokata myös myöhemmin (Geeksforgeeks 2022). Näin

saadaan myös tieto siitä kuka, mitä, milloin ja miksi muutoksia koodipohjaan on tehty.

Versionhallintajärjestelmän ominaisuuksia (GitHub 2022):

- **Repository:** Keskitetty tietokanta, joka tallentaa koodipohjan tiedostojen ja kansioiden kokoelman, sekä versiohistorian.
- **Pull Request:** Tekniikka, jolla kehittäjät luovat ehdotuksen, merkitsevät, kommentoivat ja keskustelevat niistä ennen haaran yhdistämistä pääkoodipohjaan.
- **Commit:** Tätä komentoa käytetään muutosten tallentamiseen paikalliseen arkistoon. Muutokset voidaan tehdä myös suoraan ohittamalla ”staging”- vaihe, jossa muutokset siirrettäisiin valmistelevaan vaiheeseen.
- **Branch:** Eli haara, tarkoittaa erillistä itsenäistä versiota tai kopiota koodikannasta (yleensä uusimmasta versiosta), jolloin kehittäjät voivat tehdä ominaisuuksia itsenäisesti omaan haaraan ilman, että se muuttaa pääkoodikantaa.
- **Merge:** Kun kehittäjä haluaa yhdistää tekemänsä muutokset haarastaan pääkoodikantaan, se voidaan integroida ”Merge” -komennolla.
- **Conflict:** Voi syntyä konflikti, jos useampi kehittäjä tekee samanaikaisesti sellaisia muutoksia koodiin, jotka ovat ristiriidassa toistensa kanssa. Versionhallintatyökalu auttaa ratkaisemaan konfliktit helposti.
- **Checkout:** Kun kehittäjä hakee tiedoston versionhallintajärjestelmästä ja avaa sen kehitysympäristössä, sitä kutsutaan nimellä ”checkout”.
- **Tag:** Merkintä, jota tiimiläiset voivat käyttää merkitsemään tiettyjä kohtia lähdekoodihistoriassa, kuten merkityn version julkaisupäivän. Merkintä on kuin haara, joka ei muutu.
- **Remote:** Etäkehityksen avulla voidaan tehdä osa tai kaikki työstä paikallisella työpöydällä, yrityksen palvelimella tai pilven kautta.
- **Fork:** Tällä tekniikalla voidaan luoda olemassa olevasta lähdekoodista tai arkistosta kopio toiseen arkistoon.
- **Revert:** Kehittäjät voivat palauttaa ja kumota yhden tai useamman tekemänsä muutoksen ja palata aiempaan versioon.



Kuva 6. Havainnollistava kuva versionhallinnan toiminnasta (Simplilearn 2023)

5.1 GIT

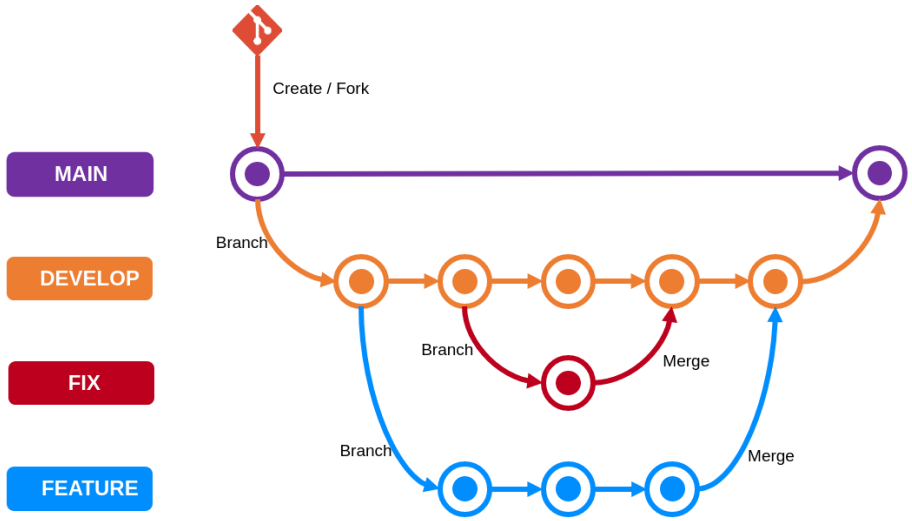
Git on vuonna 2005 kehitetty yleisimmin käytetty versionhallintajärjestelmä, jota käytetään korkeankin luokan projekteissa. Git seuraa tiedostoihin tekemiäsi muutoksia, joten kaikista muutoksista löytyy historia, joihin voi myös palata tarvittaessa myöhemmin. Git on ohjelmisto, joka asennetaan ja sitä ylläpidetään paikallisesti, eikä pilvessä, joten ei tarvita pilvipalveluita eikä edes internetiä sen käyttämiseen. Tämä tarkoittaa sitä, että tiedostosi ja niiden historia tallennetaan paikallisesti tietokoneellesi keskitettyyn paikkaan. Gitin haaroittamisen avulla voidaan luoda itsenäisiä haaroja pohjakoodista ja kokeilla uusia ideoita toiseen haaraan, mutta aina voidaan kuitenkin palata takaisin aikaisempiin haaroihin. (Nobledesktop 2023.) Git tarjoaakin monipuoliset mahdollisuudet ja työkalut tiimityöskentelyyn myös etänä, koska kehittäjät voivat nähdä ja muokata toisten kirjoittamaa koodia, nähdä projektin koodiin tehtyjen muutosten historian ja antaa myös palautetta toisille.

Tiedostolla on gitissä kolme tilaa: **committed**, **modified** ja **staged**. Tietoa tallennetaan Git-projektiin committeina, ja kun tiedoston tila on committed, se tarkoittaa, että tiedosto on tallennettu lokaalisti sellaisena kuin se sillä hetkellä on. **Commit** on kuin paketti projektin tiedostoihin tehtyjä muutoksia, ja muutokset tarkoittavat esimerkiksi sitä, että tekstiä lisättiin tai poistettiin jostain projektin tiedostosta. **Modified** tiedosto tarkoittaa, että tiedostoon on tehty joitain

muutoksia, mutta niitä ei ole vielä tallennettu lokaaliin git-kantaan. **Staged** tarkoittaa, että muokattu (modified) -tiedosto on merkitty tallennettavaksi seuraavassa commitissa lokaaliin kantaan. (Versionhallinta ja Git s.a.)

Git komentoja (Sirotka 2022):

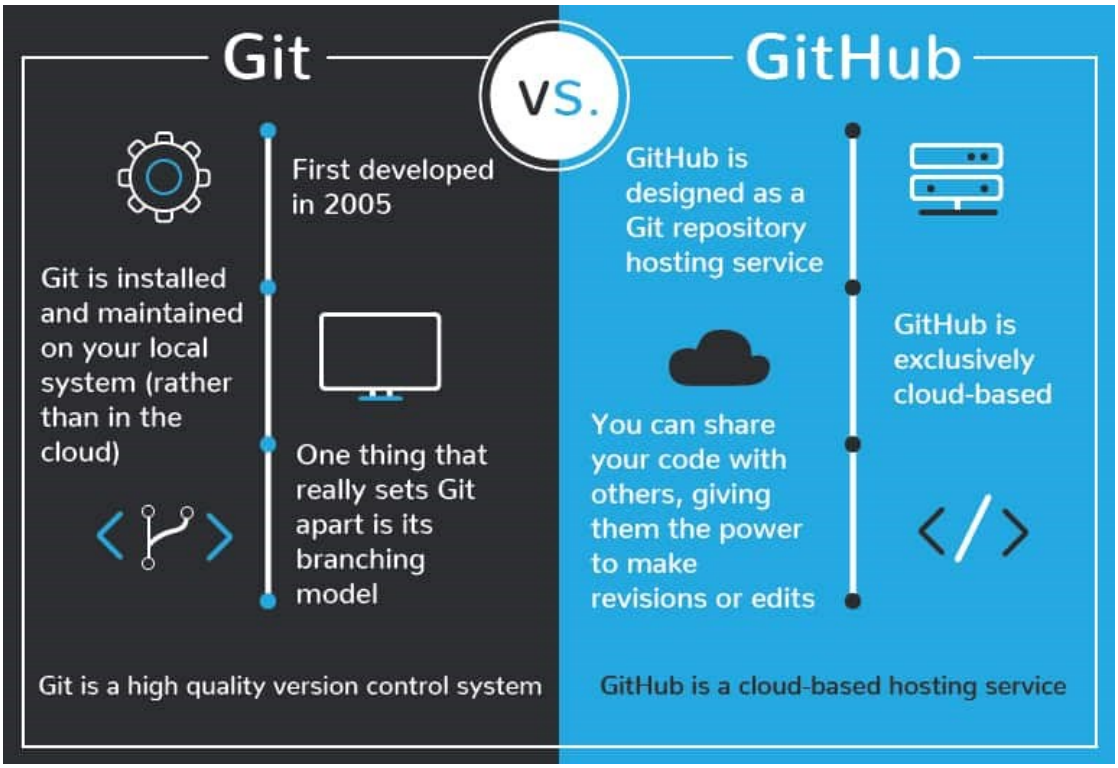
- **Git Init:** Tällä komennolla voidaan luoda kansiosta uusi Git-projekti, jonka jälkeen voidaan suorittaa muita Git-komentoja.
- **Git Clone:** Tämän komennon avulla voidaan tehdä lokaali kopio olemassa olevasta projektista, joka sijaitsee GitHubissa.
- **Git Add:** Valmistelee muutoksen, eli lisää kaikki tiedostoon tehdyt muutokset staged vaiheeseen.
- **Git Merge:** Yhdistää kaksi haaraa, jotka eivät ole ristiriidassa keskenään.
- **Git Pull:** Tällä komennolla voidaan hakea samassa projektissa olevan toisen henkilön tekemät koodimuutokset GitHubista itselle.
- **Git Push:** Käytetään kun halutaan ladata paikallisesti tehdyt koodimuutokset etäarkistoon (Repository).



Kuva 7. Git prosessi kuvattuna visuaalisesti (Git Process s.a.)

5.2 GitHub

GitHub on pilvipohjainen lähdekoodin versionhallintajärjestelmä palvelu, joka perustuu Git-versionhallintajärjestelmään ja tarjoaa käyttöliittymän Gitin toimintojen hallitsemiseen. GitHubissa voidaan säilyttää ja julkaista Git-projekteja. GitHubiin voidaan myös luoda projekteista etätietovarastoja (”remote repositories”). Näiden etävarastojen avulla voidaan tehdä yhteistyötä monien kehittäjien kanssa saman projektin parissa, kun kehittäjät voivat ottaa omia kopioitaan tästä GitHubissa olevasta tietovarastosta ja myöhemmin liittää omaan versioon tehdyt muutoksensa sinne takaisin. Haarukat (Forks) ovat tärkeitä avoimen lähdekoodin kehitykselle. (Heller 2024.) Fork tarkoittaa sitä, että kuka tahansa voi ottaa kopionsa keskitetystä tietovarastosta (Repository), tehdä muutoksia ja pyytää niiden liittämistä takaisin pääprojektiin tekemällä pyynnön (Pull request). Muita GitHubia vastaavia palveluita ovat mm. GitLab, Bitbucket ja Azure DevOps.



Kuva 8. Havainnollistava kuva Gitin ja GitHubin eroista (Devmountain s.a.)

Gitillä ja GitHubilla on paljon yhtäläisyyksiä ja ne auttavat koodin hallinnassa, mutta niillä on myös erilaisia tarkoituksia sekä ominaisuuksia. Alla on muutamia yhtäläisyyksiä ja eroja GitHubista sekä Gitistä:

Yhtäläisyydet:

- Git ja GitHub molemmat mahdollistavat koodin versionhallinnan sekä jakamisen, joka edistää yhteistyötä ohjelmistokehityksessä.

- Molemmilla on laaja käyttäjäkunta ja aktiivinen kehittäjäyhteisö, sekä ne tarjoavat ilmaisia ratkaisuja avoimen lähdekoodin projekteille, mutta myös maksullisia vaihtoehtoja.

Erot:

- Git on hajautettu versionhallintaohjelmisto, kun taas GitHub on verkko-pohjainen Git-repositorion isännöintipalvelu.

- GitHub on keskitetty palvelu, ja Microsoftin omistama, kun taas Git on täysin riippumaton.

- Gitiä voi käyttää ilman GitHubia, mutta GitHub vaatii Gitin toimiakseen. (Diachenko 2023.)

6 CLEAN ARCHITECTURE – MALLI

Clean Architecture on ohjelmistokehityksen arkkitehtuurimalli, jonka pohja perustuu muihin arkkitehtuurimalleihin, kuten Hexagonal- ja Onion- arkkitehtuurimalleihin. Sen idea on korostaa vastuiden erottelua ja ohjelmakoodin ylläpidettävyyttä. Clean Architecture -mallin esitteli Robert C. Martin vuonna 2012. (Anwar 2023.)

Clean Architecturen tavoitteena on luoda arkkitehtuurimalli, joka mahdollistaa muutoksien ja päivitysten tekemisen sovellukseen ilman, että se vaikuttaisi koko järjestelmäkokonaisuuteen. Se sisältää useita eri komponentteja, sekä neljä kerrosta: esityskerroksen, sovelluskerroksen, domain-kerroksen ja infrastruktuurikerroksen. (Patel 2023.)

6.1 Clean Architecturen periaatteet

Monissa tapauksissa perinteisellä arkkitehtuurimallilla rakennetut sovellukset sitovat sovelluksen ydinlogiikan ulkoisiin riippuvuuksiin, kuten kehyksiin, tietokantoihin ja käyttöliittymään. Tämänlainen riippuvuus muista sovelluksen osista tekee muutosten ja uusien ominaisuuksien lisäämisestä haastavaa. Myös komponenttien poistaminen on haastavaa ilman, että se vaikuttaisi koko järjestelmään. Tämänkaltaisessa arkkitehtuurissa testaaminen on myös haastavaa, koska ydinliikelogiikan ja muiden komponenttien välisen riippuvuuden

puuttuminen tekee siitä vaikeaa, joka voi johtaa virhealttiiseen koodiin. Clean Architecturen avulla voidaan ratkaista tämänlaiset ongelmat seuraavien periaatteiden avulla. (Singh 2023.)

Huolenaiheiden erottaminen

Clean Architecture malli erottaa sovelluksen ydinliiketoiminnan logiikan sovelluksen ulkoisista osista ja komponenteista. Sovelluksen ydinliiketoiminnan logiikan ja ulkoisten komponenttien erottaminen saavutetaan erilaisten kerrosten avulla. Jokaisella kerroksella on oma tehtävänsä ja vastuunsa sovelluksessa. (Singh 2023.)

Erottelu mahdollistaa helpomman ylläpidon, testauksen ja päivityksien tekemisen, koska jokainen huolenaihe voidaan käsitellä omana kokonaisuutenaan vaikuttamatta muihin sovelluksen tai järjestelmän osiin. Se myös edistää modulaarista ja joustavaa suunnittelua. (Kozon 2023.)

Riippuvuussääntö

Clean Architecturessa riippuvuudet suuntavat ydinliiketoimintalogiikkaan. Tämä tarkoittaa sitä, että riippuvuutta ei synny korkean- ja matalan tason moduuleille. Tällä periaatteella varmistetaan se, että ulkoisiin komponentteihin tehdyt muutokset eivät vaikuta ydinliiketoiminnan logiikkaan. (Singh 2023.)

Kozonin (2023) mukaan riippuvuussääntöä noudattamalla koodista saa modulaarisempaa ja uudelleenkäytettävää ja se edistää huolenpidon erottamista ja tarjoaa paremmat mahdollisuudet koodin testaukselle. Riippuvuussääntöä vahvistetaan yleensä käyttämällä riippuvuuden kääntötekniikoita, kuten ohjausinversiota (IoC) ja riippuvuuden injektointia (DI). Kyseiset tekniikat varmistavat sen, että riippuvuuksia voidaan hallita ulkoisesti ja, että niitä voidaan helposti vaihtaa tai ”simuloida” testauksen aikana.

Testattavuus

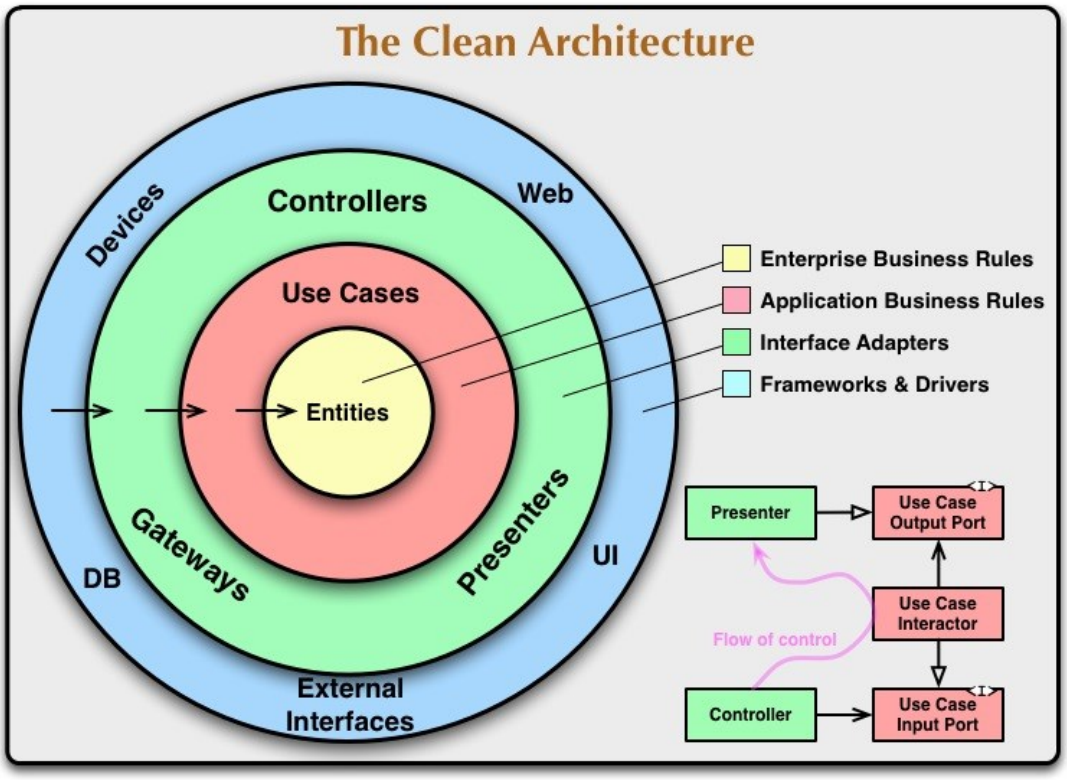
Clean Architecture malli mahdollistaa liiketoimintalogiikkaan kohdistuvien yksikkötestien kehittämisen ulkoisista riippuvuuksista huolimatta. Erottelu mahdollistaa sen, että voidaan kehittää tehokkaita ja kattavia ydinliiketoiminnan

testejä ilman, että jokaista muutosta varten täytyy luoda uusia integraatiotestejä. (Singh 2023.)

Alustan riippumattomuus

Liiketoimintalogiikka on riippumaton sovelluksessa käytetystä alustasta tai viitekehyksestä. Se helpottaa sovelluskokonaisuuden teknologioiden vaihtamista tai sovelluksen mukauttamista eri alustaan ilman, että ydinlogiikkaa tarvitsee kirjoittaa uudelleen. (Singh 2023.)

6.2 Arkkitehtuurikerrokset ja niiden roolit



Kuva 9. Clean Architecture kaavio (Martin 2012)

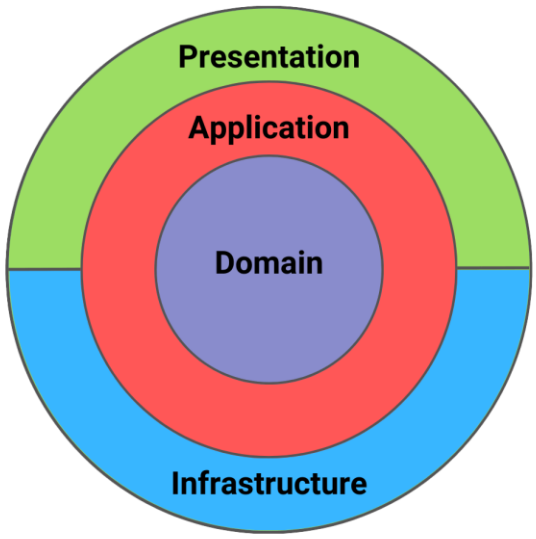
Clean Architecture malli perustuu siihen, että entiteetit (Entities) ovat sovelluksen ensisijaisia peruskomponentteja ja ne edustavat liiketoimintaobjekteja, joita käytetään käyttötapauksien suorittamiseen. Entiteetit määräävät ydinliiketoiminnan säännöt, eikä niihin vaikuta esimerkiksi käyttöliittymä tai muu ulkoinen tekijä. Käytännössä entiteetit muodostavat sovelluksen päälogiikan ja idean. (Bitloops s.a.)

Käyttötapaukset (Use Cases) ovat Clean Architecture mallin seuraava kerros

ja ne edustavat ohjelman liiketoimintasääntöjä, jotka ovat spesifisiä sovellukselle, mutta ei sovelluksen hallinnoivalle yritykselle tai yksilölle. Entiteettejä käytetään käyttötapausten luomiseen ja jokainen käyttötapaus tarkoittaa toimintoa, jonka käyttäjä suorittaa esimerkiksi käyttöliittymän avulla. Käyttötapaukset edustavat siis erilaisia tapoja, joilla sovellusta voi käyttää. (Bitloops s.a.)

Rajapinta-adapterit (Interface-adapters) ovat seuraava kerros tässä mallissa. Niiden tehtävä on käsitellä sovelluksessa käytettävää dataa. Tähän kerrokseen kuuluu esimerkiksi ohjaimet (Controllers), jotka käsittelevät käyttäjän syötteitä. Portit (Gateways) vastaavat ulkoisten palveluiden, kuten tietokantojen tai verkkopalveluiden käytöstä. Esittäjät (Presenters) vastaavat datasta, joka näytetään käyttöliittymässä. (Bitloops s.a.)

Ulkoiset rajapinnat edustavat järjestelmän rajapintoja, jotka sijaitsevat pääjärjestelmän ulkopuolella. Nämä rajapinnat voivat olla esimerkiksi käyttöliittymiä tai tietokantoja. Ulkoiset rajapinnat ovat vastuussa datan kääntämisestä sovelluksen ja ulkoisten järjestelmien välillä. (Bitloops s.a.)



Kuva 10. Clean Architecturen malli, joka kuvastaa sovelluksen projektirakennetta. Jokainen kerros vastaa yhtä projektia järjestelmäratkaisun sisällä (Jovanovic 2022)

6.3 Hyödyt ohjelmistokehityksessä

Clean Architecture mahdollistaa useita etuja, koska se tukee ylläpidettävien, skaalautuvien ja mukautettavien ohjelmistojärjestelmien luomista. Näiden etujen avulla se on tehokas ja hyödyllinen lähestymistapa ohjelmistokehitykselle. Clean Architecturen pääasiallisiin hyötyihin kuuluu: (Bitloops s.a.)

Kehysten ja työkalujen riippumattomuus, koska se tarjoaa ohjelmistokehittäjille mahdollisuuden muuttaa yksitäisen komponentin muuttamista ilman, että sillä on vaikutusta muuhun järjestelmään. Riippumattomuus kehysten ja työkalujen välillä mahdollistaa mukautuvamman ohjelmistokehityksen silloin, kun liiketoiminnan vaatimuksiin tulee muutoksia, tai teknologioita vaihdetaan.

Joustavuus ja skaalautuvuus on yksi Clean Architecturen hyödyistä. Sen kerrostettu arkkitehtuuri antaa järjestelmäkokonaisuudelle selkeät rajat, joka mahdollistaa helpon järjestelmä muokkauksen tai uusien ominaisuuksien lisäämisen.

Testattavuus on Clean Architecturen mukaisen arkkitehtuurimallin sovelluksessa huomattavista helpompaa, koska toimialueen logiikka on erotettu infrastruktuurista. Kerrosten avulla on helppoa kirjoittaa yksikkötestejä jokaiselle kerrokselle ja niiden logiikalle.

Clean Architecture tarjoaa rakenteen, joka helpottaa järjestelmän **ylläpitoa ja muokkaamista**. Riippuvuuksien vähäisyys ja huolenaiheiden erottelun avulla virheiden tunnistaminen, korjaus ja uusien ominaisuuksien lisääminen on helppoa. Tämä malli myös tukee ketterää kehitystä ja jatkuvaa toimitusta, jota käytetään yleensä projekteissa, jonka liiketoiminnan vaatimukset muuttuvat säännöllisesti.

7 SOVELLUKSEN ARKKITEHTUURIN KERROKSET JA NIIDEN TOIMINTA .NET YMPÄRISTÖSSÄ

7.1 Domain Layer (Domain-kerros)

Domain-kerros on koko järjestelmän ydin, joka koostuu ydintoimintojen entiteeteistä. Domain-kerros ei ole millään tavalla riippuvainen muista kerroksista, vaan päinvastoin jokainen muu kerros sovelluksessa on riippuvainen Domain-kerroksesta. (Muhammad 2024.)

```
// Domain Layer
public class Order
{
    public int Id { get; set; }
    public string CustomerName { get; set; }
    public List<OrderItem> Items { get; set; }

    public decimal GetTotal() => Items.Sum(i => i.Price * i.Quantity);
}
```

Kuva 11. Kuvakaappaus Order entiteetistä (Staael 2024)

Tässä esimerkissä luokka ”Order” edustaa liiketoimintaentiteettiä. Se myös sisältää liiketoimintalogiikkaa, jossa lasketaan kokonaishinta tuotteille. (Staael 2024.)

7.2 Application Layer (Sovelluskerros)

Sovelluskerroksessa tapahtuu kaikki sovelluksen liiketoimintalogiikka. Sovelluskerros hallitsee tiedonvirtausta liiketoimintaentiteettien välillä. Tämä kerros yleensä sisältää palvelut (Services), komennot (Commands), kyselyt (Queries), poikkeukset (Exceptions) ja logit (Logs). (Muhammad 2024.)

```
// Application Layer (Use Case)
public class CreateOrderUseCase
{
    private readonly IOrderRepository _orderRepository;

    public CreateOrderUseCase(IOrderRepository orderRepository)
    {
        _orderRepository = orderRepository;
    }

    public async Task ExecuteAsync(Order order)
    {
        // Business rules validation
        if (order.Items.Count == 0)
            throw new InvalidOperationException("Order must have at least one item.");

        // Save order
        await _orderRepository.SaveAsync(order);
    }
}
```

Kuva 12. Kuvakaappaus sovelluskerroksen logiikasta (Staael 2024)

Tässä esimerkki sovelluskerroksen logiikasta, jonka avulla luodaan tilaus hyödyntäen ”Order” entiteettiä. Logiikka kommunikoi ulkoisen ”IOrderRepository” riippuvuuden kanssa, mutta tämä osa logiikasta ei ole tietoinen, miten ja mihin tilaus tallennetaan. (Staael 2024.)

7.3 Infrastructure Layer (Infrastruktuurikerros)

Infrastruktuurikerros pitää sisällään kaikki ulkoiset palvelut ja tietokannat. Jokainen ulkoinen palvelu, esimerkiksi sähköpostipalvelut, tallennusratkaisut,

viestijonot ja 3 osapuolen API-kutsut käsitellään infrastruktuurikerroksessa. (Muhammad 2024.)

```
// Infrastructure Layer (Repository Implementation)
public class SqlOrderRepository : IOrderRepository
{
    private readonly AppDbContext _dbContext;

    public SqlOrderRepository(AppDbContext dbContext)
    {
        _dbContext = dbContext;
    }

    public async Task SaveAsync(Order order)
    {
        _dbContext.Orders.Add(order);
        await _dbContext.SaveChangesAsync();
    }
}
```

Kuva 13. Kuvakaappaus infrastruktuurikerroksen logiikasta (Staael 2024)

Tässä esimerkki “SqlOrderRepository” logiikasta, joka sijaitsee infrastruktuurikerroksessa. Sen tehtävä on tallentaa “Order” entiteetti SQL tietokantaan käyttäen EF Corea. Se toteuttaa IOrderRepository rajapinnan, joka on asetettu Domain-kerrokseen. Näin varmistetaan se, että sovelluksen ydinlogiikka ei ole riippuvainen tietokantateknologioista. (Staael 2024.)

7.4 Presentation Layer (Esityskerros)

Esityskerros vastaa datan esittämisestä sovelluksen käyttäjälle helposti ymmärrettävässä muodossa. Tämä kerros on yleensä toteutettu verkkoprojektina tai API-projektina. API-projekti koostuu ohjaimista (Controllers), jotka määrittelevät toimintamenetelmiä tai API-päätepisteitä (Endpoints). (Muhammad 2024.)

```
// Interface Adapters Layer
[ApiController]
[Route("api/[controller]")]
public class OrderController : ControllerBase
{
    private readonly CreateOrderUseCase _createOrderUseCase;

    public OrderController(CreateOrderUseCase createOrderUseCase)
    {
        _createOrderUseCase = createOrderUseCase;
    }

    [HttpPost]
    public async Task<IActionResult> Create([FromBody] OrderDto orderDto)
    {
        var order = new Order
        {
            CustomerName = orderDto.CustomerName,
            Items = orderDto.Items.Select(i => new OrderItem
            {
                ProductId = i.ProductId,
                Quantity = i.Quantity,
                Price = i.Price
            }).ToList()
        };

        await _createOrderUseCase.ExecuteAsync(order);
        return Ok();
    }
}
```

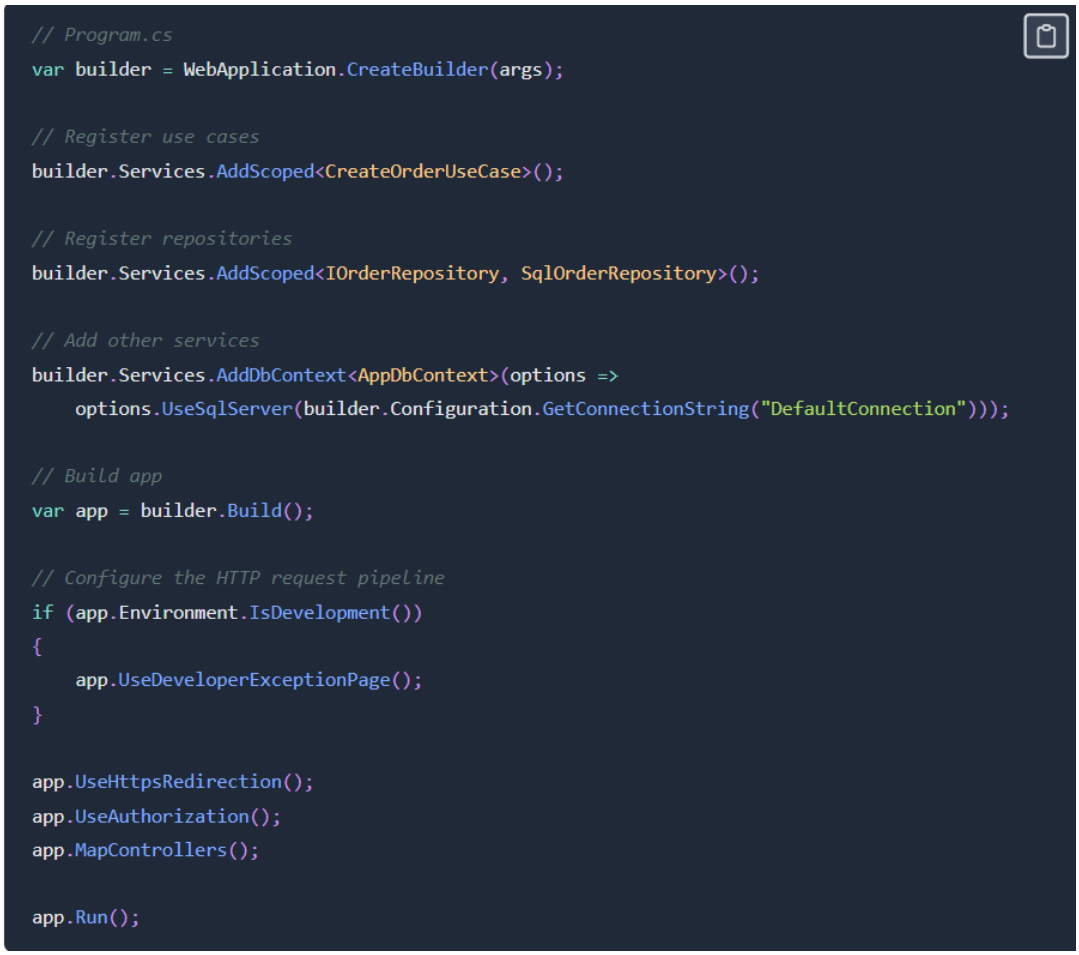
Kuva 14. Kuvakaappaus esityskerroksen logiikasta (Staael 2024)

Tässä esimerkki “OrderController” ohjaimesta. Se mahdollistaa API-pyynnön, jota sovelluskerros voi käyttää “Order” entiteetin luomiseen. Ohjain (Controller) käsittelee HTTP-pyyntöjä ja prosessoi syötetyt tiedot DTO (Data transfer object) muodossa, jonka jälkeen se välittää ne sovelluskerroksen logiikalle. Tämä kerros on myös vastuussa vastauksen lähettämisestä takaisin käyttäjälle. (Staael 2024.)

7.5 Riippuvuuden inversioperiaate (Dependency Inversion Principle, DIP)

DIP-periaate varmistaa sen, että ylemmän tason kerrokset kuten sovelluskerros ei ole riippuvainen muista alemman tason kerroksista ja moduuleista, kuten tietokannasta- tai API-toteutuksista. DIP mahdollistaa sen, että kaikki ovat riippuvaisia abstraktioista. (Staael 2024.)

Kuvan 12 esimerkissä ”CreateOrderUseCase” on riippuvainen ”IOrderRepository” rajapinnasta, mutta ei mistään konkreettisesta toteutuksesta. Tämän toteutuksen avulla on mahdollista vaihtaa tietokantatoteutus toiseen ilman, että se vaikuttaisi sovelluksen ydintoiminnallisuuteen. (Staael 2024.)
Riippuvuuksien injektointia (Depency Injection, DI) käytetään riippuvuuksien injektointia ohjaimiin (Controllers), käyttötapauksiin (Use Cases), ja repositorioihin (Reporitories). (Staael 2024.)



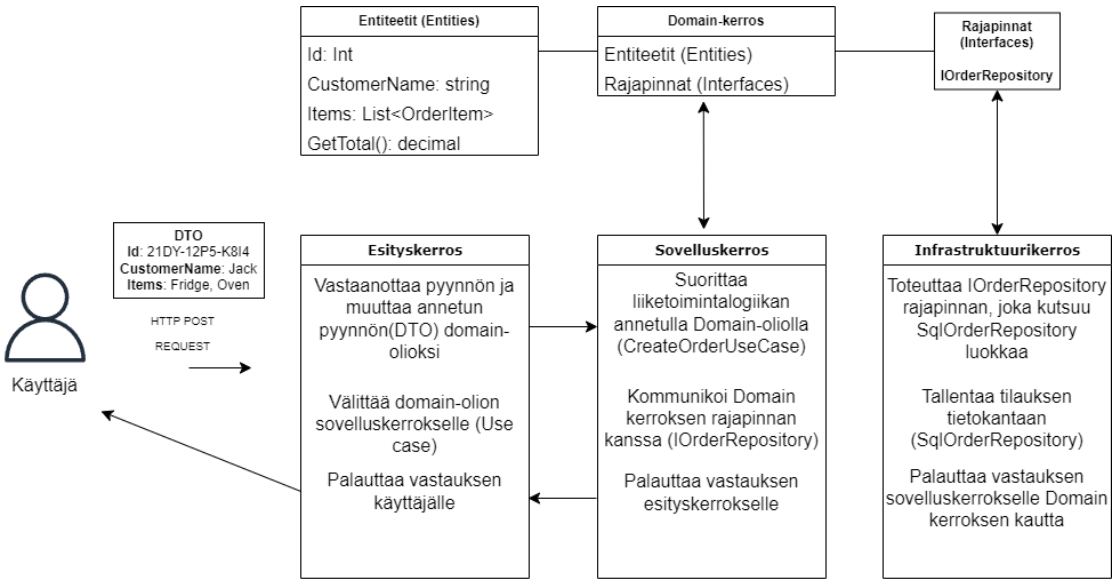
```
// Program.cs
var builder = WebApplication.CreateBuilder(args);

// Register use cases
builder.Services.AddScoped<CreateOrderUseCase>();

// Register repositories
builder.Services.AddScoped<IOrderRepository, SqlOrderRepository>();

// Add other services
builder.Services.AddDbContext<AppDbContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

// Build app
var app = builder.Build();

// Configure the HTTP request pipeline
if (app.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}

app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers();

app.Run();
```

Kuva 15. Kuvakaappaus .NET program.cs tiedostosta, johon on injektoitu käyttötapaukset ja repositoriot (Staael 2024)

Kuvan 15 ohjelmakoodi rekisteröi ”CreateOrderUseCase” ja ”SqlOrderRepository” luokkien toteutukset DI-konttiin. Se myös rekisteröi Entity Framework Core:n DbContext-luokan. (Staael 2024.)

7.6 Arkkitehtuurikerrosten tapahtumien kulku

Alla esitetyssä kuvassa on esimerkki siitä, miten käyttäjän syötteet ja data käsitellään Clean Architecture mallin sovelluksessa. Kaaviossa on kuvattuna Clean Architecture mallin kerrokset ja niiden tehtäviä. Kaavion pohjana on käytetty Martin Staeelin tekstin pohjalta laadittua esitystä Clean Architecture mallista.



Kuva 16. Esimerkki tapahtumien kulusta Clean Architecture mallissa

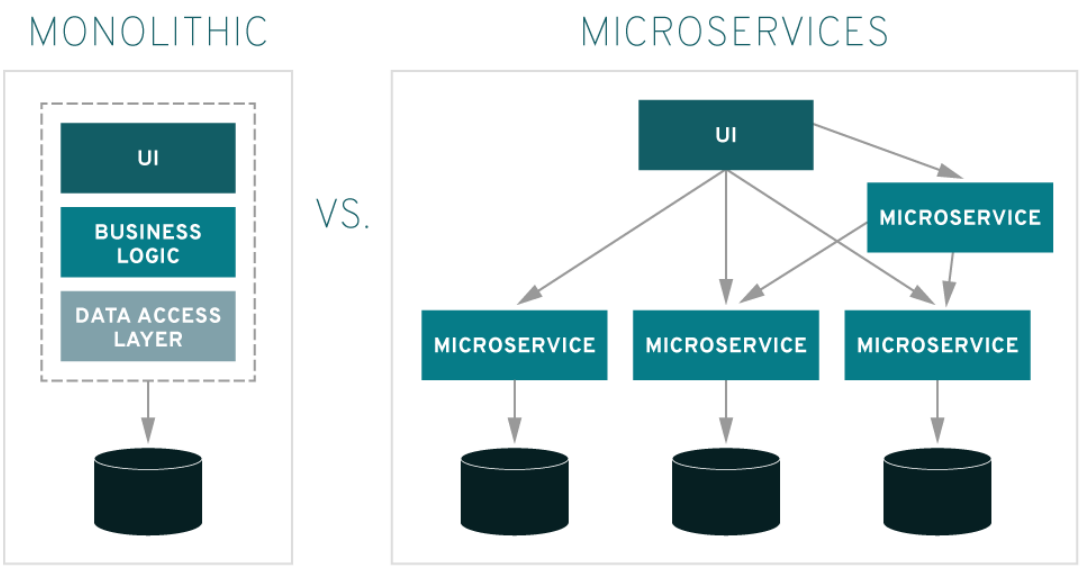
Ensin käyttäjä lähettää HTTP POST pyynnön palvelimelle, jonka ottaa vastaan esityskerroksen ohjain. Esityskerroksessa saatu data (DTO) muutetaan domain-olioksi. Tämä olio välitetään sovelluskerrokselle, jossa käytetään vastaanotettua dataa liiketoimintalogiikan suorittamiseen. Sovelluskerroksessa kutsutaan rajapintaa, jonka avulla voidaan ohjata saatu data infrastruktuurikerrokselle, joka on vastuussa tietojen tallentamisesta tietokantaan. Tämän jälkeen tulokset palautetaan esityskerrokseen ja sen jälkeen käyttäjälle. Esitys, sovellus ja- infrastruktuurikerroksen eriytys parantaa järjestelmän testattavuutta ja ylläpidettävyyttä, koska jokainen kerros on erillään toisistaan. (Staael 2024.)

8 MIKROPALVELUARKKITEHTUURI

Mikropalvelut viittaavat sovellusarkkitehtuurin lähestymistapaan, jossa suurempi sovellus rakennetaan käyttämällä itsenäisiä palveluita ja komponentteja (Red Hat 2023). Nämä itsenäiset palvelut kommunikoivat keskenään kevyiden rajapintojen kautta ja jokainen mikropalvelu keskittyy yhteen toiminnallisuuteen ja toimii siten erillään muista palveluista.

Mikropalveluarkkitehtuuri keskittyy pääasiassa suurten ja monimutkaisten sovellusten hajauttamiseen pienemmiksi palasiksi, joka helpottaa niiden ylläpitoa, päivityksiä ja skaalautuvuutta (Choudhary 2024). Jokainen mikropalvelu käsittelee sovelluksen tiettyä toimintoa ja osa-aluetta, joten se mahdollistaa sulavamman kehityksen, koska tiimit voivat työskennellä eri mikropalveluiden parissa samaan aikaan. Mikropalvelut mahdollistavat myös eri teknologioiden ja ohjelmointikielin käyttämisen samaan aikaan eri palveluissa.

Mikropalveluarkkitehtuuri on vastaus monoliittisen arkkitehtuurin haasteisiin, koska monoliittiset sovellukset ovat erittäin rajoitettuja ja epävakaita. Tässä lähestymistavassa jokainen mikropalvelu voidaan ottaa itsenäisesti käyttöön, sekä ne voivat kommunikoida keskenään. (Choudhary 2024.)



Kuva 17. Monoliittiarkkitehtuurin ero mikropalveluarkkitehtuuriin (Red Hat 2023)

Aiemmin ennen mikropalveluiden yleistymistä on sovelluksia rakennettu useimmiten monoliittisella arkkitehtuurilla, jossa kaikki sovelluksen toiminnot ja palvelut ovat yhdessä paikassa, ja toimivat siten yhtenä kokonaisuutena. Kun sovellukseen halutaan lisätä suuria ominaisuuksia tai sitä parannetaan jollakin tavalla, arkkitehtuuri muodostuu monimutkaisemmaksi. (Red Hat 2023.)

8.1 Mikropalveluarkkitehtuurin hyödyt

Hyvärisen (2019) mukaan mikropalveluarkkitehtuuri antaa sovellukselle parempaa vikasietoisuutta ja suorituskykyä kuin monoliittinen arkkitehtuuri. Vikasietoisuudella tarkoitetaan mikropalveluissa sitä, että yhden mikropalvelun kaatuminen ei yleensä kaada koko järjestelmää. Kehittämisen näkökulmasta mikropalveluarkkitehtuuri voi nopeuttaa tuotteen toimitusta, koska kehitystiimit voivat työskennellä itsenäisesti.

Yksittäisiä mikropalveluita on helpompi päivittää ja ylläpitää kuin yhtä suurta monoliittista sovellusta. Palveluita voidaan skaalata erikseen yksi kerrallaan, mikä antaa sovellukselle joustavuutta ja tehokkuutta. Mikropalveluita voidaan myös testata palvelukohtaisesti, ne ovat tietoturvallisia ja ne sopivat hyvin CI/CD – teknologioihin käytettäväksi kertoo Andtfolk (2023).

8.2 Mikropalveluarkkitehtuurin haasteet

Mikropalvelut eivät kuitenkaan välttämättä sovi jokaiseen sovellusarkkitehtuurin ratkaisuun, koska ne saattavat olla myös haastavia ja monimutkaisia. Mikropalveluita ei kannata käyttää pieniin yksinkertaisiin sovelluksiin, sillä ne saattavat lisätä turhaa työtä ja voivat olla liian monimutkaisia pienille projekteille. (Andtfolk 2023.)

Tozzin (2024) mukaan ei myöskään kannata olettaa mikropalveluarkkitehtuurin olevan aina parempi vaihtoehto sovelluksen rakentamiseen. Vaikka mikropalvelut saattavat olla kustannustehokkaampia, niin ne vaativat enemmän hallinnointia ja työtä, mikä voi olla haastavaa pienemmille tiimeille ja budjetille. Mikropalvelut saattavat auttaa sovelluksen toimimaan tehokkaammin ja kuluttamaan vähemmän resursseja, mutta mahdollisesta myös päinvastoin.

9 SOVELLUKSEN TESTAUS JA LAADUNVARMISTUS

9.1 Yksikkötestaus

Yksikkötestaus viittaa ohjelmointiprosessiin, jossa sovellusta testataan automaattisilla ohjelmointikielellä rakennetuilla testeillä. Yksikkötestit ovat pieniä testejä, jotka testaavat pientä osa-aluetta koko sovelluskoodista (Schults

2024). Nykyään yksikkötestaamisen osaamista jopa vaaditaan nykyaikaisessa ohjelmistokehityksessä.

Yksikkötestaamisella voidaan testata sovelluskoodia, jotta nähdään, onko se vaatimusten ja suunnitelmien mukaisesti toteutettu, tai toimiiko se odotetulla tavalla ennen tuotantoon viemistä. Jos testaaminen on toteutettu ja rakennettu oikein, niin voidaan havaita ongelmat koodissa jo varhaisessa vaiheessa, ja siten säästää kustannuksissa huomattavasti. Yleensä yksikkötestit lisätään CI/CD-putkeen, jolloin sovellus testataan jokainen kerta, kun päähaaraan yritetään tehdä uusia muutoksia. Koska yksikkötestit eivät ole yhteydessä tietokantaan eikä mihinkään API:iin, niin testattavien metodien tai funktioiden muuttujat ja/tai arvot täytyy simuloida testiä kirjoittaessa. Simulointia varten yksikkötestauksessa on olemassa työkaluja, kuten Moq, NSubstitue, FakeItEasy, joiden avulla voidaan käyttää valeobjekteja oikeiden sijasta silloin, kun oikeita objekteja ei voida käyttää. (Schults 2024.)

Kun rakennetaan .NET-sovellusta, on saatavilla useampia yksikkötestauskehyksiä, joista muutamia ovat esimerkiksi NUnit, xUnit sekä MS-testi. .NET-ohjelmointikielille, kuten C# ja F#, paras työkalu on xUnit, joka on ilmainen ja avoimen lähdekoodin yksikkötestauskehys. (Vinugayathri s.a.) xUnit ominaisuuksien avulla voidaan kirjoittaa testejä, jotka ovat helposti luettavia, yksinkertaisia ja selkeitä. Yksikkötestaamisen hyötyihin ja haittoihin kuuluu (Vinugayathri s.a.):

Yksikkötestaamisen hyödyt:

- varmistaa koodin toiminnan
- havaitaan virheet ajoissa
- Itsedokumentoiva koodi.
- helpottaa koodin refaktorointia
- selkeyttää ja tiivistää koodia
- helpottaa koodin muutoksia

Yksikkötestaamisen haitat:

- Testit eivät paljasta kaikkia virheitä.
- Pienenkin funktion tai metodin testaamiseen voi joutua kirjoittamaan useita rivejä testikoodia.
- Testien rakentaminen oikein vaatii opettelua.
- Testien kirjoittaminen voi viedä paljon aikaa.

9.2 Kuntotarkastukset (Health Checks)

Ohjelmistoihin voidaan rakentaa kuntotarkastuksia (engl. Health Check), jotka tarjoavat helpon ja nopean tavan luoda tarkistuksia, joilla voidaan nopeasti nähdä sovelluksen tai järjestelmän eri osien tilanne. Ohjelmistojen kuntotarkastukset ovat erittäin tärkeitä, koska nykyään rakennetaan entistä laajempia ja monimutkaisempia mm. mikropalveluarkkitehtuuria hyödyntäviä sovelluksia. Sovelluksen seuranta on tärkeää käytettävyyden, resurssien käytön ja sovelluksen suorituskyvyn muutosten seuraamiseksi etenkin hajautetuissa järjestelmissä. (Jovanovic 2023.) ASP.NET Core tarjoaa valmiin Health Checks -työkalun sekä kirjaston, joilla voidaan tarkastella ja raportoida sovelluksen toimintaa. Monesti sovelluksien kuntotarkastukset ovat määritelty tarkistamaan sovelluksen kykyä käsitellä pyyntöjä ja sovellusta pidetään toimivana, jos se pystyy vastaamaan tarkastuksen päätepisteen URL-osoitteeseen. (Condron & Gutsch 2024.)

Code Mazen (2024) mukaan sovelluksen kuntotarkastuksia on erilaisia tyyppejä, koska sovellusta rakennettaessa voi olla monia riippuvuuksia tai palveluita, joita sovellus tarvitsee toimiakseen oikein. Tässä muutamia erilaisia sovelluksen kuntotarkastuksen muotoja:

- **Perus:** Yksinkertaisin tarkistusmuoto, jossa määritellään URL-osoite, josta voidaan tarkistaa sovelluksen tila. Jos tämä osoite vastaa terveellä tilaviestillä, tiedetään nopeasti, onko sovellus toimintakunnossa vai ei.

- **Järjestelmä:** Nämä kuntotarkastukset antavat tietoa palvelimen tilasta, kuten levytallennustilasta ja muistinkäytöstä. Tällä varmistetaan, että alustan toiminta on kunnossa, jolla sovellus toimii.

- **Tietokanta:** Näillä tarkistuksilla voidaan tarkistaa, onko tietokantapalvelu saatavilla ja voiko sovellus kommunikoida sen kanssa.

- **Mukautettu:** Tällä voidaan tarkistaa mikä tahansa ulkoinen palvelu tai API. Näitä ovat esimerkiksi kolmannen osapuolen palvelut tai tarkistukset, että lokitilaa on riittävästi.

9.3 GitHub Rulesets

GitHub Rulesets on työkalu, joka auttaa hallitsemaan, valvomaan ja suojaamaan GitHub-repositoriota sääntöjoukkojen avulla. Tällä työkalulla voidaan

asettaa selkeitä sääntöjä versionhallinnalle, mikä auttaa tiimejä hallitsemaan koodin selkeyttä ja turvallisuutta. (About rulesets s.a.) Sääntöjen (Rulesets) avulla voidaan sallia tietyille käyttäjille oikeus ohittaa sääntöjoukon säännöt, mutta voidaan myös rajoittaa toisien käyttäjien oikeuksia. GitHub Enterprise- ja GitHub Team tilauksilla voidaan määrittää sääntöjoukkoja organisaatiotasolla. GitHub Ruleset -työkalun avulla voidaan valvoa repositorion käyttöä ja varmistaa korkea laatu ja tietoturvallisuus. Tässä muutamia GitHub Ruleset -työkalun ominaisuuksia:

- **Käyttäjäoikeudet:** Voidaan määrittää tietyille käyttäjille oikeuksia muutoksien tekemiseen repositoriossa.
- **Haaran suojaaminen:** Voidaan hallita haarojen suojaamista estämällä esimerkiksi suorat muutokset päähaaraan. Voidaan vaatia koodin katselmointia ennen muutosten hyväksymistä.
- **Tietoturvallisuus:** Voidaan asettaa sääntöjä, joiden avulla varmistetaan koodin tietoturvavaatimukset mm. voidaan estää avaimien tai salasanojen lisääminen julkiseen repositorioon (About rulesets s.a.)

GitHub Ruleset -työkalulla voidaan myös hallita Pull Requesteja (PR), jotta koodin laatu säilyy korkeana ja voidaan olla varmoja, että muutokset ovat projektin standardien mukaisia. PR-hyväksymisoikeuksilla voidaan määrittää kuka voi tarkastella ja hyväksyä PR-pyynnön muutoksia. Yleensä PR-pyynnöt voi hyväksyä organisaation jäsenet, koodin omistajat sekä repositorion yhteiskäyttäjät.

PR:n hyväksymiseen voidaan luoda sääntöjä, joilla voidaan vaatia koodin katselmointia ennen PR:n hyväksymistä. Tämä sääntö auttaa tarkistamaan koodin vähintään kaksi kertaa ja näin saadaan myös minimoitua virheitä, sekä parannettua laatua. PR:n koodin katselmoinnissa voidaan jättää kommentteja PR:n tekijälle, pyytää tekemään muutoksia tai hyväksyä muutosten liittäminen. (DuMez 2024.) PR:lle voidaan myös asettaa sääntö, jossa CI (Continuous Integration) -prosessin täytyy mennä onnistuneesti läpi, ennen kuin sitä voidaan liittää päähaaraan (Triggering a workflow s.a.).

10 AZURE

Azure on Microsoftin pilvialusta, joka sisältää yli 200 tuotetta ja pilvipalvelua. Sen avulla voidaan rakentaa, käyttää ja hallita sovelluksia eri pilviympäristöissä, paikallisympäristöissä sekä reunaratkaisuissa hyödyntäen valitsemiaan työkaluja ja kehyksiä. (Microsoft s.a.)

10.1 Azure App Service

Azure App Service mahdollistaa verkkosovellusten, mobiilialustojen ja RESTful rajapintojen isännöinnin ja luomisen eri ohjelmointikielillä ilman infrastruktuurin hallintaa. App Service tarjoaa automaattisen skaalauksen ja korkean käytettävyyden ja se tukee Windows ja Linux ympäristöjä. Se myös mahdollistaa jatkuvan julkaisemisen ja toimituksen GitHubista, Azure DevOps ympäristöstä. (App Service Documentation s.a.)

App Service on automaattisesti hallittu palvelualustaratkaisu (PaaS). Sen ominaisuuksiin kuuluu esimerkiksi:

- **Tuki kielille ja kehyksille:** App service tukee ASP.NET, ASP.NET Core, Java, Node.js, PHP ja Python kieliä ja kehyksiä.
- **Hallittu tuotantoympäristö:** App service huolehtii automaattisesti käyttöjärjestelmän ja kielikehysten ylläpidosta ja päivityksistä.
- **Konttiteknologiat ja Docker:** App service mahdollistaa Windows tai Linux-konttien ylläpidon.
- **DevOps optimointi:** Jatkuva integraatio ja käyttöönotto
- **Globaali skaalaus ja korkea saatavuus:** App service mahdollistaa sovelluksen resurssien lisäämisen tai laajentamisen manuaalisesti ja automaattisesti.
- **Turvallisuus ja vaatimustenmukaisuus:** App service mahdollistaa IP-osoitteiden rajoittamisen ja palvelutunnusten hallitsemisen.
- **Autentikointi:** App service tarjoaa erilaisia autentikointikomponentteja.
- **Sovellusmallit:** Sovellusmallikokoelma sisältää eri sovellusmalleja, kuten WordPress, Joomla ja Drupal.
- **Visual Studio- ja Visual Studio Code -integraatio**: Visual studio ja Visual Studio Code sisältää työkaluja, jotka tehostavat sovelluksen luontia, käyttöönottoa ja virheenkorjausta.

- **API- ja mobiiliominaisuudet**. App Service tarjoaa valmiin CORS-tuen RESTful API -käyttötapauksia varten. Se myös yksinkertaistaa mobiilisovellusten käyttötapauksia mahdollistamalla autentikoinnin, offline-tietosynkronoinnin ja push-ilmoitukset (App Service Overview 2024.)

Yllä olevan luettelon ominaisuuksien avulla sovellusten kehitys, ylläpito ja turvallisuus voidaan toteuttaa tehokkaasti modernissa pilviympäristössä.

10.2 Azure Key Vault

Azure Key Vault on pilvipalvelu, jota käytetään salaisuuksien turvalliseen käyttöön ja tallentamiseen. Salaisuus tarkoittaa mitä tahansa, jonka käyttöoikeuksia halutaan hallita. Salaisuuksiin kuuluu esimerkiksi API-avaimet, salasanat, varmenteet tai salausavaimet. (Azure Key Vault basic concepts 2024.) Key Vault myös mahdollistaa salausavaimien hallinnan, joita käytetään tietojen salaamiseen sekä julkisten ja yksityisten Transport Layer Security/Security Sockets Layer (TLS/SSL) sertifikaattien hallintaan ja käyttöönottoon. (About Azure Key Vault 2024.)

Salaisuuksien asettaminen Azure Key Vault palveluun mahdollistaa salaisuuksien jakelun hallinnan. Key Vault vähentää vahingossa tapahtuvien tietojen paljastumisen riskiä. Palvelun avulla ohjelmistokehittäjien ei tarvitse tallentaa tietoturvallisia tietoja sovellukseen ja osaksi ohjelmakoodia. Esimerkiksi sen sijaan, että tietokannan yhteysmerkkijono tallennettaisiin ohjelmakoodiin, se voidaan tallentaa Key Vault palveluun. (About Azure Key Vault 2024.)

Key Vault käyttää asiakasjärjestelmän ja Key Vault:in välillä kulkevien tietojen suojaamiseen Transport Layer Security (TLS) protokollaa. TLS mahdollistaa vahvan todennuksen, viestien yksityisyyden ja eheyden, joka mahdollistaa viestien manipuloinnin, väliintulon ja väärennösten havaitsemisen. PFS tai Perfect Forward Secrecy suojaa asiakkaiden ja Microsoftin pilvipalveluiden välistä yhteyttä yksilöllisillä avaimilla. Yhteyksien suojaamiseen käytetään myös RSA-pohjaisia 2048 bittisiä salausavaimia. TLS ja PFS yhdistettynä mahdollistaa turvallisen tiedonsiirron ja se vaikeuttaa väliintuloa ja tiedon sieppaamista siirron aikana. (Azure Key Vault basic concepts 2024.)

Pääsy Key Vault resursseihin vaatii todentamisen ja valtuutuksen ennen kuin käyttäjä tai sovellus saa käyttöoikeuden. Todentaminen (Authentication) määrittelee käyttäjän identiteetin ja valtuudet (Authorization) määrittää sallitut toiminnot. Key Vault palvelussa todentaminen tapahtuu Microsoft Entra Id avulla, kun taas valtuutus todetaan Azure-roolipohjaisella pääsynhallinnalla (Azure RBAC) tai Key Vault pääsykäytännöillä (Access Policies). RBAC ominaisuutta käytetään holvien (Vaults) hallintaan ja tallennettujen tietojen käyttöön ja pääsykäytännöillä voidaan hallita vain tietojen käyttöä holvissa. (About Azure Key Vault 2024.)

10.3 Azure Monitor & Azure Application Insights

Azure Monitor on valvontaratkaisu, joka mahdollistaa valvontatietojen analysoinnin, keräämisen ja järjestelmätapahtumiin reagoimisen pilvi- ja paikallisympäristöissä sekä manuaalisesti että ohjelmallisesti. Azure Monitor palvelua voidaan käyttää sovellusten ja palveluiden saatavuuden ja suorituskyvyn maksimoimiseksi. (Azure Monitor overview 2024.)

Azure Application Insights on sovellusten suorituskyvyn valvontaratkaisu (APM) verkkosovelluksille, joka on osa Azure Monitor palvelua. Application Insights tarjoaa useita ominaisuuksia suorituskyvyn, laadun ja luotettavuuden parantamiseksi. Se mahdollistaa palvelun tutkimisen, seurannan, käytön ja ohjelmakoodin analyysin. (Application Insights overview 2024.)

Tutkinta

- **Sovelluksen kojelauta:** Mahdollistaa yleiskatsauksen sovelluksen terveydestä ja suorituskyvystä.
- **Sovelluskartta:** Näyttää visuaalisen kuvan komponenttien vuorovaikutuksesta ja sovelluksen arkkitehtuurista.
- **Live-metriikat:** Sovelluksen toiminnan ja suorituskyvyn seuraamiseen tarkoitettu reaaliaikainen analytiikkapaneeli.
- **Tapahtumahaku**: Diagnosoi ja jäljittää tapahtumia suorituskyvyn optimoimiseksi.
- **Saatavuusnäkymä:** Sovelluksen päätepisteiden (endpoints) saatavuuden ja reagointikyvyn testaus ja seuranta.
- **Virhenäkymä:** Analysoi ja tunnistaa sovelluksen virheitä käyttökatkojen minimoimiseksi.

- **Suorituskykymetriikat:** Suorituskykymittareiden analysointi ja suorituskykyongelmien tunnistaminen.

Seuranta

- **Hälytykset:** Sovelluksen osa-alueiden seuraus toimintojen aktivointi tarvittaessa.
- **Metriikat:** Analyysi suorituskykymittareista ja käyttötrendeistä (tavat, tilastot, mallit, liittyvät tiedot).
- **Diagnostiikka-asetukset:** Alustalokien ja metrikoiden määrittely valittuun kohteeseen.
- **Lokit:** Kerättyjen lokien ja tietojen analysointi.
- **Työkirjat:** Interaktiiviset raportit ja visuaaliset hallintapaneelit sovelluksen valvontatiedoista.

Käyttö

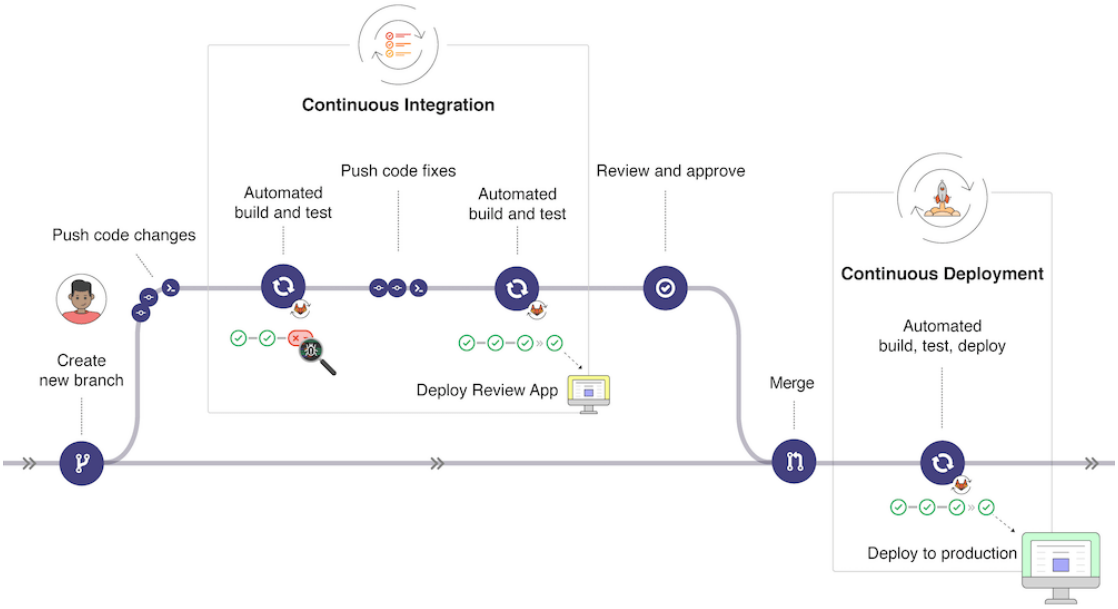
- **Käyttäjät, tapahtumat ja istunnot:** Käyttäjien vuorovaikutuksen seuraaminen sovelluksen kanssa.
- **Käyttäjäpolut:** Käyttäjäpolkujen analysointi ja kriittisten vaiheiden tunnistaminen
- **Kohortit**: Käyttäjien ryhmittely yhteisten ominaisuuksien perusteella suorituskykyongelmien ja trendien tunnistamiseksi.

Ohjelmakoodin analysointi

- **Profiilien luonti:** Suorituskykyjälkien tallentaminen ja analysointi
- **Optimointi:** Tekoälyn hyödyntäminen koodin tehokkuuden parantamiseen
- **Virheenkorjaus**: Virhetilanteiden automaattinen tilannekuvien kerääminen .NET sovelluksissa

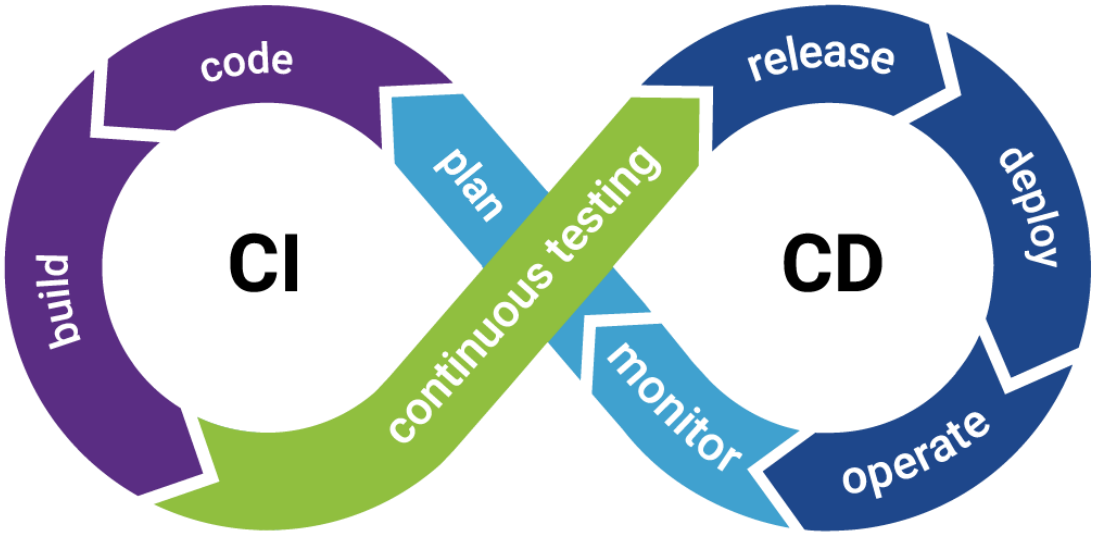
11 JATKUVA INTEGROINTI & TOIMITUS

Jatkuva integrointi (CI/Continuous Integration), jatkuva toimitus (CD/Continuous Delivery) ja jatkuva julkaisu (CD/Continuous Deployment) ovat ohjelmistokehityksen käytäntöjä, joiden avulla voidaan nopeuttaa ohjelmistokehityksen elinkaarta ja varmistetaan ohjelmiston laatua. Näitä työkaluja kutsutaan myös CI/CD-putkiksi, ja näiden työkalujen avulla kehitystiimit voivat toimittaa koodimuutoksia useammin ja nopeammin. Molemmat työkalut mahdollistavat ohjelmistojen tehokkaan julkaisun, jotta uusia laadukkaita tuotteita saadaan markkinoille nopeasti. (ABTasty s.a.)



Kuva 18. Jatkuvan integroinnin ja toimituksen prosessi (Melnyk 2021)

Kuvassa 18 esitetään kaavio CI/CD työkalujen toiminnasta. Kaavio kuvastaa uusien koodimuutoksien käyttöönoton elinkaarta.



Kuva 19. Havainnollistava kuva siitä, kuinka CI/CD-putket toimivat yhdessä (ABTasty s.a.)

Ohjelmistokehitysprosessi aloitetaan jatkuvalla integroinnilla (Continuous Integration), kuten kuvassa 19 on esitetty. Jatkuva toimitus ja jatkuva julkaisu sekoitetaan usein keskenään, vaikka jatkuva julkaisu vie asiat vielä askeleen pidemmälle.

11.1 Jatkuva integraatio (CI)

Jatkuva integraatio on ohjelmistokehityksen käytäntö, jossa kehittäjät yhdistävät tekemänsä koodimuutokset johonkin keskitettyyn versionhallintajärjestelmään, kuten mm. GitHub. Nykyaikaisessa ohjelmistokehityksessä työskennellään usein samanaikaisesti eri ominaisuuksien parissa, joten jatkuva integraatio vähentää yhteensopivuusongelmia sekä ristiriitoja muiden kehittäjien muutosten kanssa. CI-prosessin avulla voidaan integroida pieniä muutoksia usein, joka mahdollistaa nopeamman käyttöönoton, sekä palautteen, jotta kehittäjät voivat korjata mahdolliset virheet välittömästi. (ABTasty s.a.) Joka kerta, kun koodimuutoksia lähetetään, CI-prosessi käynnistää automaattisen testauksen ja rakentamisen, joka varmistaa koodin laadun ja löytää mahdolliset virheet.

Tämän prosessin avulla kehittäjien ei tarvitse odottaa muiden kehittäjien muutoksia, vaan koodimuutoksia integroidaan nimen mukaisesti jatkuvasti. Vaikka jatkuva integraatio voidaan ottaa käyttöön ilman jatkuvaa toimitusta (CD), ei voida kuitenkaan rakentaa CD:tä ilman, että CI-prosessi on rakennettu. (ABTasty s.a.)

11.2 Jatkuva toimitus & julkaiseminen (CD)

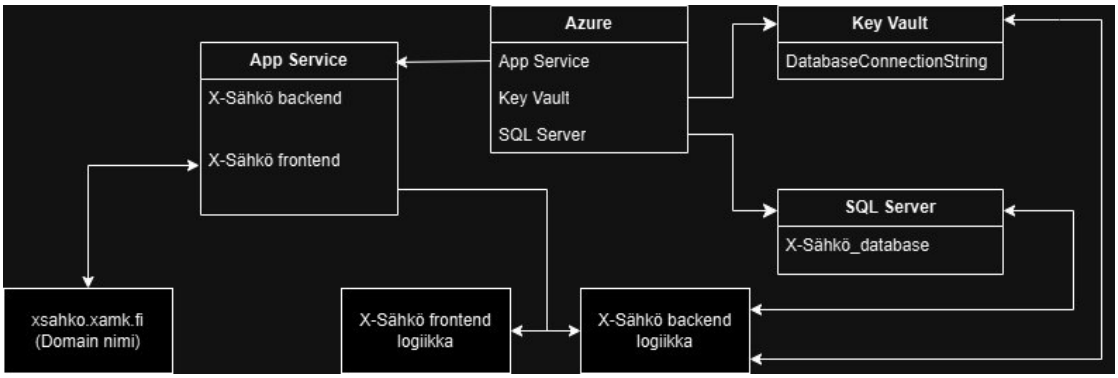
Jatkuva toimitus on ohjelmistokehityksen käytäntö, joka täydentää jatkuvaa integraatiota (CI) automatisoimalla infrastruktuurin käyttöönoton ja sovelluksen julkaisuprosessin. Kun CI-prosessi on rakentanut ja testannut ohjelmakoodin, CD täydentää viimeisen vaiheen varmistaen, että se on pakattu asiallisesti, jotta se voidaan ottaa käyttöön missä tahansa ympäristössä. Jatkuva toimitus kattaa infrastruktuurin ja sovelluksen käyttöönoton testaus tai tuotantoympäristöissä. Jatkuvan toimituksen avulla ohjelmisto voidaan rakentaa niin, että se voidaan ottaa käyttöön tuotannossa milloin tahansa. Tämän vaiheen jälkeen käyttöönotto voidaan käynnistää joko manuaalisesti tai siirtyä jatkuvaan julkaisemiseen, joka automatisoi käyttöönotot. (GitLab s.a.)

Jatkuva julkaisu tarkoittaa sovelluksen käyttöönottoa ilman, että kehittäjän tarvitsee manuaalisesti puuttua käyttöönotto prosessiin. Jatkuvassa julkaisussa DevOps-tiimit määrittelevät ohjelmakoodin julkaisukriteerit ja kun kriteerit vah-

vistetaan, ohjelmakoodi otetaan käyttöön tuotantoympäristössä. Jatkuva julkaisu mahdollistaa organisaatioille ketterämmän toiminnan ja nopeamman ominaisuuksien toimittamisen käyttäjille. (GitLab s.a.)

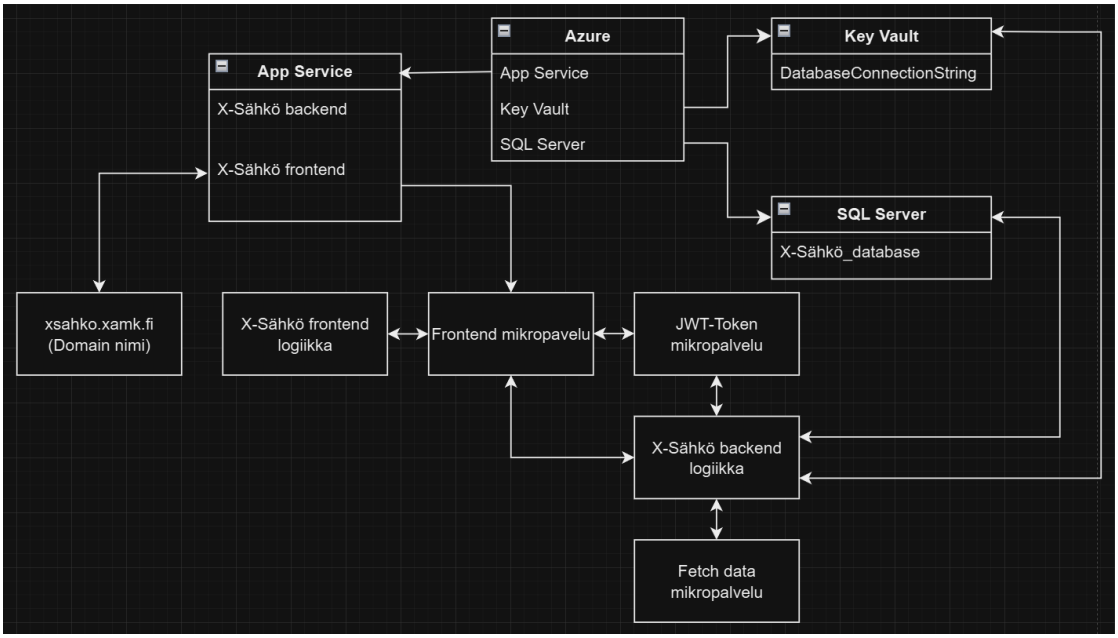
12 TOTEUTUS

Sovelluksen toteutus aloitettiin suorittamalla valmiiksi määriteltyjä tehtäviä, jotka oli luotu Linear-ohjelmistoon. Linear on tehokas työkalu projektinhallinnan ja työnkulun prosessien seurantaan ja tehostamiseen. Alkuvaiheessa suoritettiin paljon tiedonhakua, sekä datalähteiden etsimistä. Kun sopiva API oli löytynyt, aloitimme rakentamaan back-end-logiikkaa, jota testasimme jatkuvasti Swagger ja Postman -työkalujen avulla. Swagger ja Postman ovat työkaluja, jotka yksinkertaistavat ja helpottavat RESTful-palveluiden suunnittelua ja rakentamista.



Kuva 20. Projektin rakenne

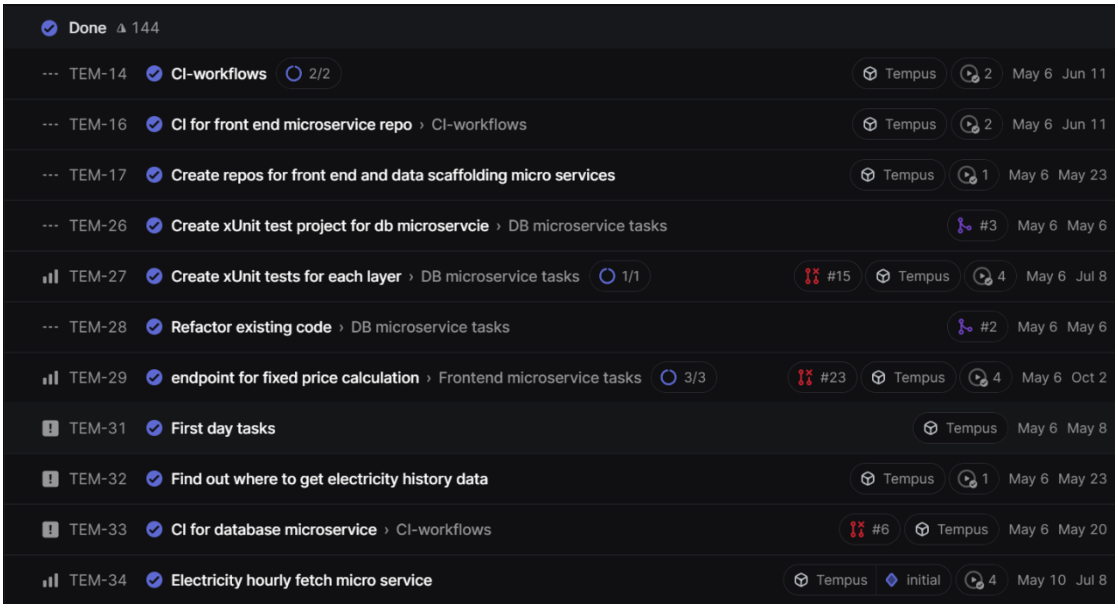
Kuvassa 20 esitetään kaavio X-Sähkö-projektin rakenteesta. Palvelukokonaisuus on julkaistu Azure-ympäristöön, joka sisältää kaikki palvelun elementit domain-nimeä lukuun ottamatta. Projektin käyttöliittymä ja palvelin logiikka on julkaistu Azure App Service -palveluun, ja ne kommunikoivat keskenään. Palvelussa tarvittava tietokanta sijaitsee myös Azuressa ja sen yhteysmerkkijono on asetettu Azuren Key Vault -palveluun salaisuutena. Back-end-logiikka kommunikoi tietokannan sekä Key Vaultin kanssa.



Kuva 21. Projektin rakenne mikropalveluita käyttäen

Alun perin sovellus rakennettiin mikropalveluiden avulla, joita oli yhteensä 4. Kuvassa 21 esitetään projektin rakennetta mikropalveluarkkitehtuuria käyttäen. Siinä ideana oli toteuttaa datan hakeminen, tietokanta, suojausavaimien palvelu, sekä laskentapalvelut omissa projekteissaan mikropalveluina. Suojausavainpalvelun ideana oli jakaa salausavaimia front-end-mikropalvelulle ja back-end palvelulle, jolloin tietoturvallisuus olisi parantunut. Front-end-mikropalvelun tehtävänä oli suorittaa kaikki laskutoimituksiin liittyvät tehtävät, ja back-end olisi ollut vastuussa vain datan tallentamisesta ja hakemisesta.

Mikropalveluarkkitehtuurista kuitenkin luovuttiin, koska projekti ei ollut niin laaja. Lopuksi yhdistimme kaikki rakentamamme palvelut yhteen projektiin, joka onnistui melko vaivattomasti käyttämämme arkkitehtuurimallin ansiosta.



Kuva 22. Linear ohjelmistoon annettuja tehtäviä

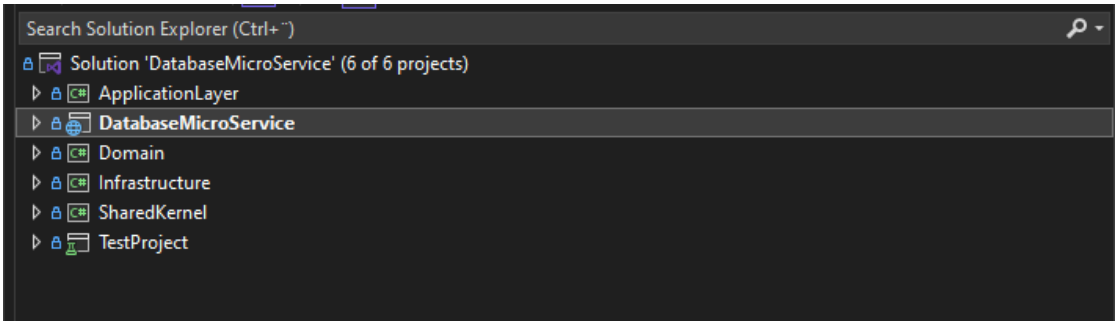
Kuvassa 22 on Linear-ohjelmistossa annettuja tehtäviä, jotka ovat suoritettuja. Jokainen tehtävä sisältää tarkasti määritellyt ohjeet ja ne siirtyvät keskeneräisestä valmiiksi sitä mukaa, kun tehtävä on suoritettu.

12.1 Back End toteutus

Sovelluksen back-end on sovelluksen logiikan ydin, joka on vastuussa sähköhintatietojen jatkuvasta hakemisesta ja tallentamisesta tietokantaan, sekä käyttäjien syötteiden käsittelystä ja datan palauttamisesta käyttäjälle. Back-end sisältää muun muassa laskentalogiikan, joka arvioi käyttäjän sähkönkulutusta ja sen hintaa käyttäjän antamien tietojen, kuten asumismuodon ja lämmitysmuodon perusteella. Back-end sisältää myös logiikan, joka käsittelee käyttäjän syöttämän sähkönkulutustiedoston ja laskee tiedoston perusteella hinnan pörssi- ja kiinteälle sähkölle.

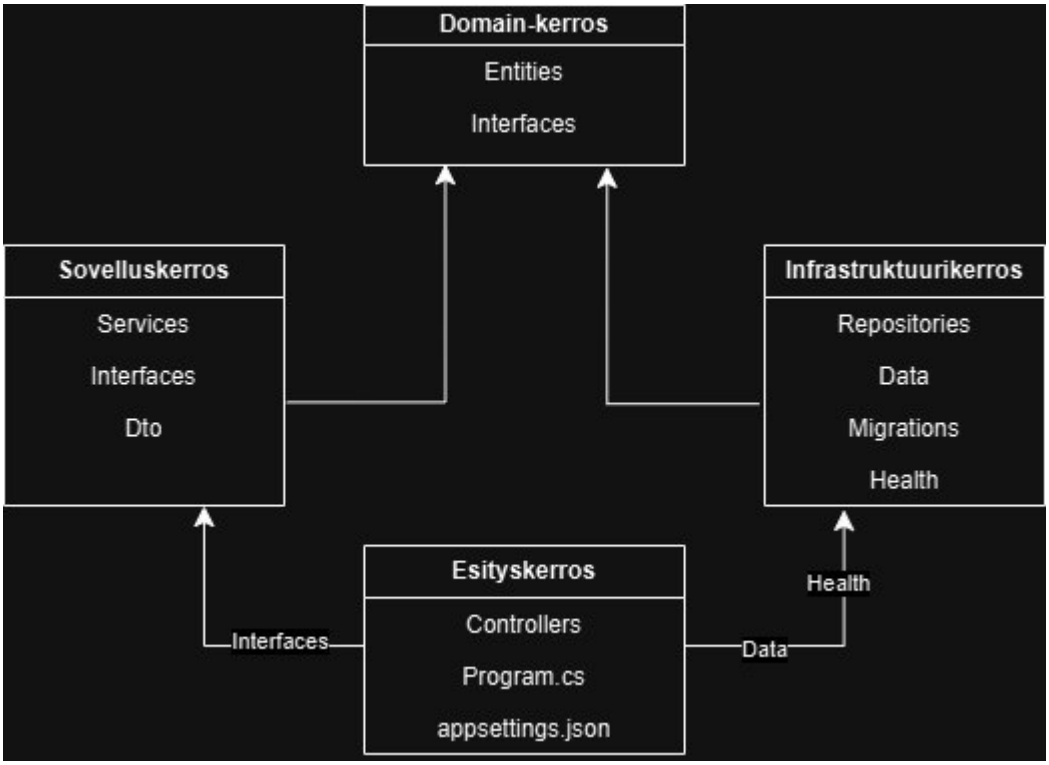
Back-end on toteutettu .NET ympäristössä hyödyntäen Clean Architecture -arkkitehtuurimallia, joka mahdollistaa modulaarisen ja ylläpidettävän ohjelmistorakenteen. Tietokantayhteyksien ja toteutuksen rakentamiseen käytetään Entity Framework Core (EF Core) ja ORM (Object Relational Mapping) tekniikoita, jotka mahdollistavat tietokannan taulujen luomisen C#-luokkien avulla sekä tietokantamigraatioiden luomisen ja hallinnan. Back-end sisältää myös erilaisia yksikkötestejä, joilla varmistetaan laskentalogiikoiden toiminta sekä kuntotarkastuksia, joilla voidaan varmistaa nopeasti palvelimen toiminta.

12.1.1 Back-end arkkitehtuuri



Kuva 23. Kuvakaappaus projektin rakenteesta

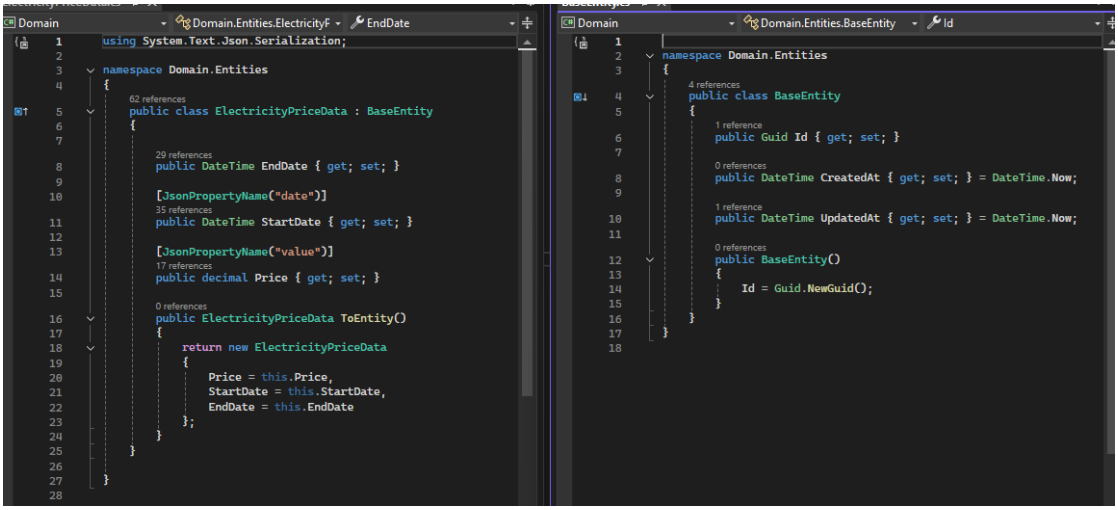
Kuvassa 23 näemme X-Sähkö-palvelun back-end toteutuksen rakenteen .NET ympäristössä, joka on rakennettu hyödyntäen Clean Architecture -mallia. Back-end sisältää Clean Architecture -mallin mukaisesti 4 kerrosta, sovelluskerroksen, domain-kerroksen, infrastruktuurikerroksen ja esityskerroksen. Kuvassa 21 esillä oleva DatabaseMicroService vastaa esityskerrosta. Back-end sisältää myös ”TestProject” projektin, joka sisältää kaikki yksikkötestit sekä ”SharedKernel” projektin, jota voidaan käyttää jatkossa muun muassa poikkeuksien, yhteisten tietotyyppien ja luokkien tai yhteisten apuluokkien ja -toiminnallisuuksien kehittämiseen kerrosten välillä.



Kuva 24. Kaavio arkkitehtuurikerroksien riippuvuuksista ja sisällöstä

Kuvassa 24 esitetään kaavio, jossa kuvataan back-end-palvelun arkkitehtuurikerroksien kansiorakennetta ja sisältöä. Kaaviossa on myös kuvattu riippuvuudet kerroksien välillä.

Clean Architecture -mallin ydin, eli **domain-kerros** sisältää tässä projektissa entiteetit, jotka ovat liiketoimintalogiikan tärkeimmät objektit. Entiteetit sisältävät luokkia, jotka määrittelevät sen, minkälaista dataa sovelluksessa käsitellään ja tallennettaan tietokantaan. Domain-kerros sisältää myös rajapinnat, jotka mahdollistavat kommunikoinnin ja datan siirron sovelluskerroksen ja infrastruktuurin välillä. Domain-kerroksen rajapintojen tehtävä on välittää entiteetit infrastruktuurikerrokselle datan käsittelyä ja tallentamista varten.



Kuva 25. Sovelluksen entiteetit

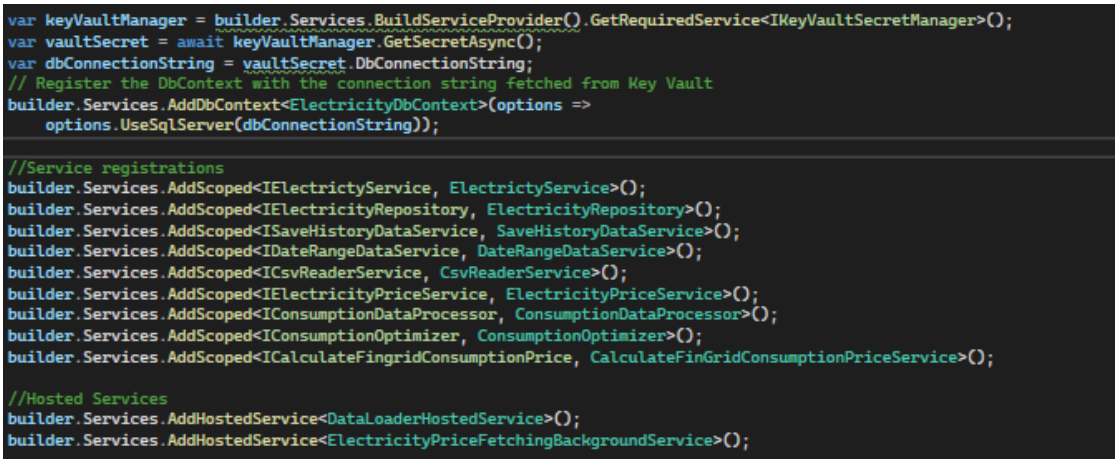
Kuvassa 25 esitetään ”ElectricityPriceData” ja ”BaseEntity” entiteetit. ”ElectricityPriceData” entiteetti toimii sovelluksen pääentiteettinä ja se määrittelee sen, minkälaista dataa tietokantaan tallennetaan ja käsitellään. Tämä luokka sisältää sähkön hinnan ja sen aikavälin. ”ElectricityPriceData” luokka perii ”BaseEntity” luokan, jonka tehtävä on antaa jokaiselle ”ElectricityPriceData” objektille oman yksilöllisen Id:n sekä sen luomisajankohdan ja päivitysajankohdan.

Sovelluskerroksen tehtävä sovelluksessa on käsitellä käyttäjän syötteitä, kuten kulutustiedostoa ja parametreja. Sovelluskerroksessa tapahtuu tärkeät las-

kutoimitukset, joiden avulla voidaan laskea esimerkiksi pörssisähkön ja kiinteän sähkön kokonaishintaa parametrien tai kulutustiedoston perusteella. Varsinaiset laskutoimitukset ja muut palvelut on asetettu ”Services” kansioon. ”Interfaces” kansio sisältää rajapintoja, joita käytetään kommunikointiin esityskerroksen kanssa. Dto (Data transfer object) -kansio sisältää kaikki datan käsittelyyn tarvittavat objektit. Dto-kansio sisältää esimerkiksi luokkia, jotka määrittelevät sen, minkälaista dataa palautetaan käyttäjälle. Sovelluskerros tarvitsee laskujen suorittamiseen sähköhintatietoja tietokannasta, ja sen takia sillä on riippuvuus domain-kerrokseen. Sovelluskerros antaa tai hakee dataa domain-kerroksen rajapinnan kautta, joka kommunikoi infrastruktuurikerroksen kanssa.

Infrastruktuurikerros on vastuussa datan tallentamisesta tietokantaan ja sen välittämisestä domain-kerrokselle. Tämän kerroksen ”Repositories” kansio sisältää kaiken sen logiikan, mikä tallentaa tai hakee dataa tietokannasta. ”Data” kansio sisältää ”Dbcontext” luokan, jota käytetään tietokannan taulujen luomiseen EF Core:n avulla. ”Migrations” -kansio sisältää kaikki tietokantamigraatiot ja Health-kansio sisältää kuntotarkastukset, joilla voidaan varmistaa tietokannan toimivuus. Infrastruktuurikerros on riippuvainen domain-kerroksen entiteeteistä ja rajapinnoista.

Esityskerros sisältää päätepisteet (endpoints), jotka ottavat vastaan käyttäjän syötteet. Nämä päätepisteet sijaitsevat ”Controllers”-kansiossa. Ohjaimien sisältävät päätepisteet määrittelevät sen, minkälaisia kutsuja päätepisteet voivat ottaa vastaan ja mitä kutsuihin vastataan. Päätepisteisiin voidaan myös määritellä tarkastuksia, joilla validoidaan saadun kutsun oikeellisuus. ”Controllers” osioon saadut kutsut ja data välitetään sovelluskerrokselle rajapintojen avulla, jossa suoritetaan varsinainen laskentalogiikka. Sovelluskerroksen rajapinnat on injektoitu ohjaimeen, ja tästä syystä syntyy riippuvuus esityskerrokselta sovelluskerrokselle.

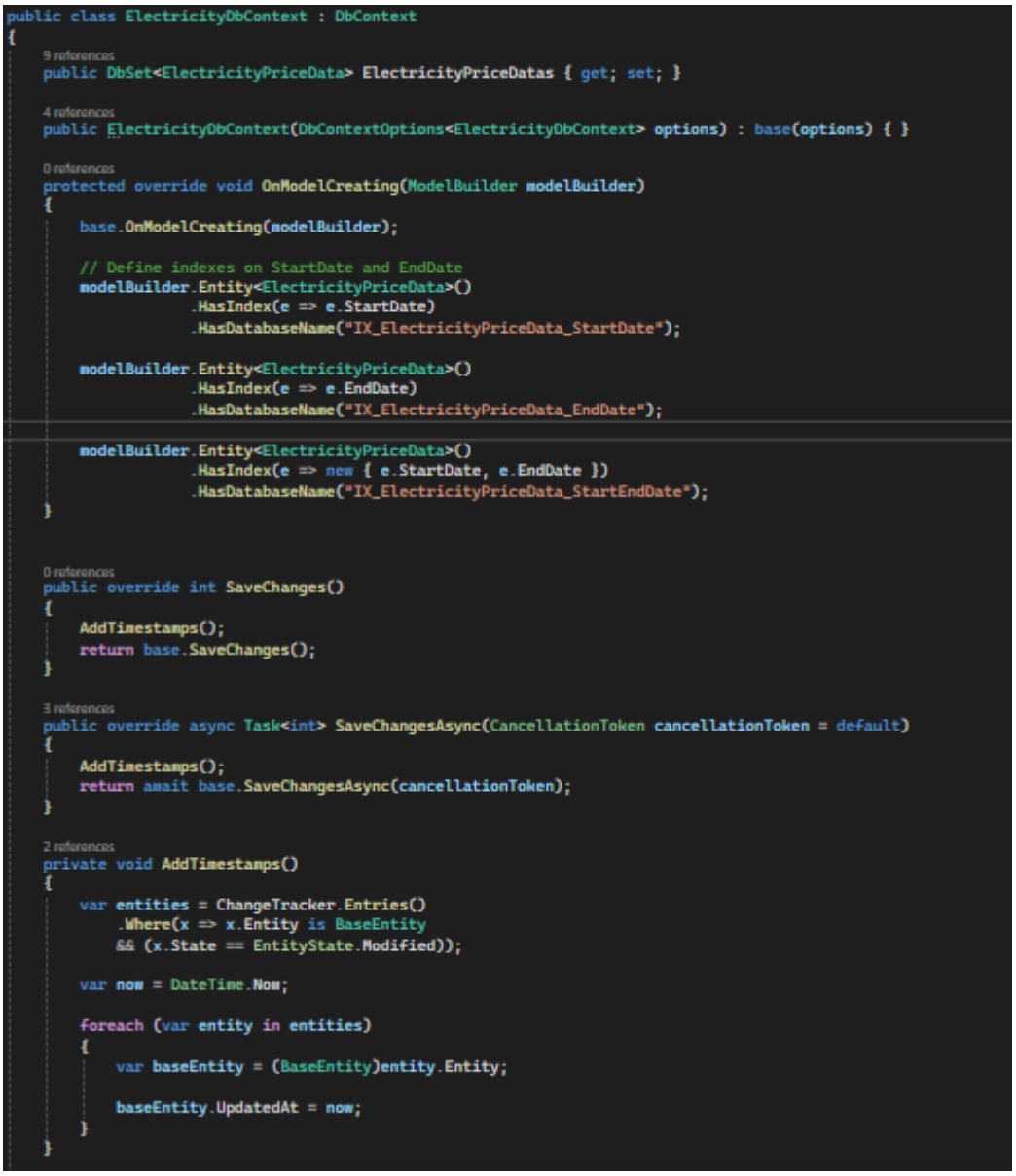


```
var keyVaultManager = builder.Services.BuildServiceProvider().GetRequiredService<IKeyVaultSecretManager>();
var vaultSecret = await keyVaultManager.GetSecretAsync();
var dbConnectionString = vaultSecret.DbConnectionString;
// Register the DbContext with the connection string fetched from Key Vault
builder.Services.AddDbContext<ElectricityDbContext>(options =>
    options.UseSqlServer(dbConnectionString));

//Service registrations
builder.Services.AddScoped<IElectrictyService, ElectrictyService>();
builder.Services.AddScoped<IElectricityRepository, ElectricityRepository>();
builder.Services.AddScoped<ISaveHistoryDataService, SaveHistoryDataService>();
builder.Services.AddScoped<IDateRangeDataService, DateRangeDataService>();
builder.Services.AddScoped<ICsvReaderService, CsvReaderService>();
builder.Services.AddScoped<IElectricityPriceService, ElectricityPriceService>();
builder.Services.AddScoped<IConsumptionDataProcessor, ConsumptionDataProcessor>();
builder.Services.AddScoped<IConsumptionOptimizer, ConsumptionOptimizer>();
builder.Services.AddScoped<ICalculateFingridConsumptionPrice, CalculateFinGridConsumptionPriceService>();

//Hosted Services
builder.Services.AddHostedService<DataLoaderHostedService>();
builder.Services.AddHostedService<ElectricityPriceFetchingBackgroundService>();
```

Kuva 26. Palveluiden ja tietokantalogiikan injektointi Program.cs tiedostoon

Esityskerros sisältää myös kaksi muuta tärkeää elementtiä, Program.cs ja appsettings.json tiedoston. Program.cs tiedoston tehtävä on rakentaa ja koota koko ohjelmakokonaisuus yhteen. Tähän tiedostoon siis injektoidaan esimerkiksi kaikki sovellus- ja infrastruktuurikerroksen palvelut. Program.cs tiedostoon injektoidaan myös tietokantalogiikka (DbContext) ja kuntotarkastukset, josta syntyy riippuvuus esityskerrokselta infrastruktuurikerrokselle. Appsettings.json tiedosto sisältää koko sovelluskokonaisuudelle tärkeitä tietoja ja arvoja, joita voidaan käyttää sovelluksessa. Se sisältää esimerkiksi Azure Key Vault osoitteen, josta haetaan tietokannan yhteysmerkkijono.

12.1.2 Tietokantatoteutus & datan tallentaminen



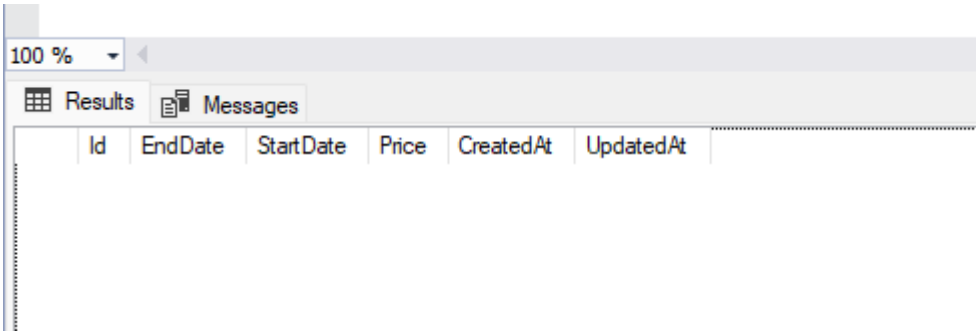
```
public class ElectricityDbContext : DbContext
{
    9 references
    public DbSet<ElectricityPriceData> ElectricityPriceDatas { get; set; }

    4 references
    public ElectricityDbContext(DbContextOptions<ElectricityDbContext> options) : base(options) { }

    0 references
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        // Define indexes on StartDate and EndDate
        modelBuilder.Entity<ElectricityPriceData>()
                    .HasIndex(e => e.StartDate)
                    .HasDatabaseName("IX_ElectricityPriceData_StartDate");

        modelBuilder.Entity<ElectricityPriceData>()
                    .HasIndex(e => e.EndDate)
                    .HasDatabaseName("IX_ElectricityPriceData_EndDate");

        modelBuilder.Entity<ElectricityPriceData>()
                    .HasIndex(e => new { e.StartDate, e.EndDate })
                    .HasDatabaseName("IX_ElectricityPriceData_StartEndDate");
    }


    0 references
    public override int SaveChanges()
    {
        AddTimestamps();
        return base.SaveChanges();
    }

    3 references
    public override async Task<int> SaveChangesAsync(CancellationToken cancellationToken = default)
    {
        AddTimestamps();
        return await base.SaveChangesAsync(cancellationToken);
    }

    2 references
    private void AddTimestamps()
    {
        var entities = ChangeTracker.Entries()
            .Where(x => x.Entity is BaseEntity
            && (x.State == EntityState.Modified));

        var now = DateTime.Now;

        foreach (var entity in entities)
        {
            var baseEntity = (BaseEntity)entity.Entity;

            baseEntity.UpdatedAt = now;
        }
    }
```

Kuva 27. ElectricityDbContext -luokka

Kuvassa 27 esitetään ”ElectricityDbContext”-luokka, joka perii Entity Framework Coren DbContext luokan. DbContext-luokka mahdollistaa tietokannan toiminnan sovelluskokonaisuuden kanssa. ”DbSet” luo tietokantataulun ”ElectricityPriceDatas” kuvan 25 entiteetti luokan avulla. Luokkaan on myös lisätty ”OnModelCreating”-metodi, joka asettaa taulun aloitus (StartDate) ja lopetus

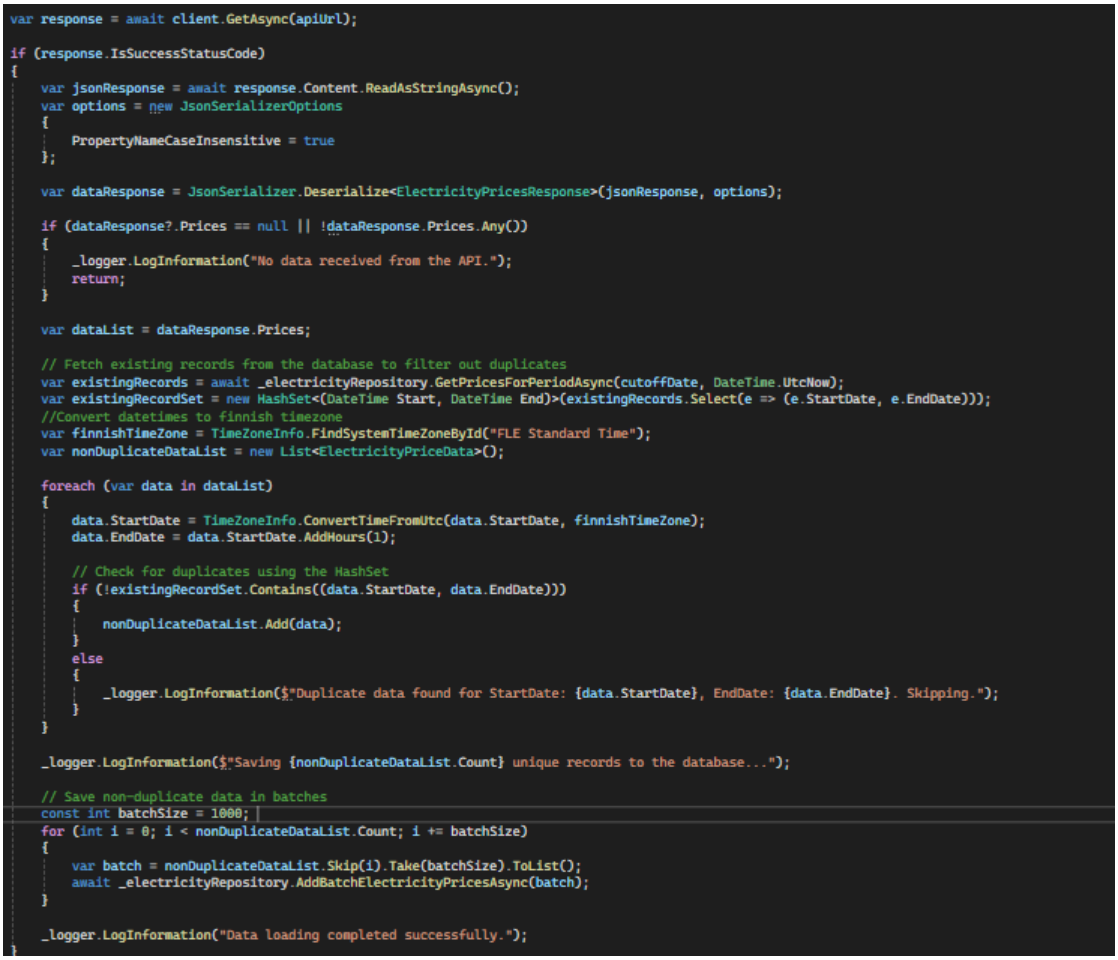
(EndDate) objekteille indeksit, joka mahdollistaa nopeamman sähkön hinta tietojen haun tietokannasta päivämäärien perusteella. ”SaveChanges” ja ”SaveChangesAsync”-metodit ovat vastuussa datan tallentamisesta tietokantaan.

Visual Studion Package Manager konsolissa voidaan ajaa ”Update-Database” komento, jolla luodaan tai päivitetään tietokantataulu, joka vastaa ”ElectricityDbContext” luokkaa.



Kuva 28. Tietokantanäkymä Update-Database-komennon jälkeen

Kun Update-Database komento on ajettu, voidaan ajaa komento Add-Migration, joka luo tietokantamigraation uusimpien muutosten pohjalta. Mikäli halutaan tehdä muutoksia tietokantaan ja myöhemmin palata takaisin edelliseen tietokantatoteutukseen, EF Core mahdollistaa migraatioiden hallinnan. Migraatioiden hallinnan avulla voidaan siis palauttaa tietokanta aikaisempaan migraatiotilaan, mikäli uusi toteutus ei olekaan miellyttävä tai tarpeellinen.



```
var response = await client.GetAsync(apiUrl);

if (response.IsSuccessStatusCode)
{
    var jsonResponse = await response.Content.ReadAsStringAsync();
    var options = new JsonSerializerOptions
    {
        PropertyNameCaseInsensitive = true
    };

    var dataResponse = JsonSerializer.Deserialize<ElectricityPricesResponse>(jsonResponse, options);

    if (dataResponse?.Prices == null || !dataResponse.Prices.Any())
    {
        _logger.LogInformation("No data received from the API.");
        return;
    }

    var dataList = dataResponse.Prices;

    // Fetch existing records from the database to filter out duplicates
    var existingRecords = await _electricityRepository.GetPricesForPeriodAsync(cutoffDate, DateTime.UtcNow);
    var existingRecordSet = new HashSet<(DateTime Start, DateTime End)>(existingRecords.Select(e => (e.StartDate, e.EndDate)));
    //Convert datetimes to finnish timezone
    var finnishTimeZone = TimeZoneInfo.FindSystemTimeZoneById("FLE Standard Time");
    var nonDuplicateDataList = new List<ElectricityPriceData>();

    foreach (var data in dataList)
    {
        data.StartDate = TimeZoneInfo.ConvertTimeFromUtc(data.StartDate, finnishTimeZone);
        data.EndDate = data.StartDate.AddHours(1);

        // Check for duplicates using the HashSet
        if (!existingRecordSet.Contains((data.StartDate, data.EndDate)))
        {
            nonDuplicateDataList.Add(data);
        }
        else
        {
            _logger.LogInformation($"Duplicate data found for StartDate: {data.StartDate}, EndDate: {data.EndDate}. Skipping.");
        }
    }

    _logger.LogInformation($"Saving {nonDuplicateDataList.Count} unique records to the database...");

    // Save non-duplicate data in batches
    const int batchSize = 1000;
    for (int i = 0; i < nonDuplicateDataList.Count; i += batchSize)
    {
        var batch = nonDuplicateDataList.Skip(i).Take(batchSize).ToList();
        await _electricityRepository.AddBatchElectricityPricesAsync(batch);
    }

    _logger.LogInformation("Data loading completed successfully.");
}
```

Kuva 29. Historiadatan hakeminen ja muuttaminen Suomen aikamuotoon

Kuvan 29 ohjelmakoodi hakee sähkönhintatietoja sähkötin.fi palvelusta JSON muodossa ja muuttaa ne Suomen aikamuotoon sekä tarkistaa onko tietokannassa duplikaatteja. Mikäli JSON-data sisältää duplikaatin, sitä ei lisätä tietokantaan. Muussa tapauksessa kaikki haetut tiedot lisätään tietokantaan.

```
private TimeSpan CalculateNextFetchTime(DateTime currentTime)
{
    var nextHour = currentTime.AddHours(1).Date.AddHours(currentTime.Hour + 1);
    var nextFetchTime = new DateTime(nextHour.Year, nextHour.Month, nextHour.Day, nextHour.Hour, 1, 0);
    return nextFetchTime - currentTime;
}

2 references
private async Task FetchElectricityPricesAsync(IElectrictyService electricityService, CancellationToken stoppingToke
{

    var spotPriceUrl = _configuration["ServiceUrls:SpotPriceUrl"];
    var httpClient = _httpClientFactory.CreateClient();
    _logger.LogInformation("Fetching electricity prices from {Url}", spotPriceUrl);
    var stopwatch = System.Diagnostics.Stopwatch.StartNew();

    try
    {
        var response = await httpClient.GetAsync(spotPriceUrl, stoppingToken);
        var responseTime = stopwatch.Elapsed;
        _logger.LogInformation("Received response in {ResponseTime}ms", responseTime.TotalMilliseconds);

        response.EnsureSuccessStatusCode();
        var content = await response.Content.ReadAsStringAsync();
        _logger.LogInformation("Prices fetched: {Content}", content);

        if (string.IsNullOrWhiteSpace(content))
        {
            _logger.LogWarning("Fetched data is empty. Retrying after 1 minute.");
            await Task.Delay(TimeSpan.FromMinutes(1), stoppingToken);
            await FetchElectricityPricesAsync(electricityService, stoppingToken);
            return;
        }
        await SaveElectricityPrices(content, electricityService);
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Error fetching electricity prices");
    }
}
```

Kuva 30. Ohjelmakoodi, joka hakee tunnin välein uusimmat sähkönhintatiedot

```
1 reference
private async Task SaveElectricityPrices(string electricityDataJson, IElectrictyService electricityService)
{
    try
    {
        var electricityData = JsonConvert.DeserializeObject<ElectricityPriceDataDtoIn>(electricityDataJson);
        if (electricityData != null && electricityData.Prices != null && electricityData.Prices.Any())
        {
            // Define the Finnish time zone
            var finnishTimeZone = TimeZoneInfo.FindSystemTimeZoneById("FLE Standard Time");

            // Adjust StartDate to Finnish time and set EndDate
            foreach (var data in electricityData.Prices)
            {
                // Convert the StartDate from UTC to Finnish time
                data.StartDate = TimeZoneInfo.ConvertTimeFromUtc(data.StartDate, finnishTimeZone);

                // Set the EndDate to one hour after the StartDate
                data.EndDate = data.StartDate.AddHours(1);
            }

            // Save the adjusted data to the database
            bool success = await electricityService.AddElectricityPricesAsync(electricityData);
            if (success)
            {
                _logger.LogInformation("Data saved successfully to the database.");
            }
            else
            {
                _logger.LogWarning("No new data was added to the database (all duplicates or invalid entries).");
            }
        }
        else
        {
            _logger.LogWarning("Invalid data received. No data saved to the database.");
        }
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Error occurred while saving data to the database.");
    }
}
```

Kuva 31. Ohjelmakoodi, joka tallentaa Kuvan 29 ohjelmakoodissa haetut tiedot tietokantaan

Kuvassa 30 ja 31 esitetään logiikkaa, joka mahdollistaa uusimpien sähkönhinta tietojen hakemisen ja tallentamisen tietokantaan. Kuvan 30 metodi ”FetchElectricityPricesAsync” hakee ensin sovelluksen konfiguraatiosta (appsettings.json) porssisahko.net API-päätepisteen URL-osoitteen ja käyttää sitä uuden datan hakemiseen. Mikäli jostain syystä dataa ei vastaanotettu, ohjelma odottaa minuutin, jonka jälkeen se yrittää hakea dataa uudestaan. Mikäli datan haku onnistuu, se annetaan Kuvassa 31 esiintyvälle ”SaveElectricityPrices”-metodille. Kuvassa 30 on esillä myös ”CalculateNextFetchTime”-metodi, jonka avulla haetaan joka tunnin välein uutta dataa API-toteutuksesta sovelluksen käynnistysajasta riippumatta.

Kuvassa 31 esillä olevan metodin ”SaveElectricityPrices” tehtävä on muuttaa uudet sähkönhinta tiedot Suomen aikamuotoon ja lisää ne tietokantaan. Logiikkaan kuuluu duplikaattien tarkistus, jonka avulla vältetään ylimääräisten hinta tietojen tallentaminen tietokantaan.

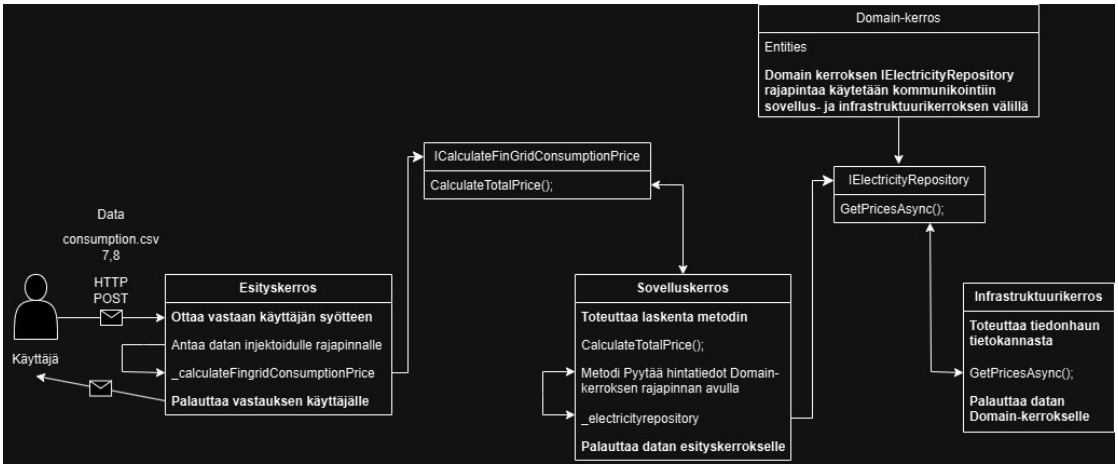
100 %

Results | Messages

	Id	EndDate	StartDate	Price	CreatedAt	UpdatedAt
1	1B552B6A-12DB-4D1A-9417-000134CDD7C9	2024-07-20 21:00:00.0000000	2024-07-20 20:00:00.0000000	3.13	2024-11-22 14:37:30.1460716	2024-11-22 14:37:30.1460717
2	1C45E884-DF1C-4C73-BEFF-0002407782A2	2019-06-25 15:00:00.0000000	2019-06-25 14:00:00.0000000	7.50	2024-11-22 14:37:30.0698422	2024-11-22 14:37:30.0698423
3	8998D74A-6209-45BC-B3EE-00028170876C	2024-01-23 17:00:00.0000000	2024-01-23 16:00:00.0000000	3.60	2024-11-22 14:37:30.1390252	2024-11-22 14:37:30.1390254
4	D5685780-5D6E-452A-AC23-000458C4C3E6	2021-11-30 19:00:00.0000000	2021-11-30 18:00:00.0000000	32.75	2024-11-22 14:37:30.1029391	2024-11-22 14:37:30.1029392

Kuva 32. Tietokantanäkymä hintatietojen tallentamisen jälkeen

12.1.3 Datan hallinta kerrosten välillä ja laskentalogiikka



Kuva 33. Esimerkki datan kulusta sovelluksen kerrosten välillä

Kuvassa 33 esitetään kaavio, joka kuvastaa sitä, miten käyttäjän syöttämä data käsitellään sovelluksen eri kerroksissa. Esimerkki kuvastaa tilannetta,

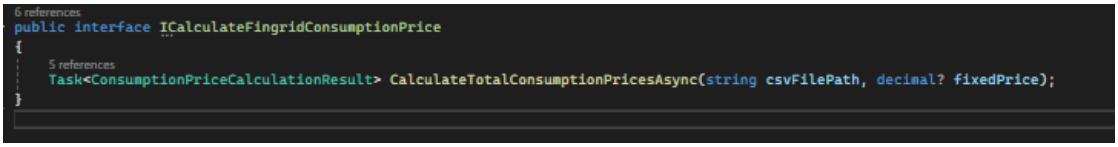
jossa käyttäjä on lähettänyt back-end-palveluun oman kulutustiedoston ja vertailtavan kiinteän sähkön hinnan.

Esityskerroksen päätepiste (endpoint) ottaa vastaan tiedoston ja kiinteän sähkön hinnan, joka annetaan sovelluskerrokselle injektoidun rajapinnan avulla. ”ICalculateFinGridConsumptionPrice”-rajapinta toteuttaa ”CalculateTotalPrice”-metodin sovelluskerroksen sisällä. Tämän metodin tehtävä on siis käsitellä kulutustiedosto ja tehdä laskut. Laskuja varten tarvitaan tietokannasta sähkönhintatietoja, jotka pyydetään injektoidun rajapinnan avulla. Tämä rajapinta sijaitsee domain-kerroksessa. Domain-kerroksen rajapinta ”IElectricityRepository” toteuttaa infrastruktuurikerroksessa sijaitsevan ”GetPricesAsync”-metodin, joka hakee tarvittavat sähkönhintatiedot. Tiedot palautetaan domain-kerroksen kautta sovelluskerrokselle, jossa laskut suoritetaan. Lopulta sovelluskerros palauttaa datan esityskerrokselle, ja esityskerros palauttaa vastauksen käyttäjälle.

```
[HttpPost("UploadFinGridConsumptionFile")]
0 references
public async Task<IActionResult> UploadCsv(IFormFile file, [FromQuery] decimal? fixedPrice)
{
    // Start a stopwatch to measure response time
    var stopwatch = Stopwatch.StartNew();

    LogRequestDetails("upload Fingrid consumption file");

    if (file == null || file.Length == 0)
        return BadRequest("File not provided.");

    var filePath = Path.Combine(Directory.GetCurrentDirectory(), file.FileName);

    try
    {
        using (var stream = new FileStream(filePath, FileMode.Create))
        {
            await file.CopyToAsync(stream);
        }

        if (!fixedPrice.HasValue)
        {
            return BadRequest("Fixed price not received");
        }

        var result = await _calculateFinGridConsumptionPrice.CalculateTotalConsumptionPricesAsync(filePath, fixedPrice);

        // Stop the stopwatch and log the response time and status code
        stopwatch.Stop();
        _logger.LogInformation("Response Time: {Time} ms", stopwatch.ElapsedMilliseconds);
        _logger.LogInformation("Response Status Code: {StatusCode}", StatusCodes.Status200OK);

        return Ok(result);
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Error processing file.");
        return StatusCode(StatusCodes.Status500InternalServerError, "Error processing file.");
    }
    finally
    {
        if (System.IO.File.Exists(filePath))
        {
            System.IO.File.Delete(filePath);
        }
    }
}
```

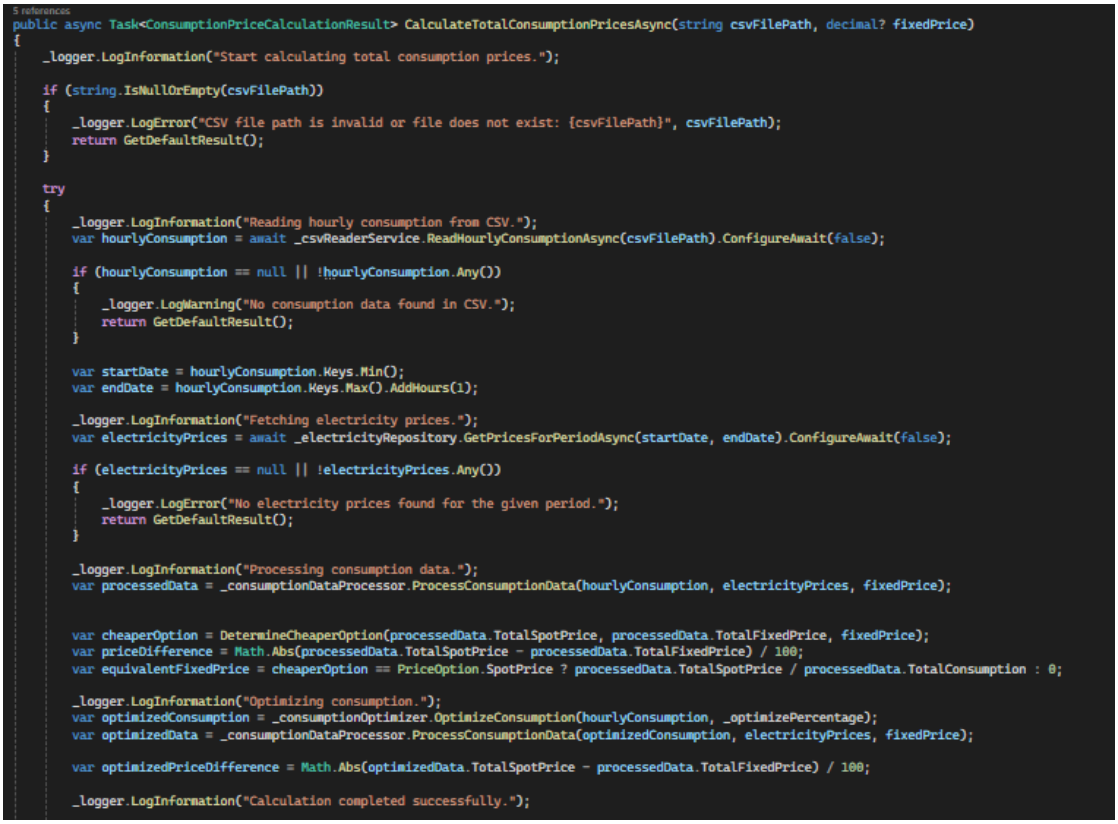
Kuva 34. Päätepiste esityskerroksessa, joka ottaa vastaan käyttäjän kulutustiedoston ja kiinteän sähkönhinnan.

Kuvassa 34 esitetään päätepiste, jonka tehtävä on ottaa vastaan käyttäjän kulutustiedosto ja kiinteä hinta. Kulutustiedosto tallennetaan väliaikaisesti sovelluksen hakemistoon ja sille asetetaan tilapäinen tiedostopolku. Kun tiedosto on käsitelty, se poistetaan hakemistosta. Tämä logiikka validoi myös syötteen ja ilmoittaa virheestä, mikäli tiedostoa tai kiinteää hintaa ei annettu. Kun saatu data on validoitu oikeanlaiseksi, se annetaan sovelluskerroksen metodille käsiteltäväksi rajapinnan kautta kohdassa ”_calculateFinGridConsumptionPrice.CalculateTotalConsumptionPricesAsync”.



Kuva 35. Rajapinta, joka ottaa vastaan kulutustiedoston ja kiinteän hinnan

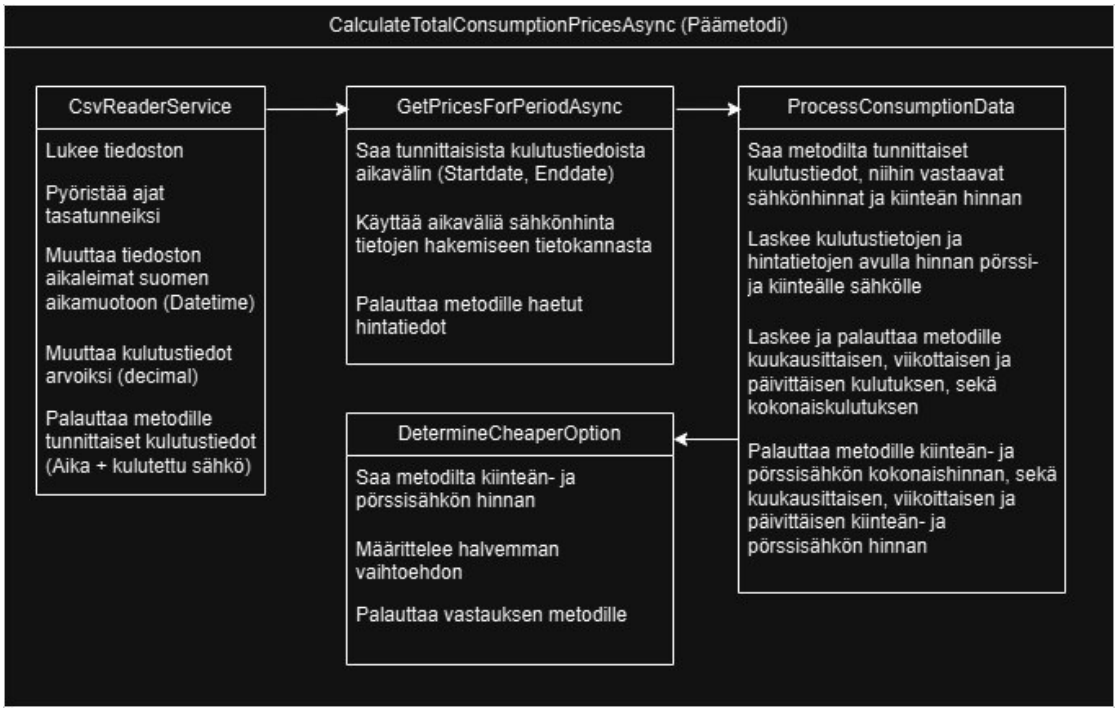
Kuvan 35 rajapinta ottaa vastaan esityskerrokselta kulutustiedoston tiedostopolun sekä kiinteän hinnan, joita käytetään metodissa laskujen suorittamiseen. Rajapinnassa esiintyvä ”ConsumptionPriceCalculationResult” tarkoittaa Dto-luokkaa, joka määrittelee sen mitä dataa esityskerrokselle palautetaan tämän rajapinnan kautta.



```
5 references
public async Task<ConsumptionPriceCalculationResult> CalculateTotalConsumptionPricesAsync(string csvFilePath, decimal? fixedPrice)
{
    _logger.LogInformation("Start calculating total consumption prices.");

    if (string.IsNullOrEmpty(csvFilePath))
    {
        _logger.LogError("CSV file path is invalid or file does not exist: {csvFilePath}", csvFilePath);
        return GetDefaultResult();
    }

    try
    {
        _logger.LogInformation("Reading hourly consumption from CSV.");
        var hourlyConsumption = await _csvReaderService.ReadHourlyConsumptionAsync(csvFilePath).ConfigureAwait(false);

        if (hourlyConsumption == null || !hourlyConsumption.Any())
        {
            _logger.LogWarning("No consumption data found in CSV.");
            return GetDefaultResult();
        }

        var startDate = hourlyConsumption.Keys.Min();
        var endDate = hourlyConsumption.Keys.Max().AddHours(1);

        _logger.LogInformation("Fetching electricity prices.");
        var electricityPrices = await _electricityRepository.GetPricesForPeriodAsync(startDate, endDate).ConfigureAwait(false);

        if (electricityPrices == null || !electricityPrices.Any())
        {
            _logger.LogError("No electricity prices found for the given period.");
            return GetDefaultResult();
        }

        _logger.LogInformation("Processing consumption data.");
        var processedData = _consumptionDataProcessor.ProcessConsumptionData(hourlyConsumption, electricityPrices, fixedPrice);


        var cheaperOption = DetermineCheaperOption(processedData.TotalSpotPrice, processedData.TotalFixedPrice, fixedPrice);
        var priceDifference = Math.Abs(processedData.TotalSpotPrice - processedData.TotalFixedPrice) / 100;
        var equivalentFixedPrice = cheaperOption == PriceOption.SpotPrice ? processedData.TotalSpotPrice / processedData.TotalConsumption : 0;

        _logger.LogInformation("Optimizing consumption.");
        var optimizedConsumption = _consumptionOptimizer.OptimizeConsumption(hourlyConsumption, _optimizePercentage);
        var optimizedData = _consumptionDataProcessor.ProcessConsumptionData(optimizedConsumption, electricityPrices, fixedPrice);

        var optimizedPriceDifference = Math.Abs(optimizedData.TotalSpotPrice - processedData.TotalFixedPrice) / 100;

        _logger.LogInformation("Calculation completed successfully.");
```

Kuva 36. Sovelluskerroksen metodi, joka tekee laskut kulutustiedoston ja kiinteän hinnan perusteella

Kuvan 36 ohjelmakoodin tehtävä on käsitellä annettu kulutustiedosto ja suorittaa laskut. Tämä metodi on ”päämetodi”, joka kutsuu muita sivupalveluita sen sisällä. Muita palveluita käytetään esimerkiksi kulutustiedoston käsittelyyn sekä hintatietojen hakemiseen.



Kuva 37. Päämetodin logiikka

Kuvassa 37 esitetään kaavio logiikasta, jonka avulla käsitellään kulutustiedosto ja lasketaan kiinteän- ja pörssisähkön hinta sekä sähkönkulutuksen määrä. Kuvassa esitetään päämetodin sisällä käytettyjä palveluita ja niiden tehtäviä. Esimerkiksi ”CsvReaderService” on vastuussa kulutustiedoston käsittelystä. Muut palvelut, kuten ”GetPricesForPeriodAsync” ja ”ProcessConsumptionData” ovat vastuussa sähkönhintatietojen hakemisesta sekä laskutoimituksien suorittamisesta.

```
return new ConsumptionPriceCalculationResult
{
    TotalSpotPrice = processedData.TotalSpotPrice / 100,
    TotalFixedPrice = processedData.TotalFixedPrice / 100,
    CheaperOption = cheaperOption.ToString(),
    TotalConsumption = processedData.TotalConsumption,
    PriceDifference = priceDifference,
    OptimizedPriceDifference = optimizedPriceDifference,
    EquivalentFixedPrice = equivalentFixedPrice,
    TotalOptimizedSpotPrice = optimizedData.TotalSpotPrice / 100,
    MonthlyData = DataFormatter.FormatMonthlyData(processedData.MonthlyData),
    WeeklyData = DataFormatter.FormatWeeklyData(processedData.WeeklyData),
    DailyData = DataFormatter.FormatDailyData(processedData.DailyData),
    StartDate = startDate,
    EndDate = endDate
};
```

Kuva 38. ”CalculateTotalConsumptionPricesAsync” metodin ohjelmakoodi, joka palauttaa Dto objektin

Kun kaikki palvelun vaiheet on suoritettu, päämetodi palauttaa tulokset Dto objektina rajapinnalle ja rajapinta palauttaa vastauksen esityskerrokselle.

Response body


```
{
  "TotalSpotPrice": 87.580433,
  "TotalFixedPrice": 98.4768,
  "CheaperOption": "SpotPrice",
  "TotalConsumption": 1230.96,
  "PriceDifference": 10.896367,
  "OptimizedPriceDifference": 14.277045,
  "EquivalentFixedPrice": 7.114807386105154,
  "TotalOptimizedSpotPrice": 84.199755,
  "MonthlyData": [
    {
      "Year": 2023,
      "Month": 1,
      "Consumption": 123.39,
      "SpotPrice": 11.739871,
      "FixedPrice": 9.8712
    },
    {
      "Year": 2023,
      "Month": 2,
      "Consumption": 104.72,
      "SpotPrice": 10.119155,
      "FixedPrice": 8.3776
    },
```


Kuva 39. Esityskerroksen palauttama data Swagger-käyttöliittymässä

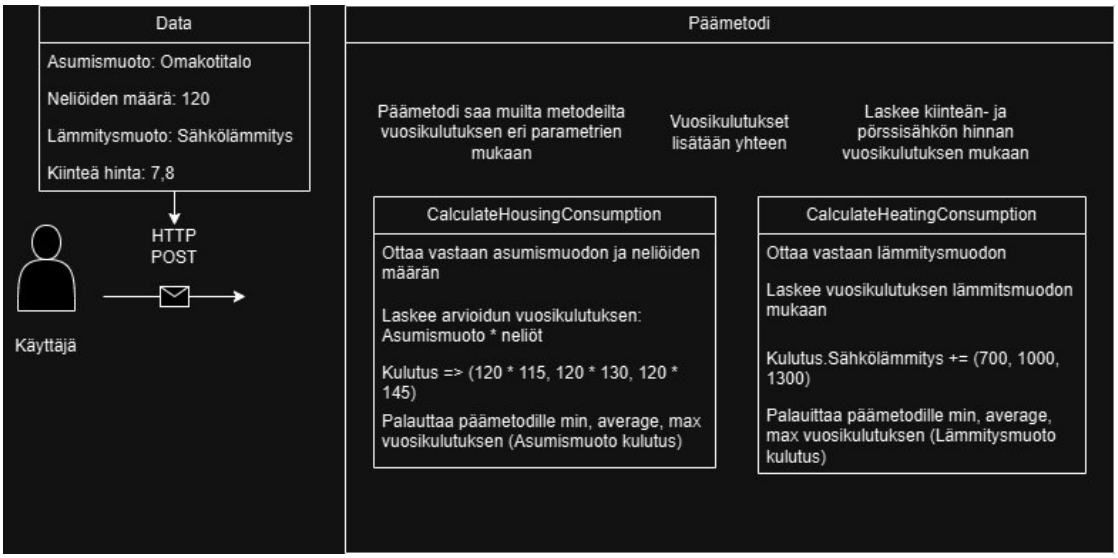
```
private ConsumptionResult CalculateTotalConsumption(CombinedRequestDtoIn request)
{
    var housingConsumption = CalculateHousingConsumption(request.HouseType, request.SquareMeters);
    var workShiftConsumption = CalculateWorkShiftConsumption(request.WorkShiftType);
    var saunaConsumption = CalculateSaunaConsumption(request.HasSauna, request.SaunaHeatingFrequency);
    var fireplaceSavings = CalculateFireplaceSavings(request.HasFireplace, request.FireplaceFrequency);
    var electricCarConsumption = CalculateElectricCarConsumption(request);
    var residentConsumption = CalculateResidentConsumption(request.NumberOfResidents, request.HouseType);
    var heatingConsumption = CalculateHeatingConsumption(request);
    var solarPanelSavings = CalculateSolarPanelSavings(request);
    var floorHeatingConsumption = CalculateFloorHeatingConsumption(request);

    decimal minConsumption = housingConsumption.Min + workShiftConsumption * 365 + saunaConsumption
                           + electricCarConsumption + residentConsumption.Min + heatingConsumption.Min
                           + floorHeatingConsumption - fireplaceSavings - solarPanelSavings;

    decimal averageConsumption = housingConsumption.Average + workShiftConsumption * 365 + saunaConsumption
                               + electricCarConsumption + residentConsumption.Average + heatingConsumption.Average
                               + floorHeatingConsumption - fireplaceSavings - solarPanelSavings;

    decimal maxConsumption = housingConsumption.Max + workShiftConsumption * 365 + saunaConsumption
                           + electricCarConsumption + residentConsumption.Max + heatingConsumption.Max
                           + floorHeatingConsumption - fireplaceSavings - solarPanelSavings;

    _logger.LogInformation("Total consumption calculated: Min={Min}, Average={Average}, Max={Max}",
        minConsumption, averageConsumption, maxConsumption);

    return new ConsumptionResult
    {
        MinConsumption = minConsumption,
        AverageConsumption = averageConsumption,
        MaxConsumption = maxConsumption
    };
}
```

Kuva 40. Metodi, joka laskee arvioiduin kulutuksen

```
// Calculate total consumption
var consumptionResult = await Task.Run(() => CalculateTotalConsumption(request));

// Calculate costs
var totalFixedPriceCost = CalculateYearlyCost(request.FixedPrice, consumptionResult.AverageConsumption);
var totalSpotPriceCost = CalculateYearlySpotPrice(electricityPriceData, consumptionResult.AverageConsumption);

// Calculate costs for min and max consumption
var minFixedPriceCost = CalculateYearlyCost(request.FixedPrice, consumptionResult.MinConsumption);
var maxFixedPriceCost = CalculateYearlyCost(request.FixedPrice, consumptionResult.MaxConsumption);
var minSpotPriceCost = CalculateYearlySpotPrice(electricityPriceData, consumptionResult.MinConsumption);
var maxSpotPriceCost = CalculateYearlySpotPrice(electricityPriceData, consumptionResult.MaxConsumption);

// Calculate monthly data
int lastMonthToConsider = endDate.Month; // Get the last month based on endDate
var monthlyData = await CalculateMonthlyDataAsync(consumptionResult.AverageConsumption, electricityPriceData, request.FixedPrice, request.Year, endDate);

var costDifference = Math.Round(Math.Abs(totalFixedPriceCost - totalSpotPriceCost), 2);
var cheaperOption = totalFixedPriceCost < totalSpotPriceCost ? "Fixed price" : "Spot price";
var averageHourlySpotPrice = CalculateAverageYearlySpotPrice(electricityPriceData);
```

Kuva 41. Päämetodin ohjelmakoodi, joka laskee hinnan kiinteän- ja pörssisähkön hinnan

Kuvassa 40 esitetään back-end-palvelun toisen päätepisteen laskentalogiikkaa, joka laskee arvioidun sähkönkulutuksen määrän. Sille tarkoitettu päätepiste ottaa vastaan esimerkiksi käyttäjän asumismuodon, asunnon neliömäärän ja työmuodon. Laskuissa huomioidaan myös muita sähköä kuluttavia ja säästäviä asioita, kuten sähköauton kulutus ja aurinkopaneelit. Kuvassa 41 esitetään päämetodin ohjelmakoodia, joka laskee kiinteän- ja pörssisähkön hinnan arvioidun kulutuksen perusteella.



Kuva 42. Arvioidun sähkönkulutuksen logiikkaa

Kuvassa 42 esitetään kaavio, joka kuvasta tilannetta, jossa käyttäjän dataa käytetään arvioidun sähkönkulutuksen ja hinnan laskemiseen. Päämetodin tehtävänä on laskea kiinteän- ja pörssisähkön arvioitu hinta kulutuksen perusteella. Muut metodit, kuten ”CalculateHousingConsumption” ovat vastuussa arvioidun vuosikulutuksen laskemisesta. Kaavion esimerkissä käytetään asumismuotoa ja asunnon neliöiden määrää arvioidun vuosikulutuksen laskemiseksi. Kaaviossa myös esitetään toisen metodin logiikkaa, joka arvioi läm-

mitysmuodon vuosikulutusta. Jokainen metodi, joka arvioi vuosittaista sähkönkulutusta tai sitä säästäviä tekijöitä palauttaa päämetodille kolme eri kulutus arvoa, arvioitu minimi, maksimi ja keskiarvo kulutus. Päämetodi lisää kaikki metodeilta saadut kulutukset yhteen ja laskee sen perusteella arvioivan hinnan kiinteälle- ja pörssisähkölle.

```
1 reference
private (decimal Min, decimal Average, decimal Max) CalculateHeatingConsumption(CombinedRequestDtoIn request)
{
    if (request.HouseType == HouseType.ApartmentHouse)
    {
        return (0, 0, 0);
    }

    return request.HeatingType switch
    {
        HeatingType.ElectricHeating => (700, 1000, 1300),
        HeatingType.DistrictHeating => (650, 800, 950),
        HeatingType.GeothermalHeating => (400, 500, 600),
        HeatingType.OilHeating => (400, 500, 600),
        _ => throw new ArgumentException("Invalid heating type")
    };
}
```

Kuva 43. Ohjelmakoodi, joka laskee arvioidun vuosikulutuksen lämmitysmuodon perusteella

Response body


```
{
  "TotalFixedPriceCost": 125.56,
  "TotalSpotPriceCost": 105.92,
  "TotalDirectiveConsumption": 1569.5,
  "EstimatedMinConsumption": 1151.5,
  "EstimatedMaxConsumption": 2093.5,
  "MinFixedPriceCost": 92.12,
  "MaxFixedPriceCost": 167.48,
  "MinSpotPriceCost": 77.71,
  "MaxSpotPriceCost": 141.28,
  "CalculationYears": "2023 - 2024",
  "CheaperOption": "Spot price",
  "CostDifference": 19.64,
  "AverageHourlySpotPrice": 6.66,
  "MonthlyData": [
    {
      "Month": 1,
      "Consumption": 188.34,
      "SpotPriceAverageOfMonth": 8.65,
      "FixedPriceAverageOfMonth": 8,
      "FixedPriceTotal": 15.07,
      "SpotPriceTotal": 16.30,
      "AverageConsumptionPerHour": 0.25
    },
```

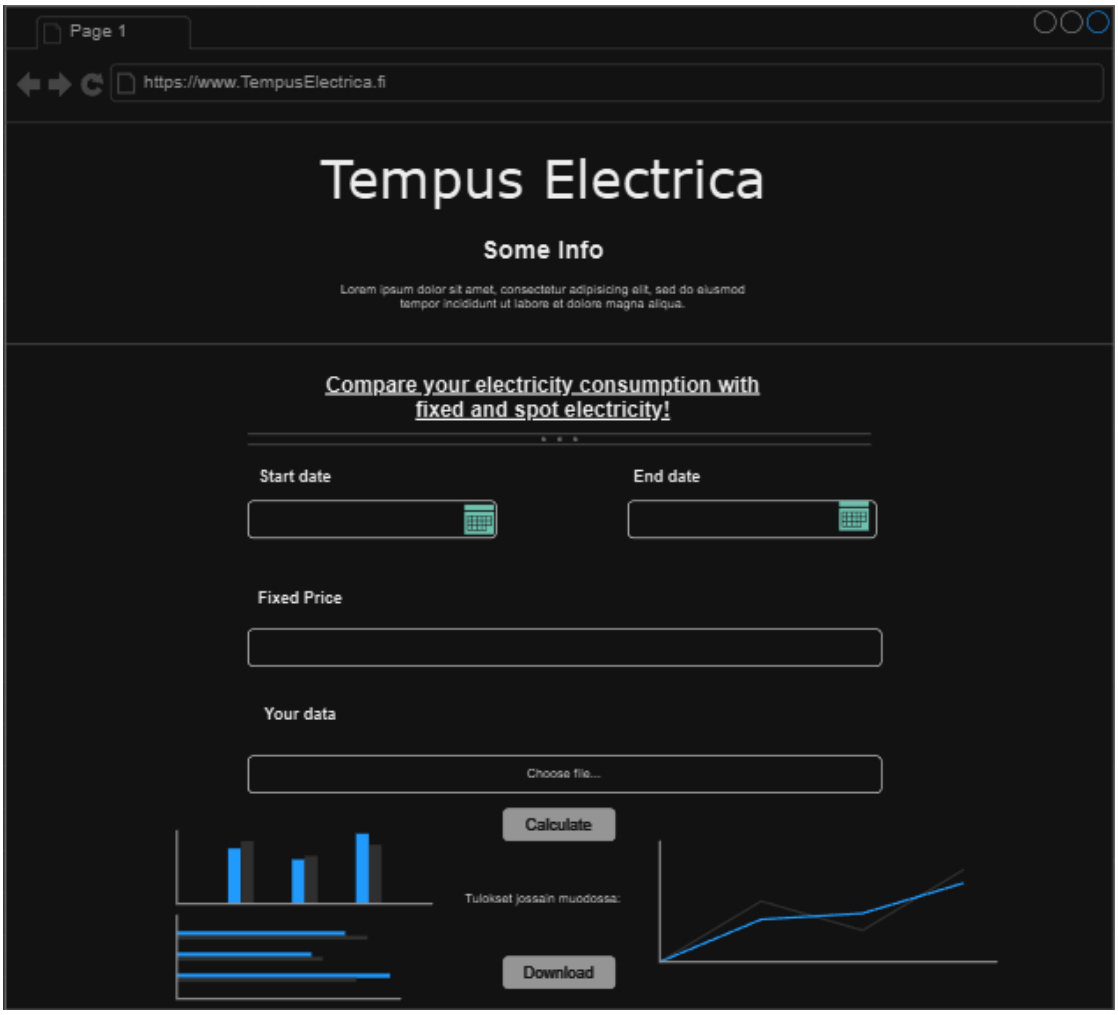

Kuva 44. Arviointityökalun palautettu data Swagger-näkymässä

Kuvassa 44 on esitetty kuvankaappaus arviointityökalun palautetusta datasta JSON-muodossa.

12.2 Front End toteutus

Sovelluksen front-end vastaa sovelluksen käyttöliittymän toiminnasta ja sen logiikasta. Front-end sisältää kaikki visuaaliset elementit, joita käyttäjä voi tarkastella ja käyttää. Front-end on rakennettu käyttäen React-frameworkia, sekä TypeScript- ja CSS-ohjelmointikieliä.

Sovelluksen front-end aloitettiin luomalla repositorio GitHubiin ja sen jälkeen luomalla React-projekti käyttäen TypeScript-mallia. Projektin luomisen jälkeen alkoi käyttöliittymän suunnittelu käyttäen Draw.io -sovellusta, sen tarkoituksena oli saada havainnollistava näkemys visuaalisesti sovelluksen käyttöliittymästä.

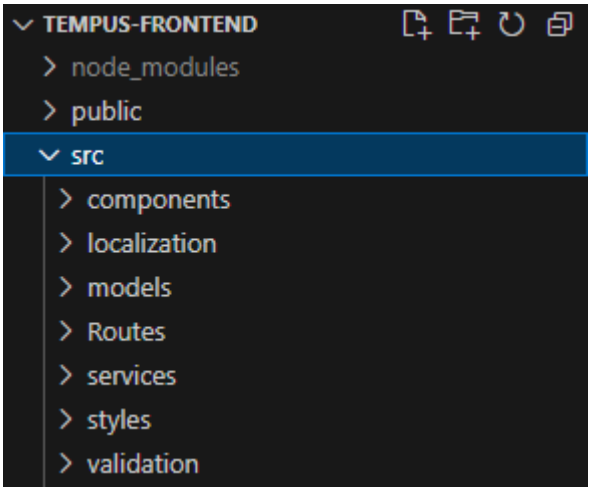


Kuva 45. Käyttöliittymäsuunnittelua sovelluksen mahdollisesta etusivusta

Kuvassa 45 on alussa toteutettua suunnitelmaa sovelluksen etusivusta, joka on toteutettu Draw.io-sovelluksella. Suunnitelmasta poiketen ohjelmakoodi lukee automaattisesti käyttäjän antamasta csv-tiedostosta alku- ja loppupäivämäärän, jonka perusteella laskelma voidaan suorittaa. Ideana oli kuitenkin se, että käyttäjä pystyy syöttämään kiinteän hinnan, johon vertaa omaa kulutustiedeostoaan, jonka jälkeen se esitetään käyttäjälle visuaalisesti pylväsdiagrammin muodossa.

Kuva 46. Käyttöliittymäsuunnittelua arviointityökalun käyttöliittymästä

Kuvassa 46 on suunnitelma siitä, kuinka arvioivan laskurin vaiheet tulisivat näkymään käyttöliittymässä. Lopputuloksena olikin melko samankaltainen vaiheittainen kysely käyttäjän tiedoista, joilla saadaan laskelma suoritettua.



Kuva 47. Front-end kansiorakenne

Kuvassa 47 on sovelluksen front-end kansiorakenne, jossa näkyy tärkeimmät kansiot käyttöliittymän kannalta. Komponenteista löytyy valintapainikkeet laskureiden välillä, arvioiva laskuri, kulutustiedostolaskuri, alatunniste, otsikko, sekä ohjeet. Front-end on jaettu selkeisiin kansioihin, joka helpottaa koodin hallintaa ja selkeyttää tiedostojen hakua.

Sovelluksen kieli on mahdollista valita suomeksi tai englanniksi, ja tämä on toteutettu ”Localization” -kansiossa, jossa on kirjoitettu suomen- ja englanninkieliset käännökset kaikista teksteistä. Käännökset on toteutettu i18next-kirjastolla, jonka tarjoama käännösfunktio ottaa vastaan käännösavaimen argumenttina ja palauttaa vastaavan käännetyn merkkijonon käännöstiedostosta.

Back-end-palveluun lähetettävät ja vastaanotettavat parametrit alustetaan ”models”-kansiossa, josta löytyy ”DataParams” ja ”FormDataParams”. Näitä malleja käytetään määrittelemään rajapinnat, jotka tyypittävät sovelluksen lähettämän ja vastaanottaman datan.

Sovelluksen komponenttien ulkoasun ja tyylien toteutus löytyy ”styles”-kansiosta, joka sisältää CSS-kooditiedostot. Tyylien käyttäminen ja toteutus oli melko vapaasti toteutettavissa, mutta käytimme Xamkin brändin päävärien keltaista #f5ba3c sekä Xamkin logoa. Tyyleissä on määritelty kaikkien painikkeiden, elementtien, tekstien sekä ikonien koot ja värit.

Arviointityökalun tietojen varmistus ja validointi tapahtuu ”validation”-kansiossa, jossa tarkistetaan, että käyttäjän syöttämät tiedot täyttävät vaaditut ehdot ennen niiden käsittelyä. Jos ehdot eivät täyty, käyttäjä näkee virheviestin, joka auttaa käyttäjää korjaamaan syötetyn virheen. Validointi on hyödyllinen tapa, koska tässä laskurissa on monta vaihetta, jossa tietoja kerätään.

Navigointi sovelluksen tietosuojaseloste- ja ohjesivulla toteutetaan ”Routes”-kansiossa, jossa määritetään mitä sivua näytetään. Tämän reitityksen avulla sovellus toimii ilman uudelleenlataamista, koska React päivittää ainoastaan näkymän, eikä koko sivua.

Kommunikointi back-endin kanssa

Front-end kommunikoi sovelluksen back-end-palvelun kanssa ja hakee sieltä dataa JSON-muodossa. Data haetaan ”FetchDataService”- ja ”FetchDirectiveData”-palveluilla, jotka esitetään Kuvan 47 ”services”-kansiosta. Nämä palvelut hakevat back-end-palvelun osoitteen env-tiedostoista, jota käytetään ympäristömuuttujien tallentamiseen.

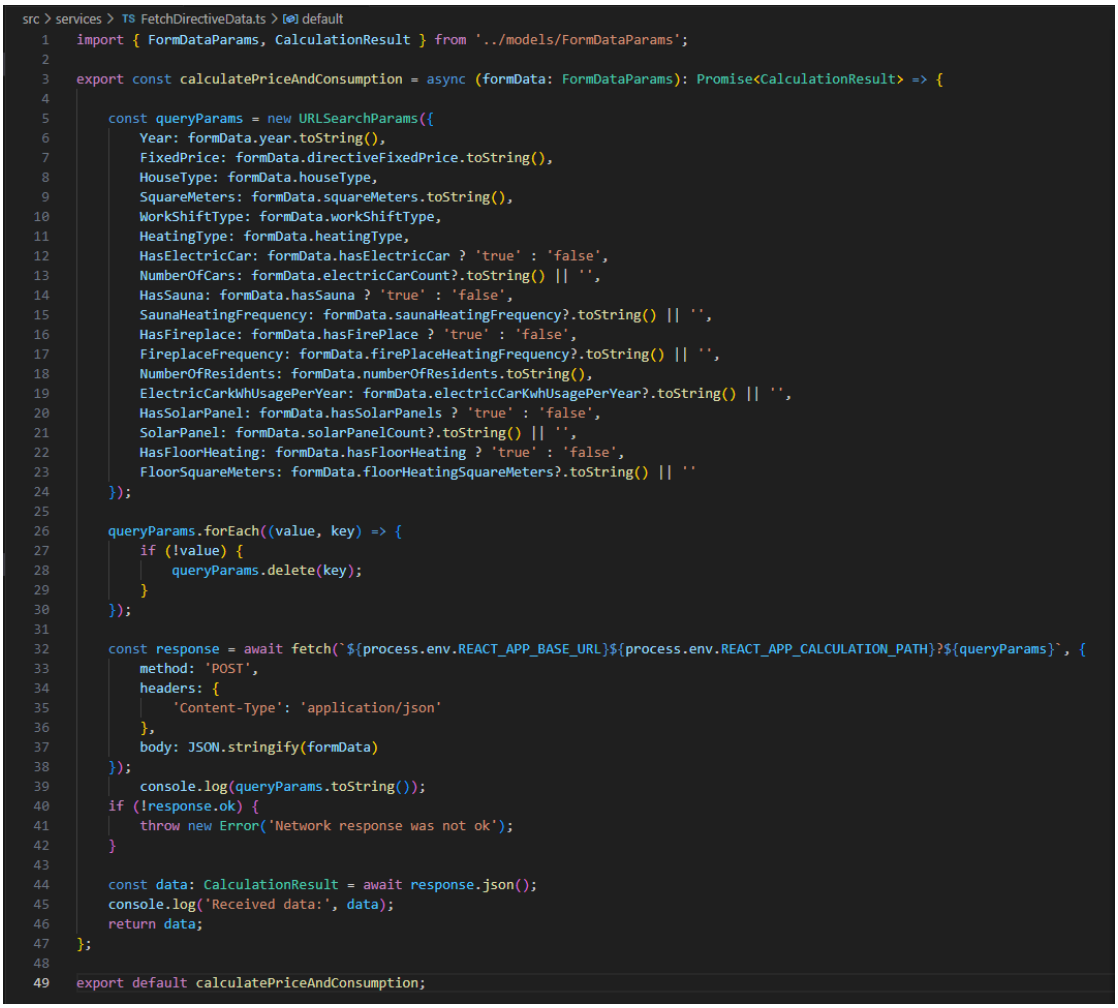
src > services > TS FetchDataService.ts > ...

```
1  import { DataParams } from '../models/DataParams';
2  import axios from 'axios';
3
4  export const FetchDataService = async (params: DataParams): Promise<DataParams> => {
5    const formData = new FormData();
6    if (params.csvFile) {
7      formData.append('file', params.csvFile);
8    }
9
10   try {
11     const response = await axios.post(
12       `${process.env.REACT_APP_BASE_URL}${process.env.REACT_APP_FINGRID_PATH}?fixedPrice=${params.fixedPrice}`,
13       formData,
14       {
15         headers: {
16           'Content-Type': 'multipart/form-data',
17         },
18       }
19     );
20     return response.data as DataParams;
21   } catch (error) {
22     throw error;
23   }
24 };
```

Kuva 48. "FetchDataService" -palvelun koodi

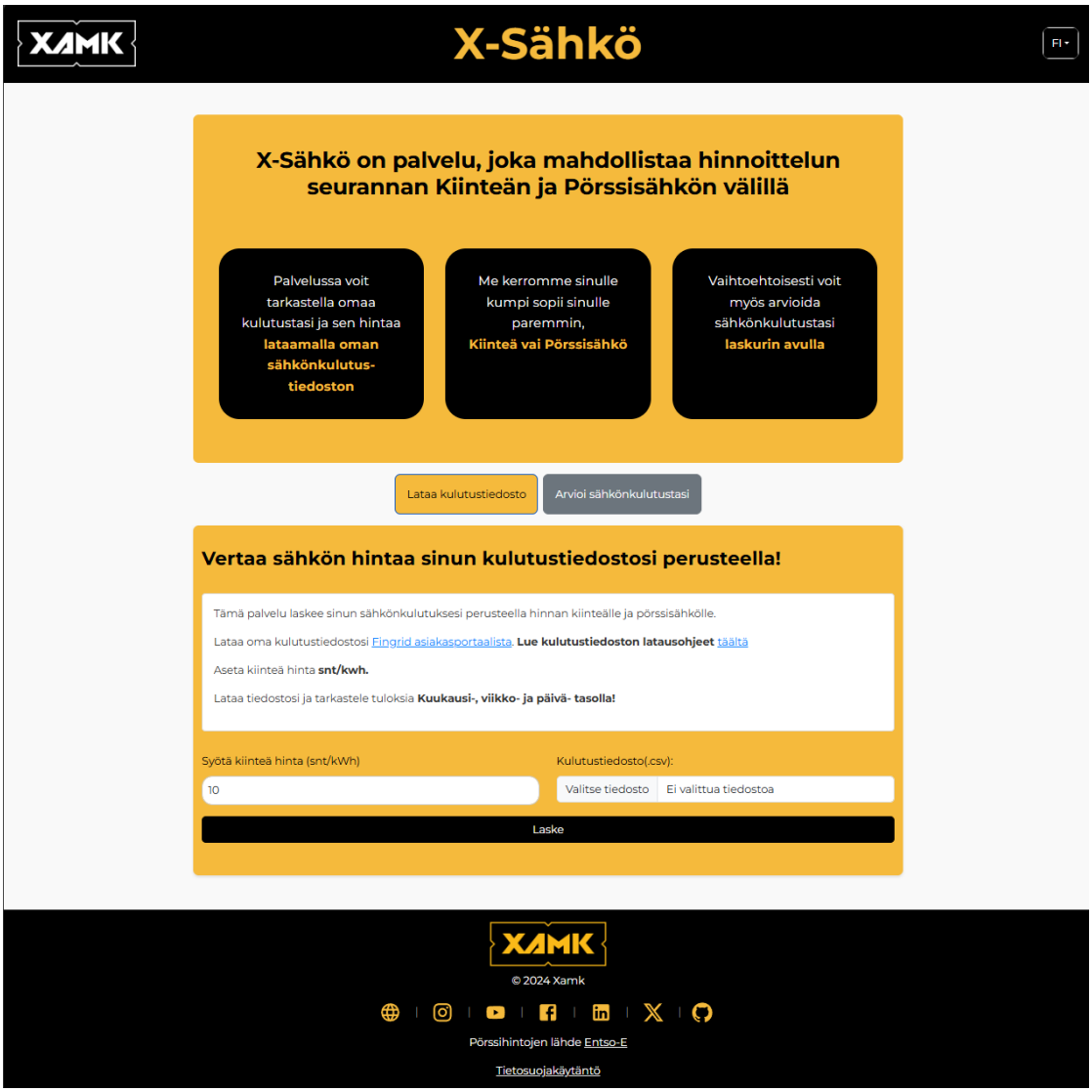
Kuvan 48 palvelussa lähetetään käyttäjän antama csv-tiedosto back-end-palvelulle, joka prosessoi tiedoston ja toteuttaa tarvittavat laskelmat. Tämän jälkeen tiedosto palautetaan takaisin ja se tallennetaan ”DataParams”-rajapin-

taan, jossa datalle annetaan oikeanlaiset tyypit kuten string, number, file. Tämän jälkeen haettua dataa käytetään ”FinGridCalculation”-komponentissa, jossa toteutetaan visuaalisesti käyttäjän kulutustiedoston laskelma.



Kuva 49. "FetchDirectiveData" -palvelun koodi

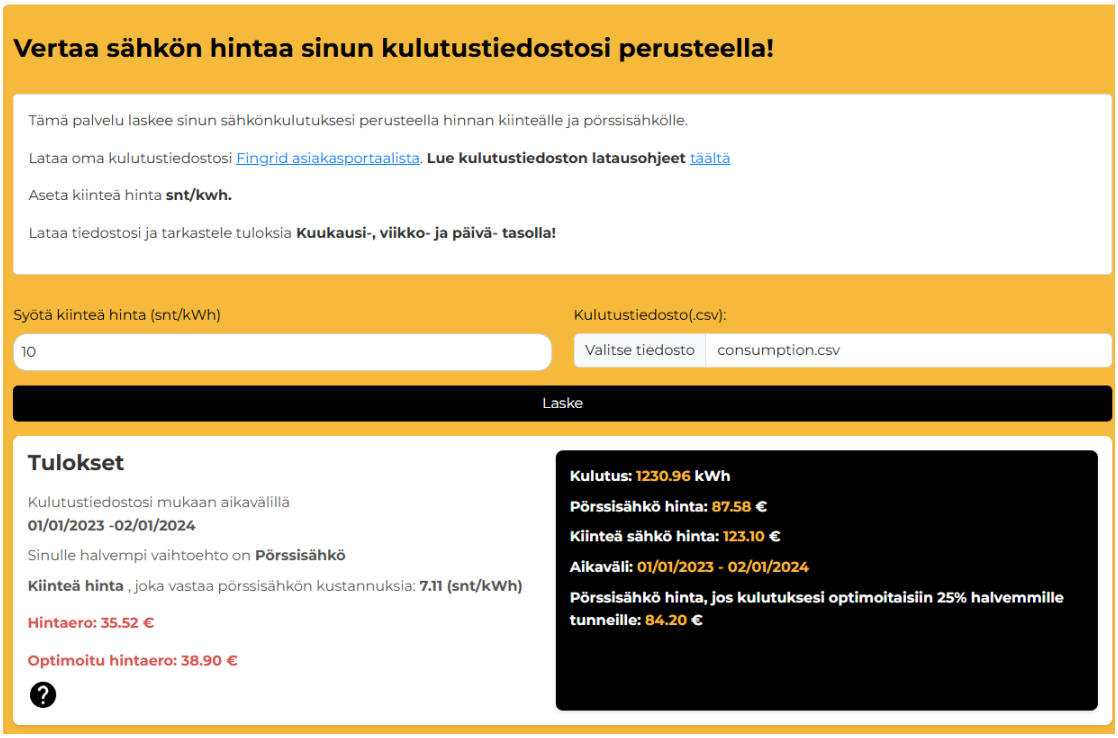
Kuvan 49 palvelu lähettää käyttäjän syöttämät parametrit kyselynä (Query) back-end-palveluun, joka toteuttaa laskentalogiikan. Parametrit ovat valmiiksi tyypitetty vastaamaan back-end-palvelun odottamaa muotoa ”FormDataParams”-tiedostossa, joka löytyy ”models”-kansiosta. Kun data on palautunut onnistuneesti back-end-palvelulta, sitä käytetään ”DirectiveCalculation”-komponentissa, jossa toteutetaan arvioiva laskuri visuaalisesti. Molemmat palvelut käyttävät Chart.js-kirjastoa luomaan pylväskaavioita, jotka ovat myös toteutettu näkymään mobiilinäkymällä siten, että kaaviot näkyvät käyttäjäystävällisesti.



Kuva 50. Sovelluksen etusivunäkymä

Kuvassa 50 näkyy tietokoneselaimella sovelluksen etusivu, jossa on valittuna kulutustiedostolaskuri. Etusivulla käyttäjälle kerrotaan infolaatikoissa mistä palvelussa on kyse ja mitä se sisältää. Oikeasta yläkulmasta käyttäjä voi valita kieleksi joko suomen tai englannin. Tässä laskurissa ohjeistetaan käyttäjä lataamaan oma kulutustiedosto Fingridin asiakasportaalista, jonka lataamiseen löytyy myös oma ohjesivusto viereisestä linkistä. Tämän jälkeen käyttäjä valit-

see kiinteän hinnan, jota haluaa verrata omaan kulutustiedostoon, lataa kulutustiedostonsa palveluun ja painaa laske -painiketta, jolloin tulokset latautuvat käyttäjälle.

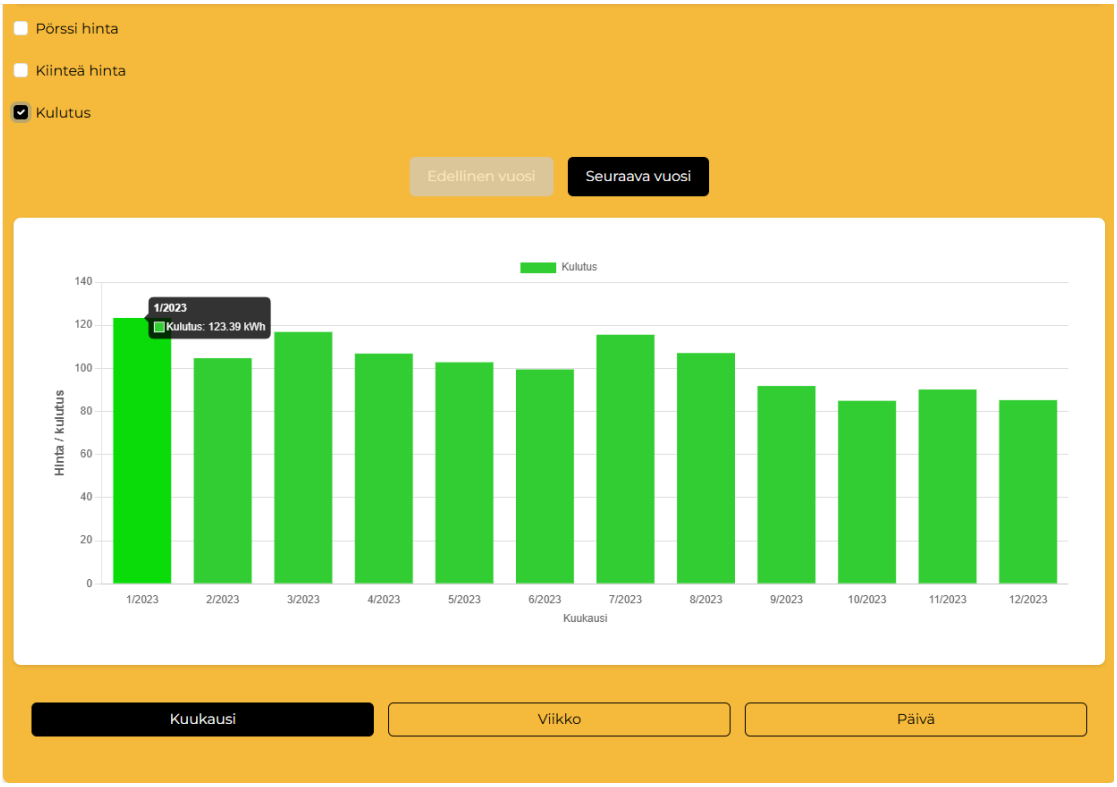


Kuva 51. Tulosnäkymä kulutustiedoston laskelmasta

Kuvassa 51 on tulokset laskelmasta, kun käyttäjä on valinnut kiinteäksi hinnaksi 10snt ja ladannut kulutustiedoston palveluun. Käyttäjä näkee välittömästi oleelliset tiedot laskelmasta, kuten halvemman sähkösopimus vaihtoehdon sekä esimerkiksi hintaeron halvemman ja kalliimman sopimuksen välillä.

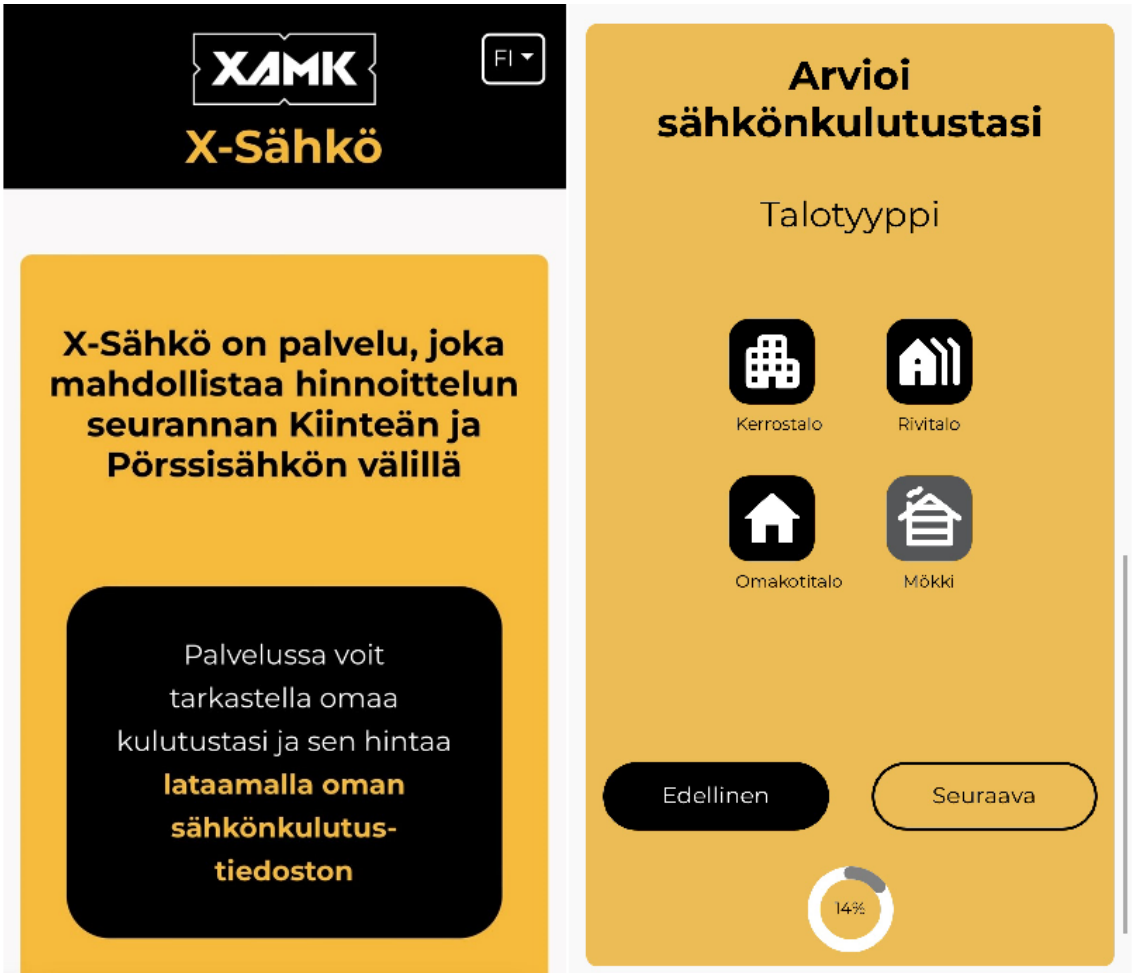
Kuva 52. Pylväsdiagrammi kulutustiedoston laskelman tulosnäkymästä

Kuvassa 52 on kuvan 51 laskelmasta esitetty visuaalinen pylväsdiagrammi, jossa näytetään oletusasetuksena käyttäjälle pörssisähkön hinta sinisenä, sekä kiinteän sähkön hinta punaisena. Klikkaamalla vasemmassa yläkulmassa olevia vaihtoehtoja käyttäjä voi valita näkymään joko pörssihinnan, kiinteän hinnan, kulutuksen, tai kaikki samanaikaisesti. Jos kulutustiedosto on suuremmalta aikaväliltä kuin yksi vuosi, käyttäjä voi navigoida seuraavaan vuoteen klikkaamalla ”seuraava vuosi” -painiketta. Kun halutaan tietää tietyn päivämäärän kulutus, voidaan klikata pylväsdiagrammin palkkia, jolloin käyttäjälle ilmestyy tämän palkin hintatieto. Tässä näkymässä voidaan myös tarkastella kulutustiedostoa tarkemmin ja navigoida kuukausi, viikko ja päivätasolla.



Kuva 53. Pylväsdiagrammi kulutustiedoston laskelman tulosnäkymästä, jossa on valittu näkymäksi pelkkä kulutuksen määrä

Kuva 53 on kuvien 52 ja 51 tulosten tarkastelua, jossa on valittuna ainoastaan kulutus. Samoin kuin kuvassa 52, myös kulutus voidaan nähdä valitsemalla ”kulutus” vasemmasta yläkulmasta ja sitten klikkaamalla palkkia.



Kuva 54. Kuvakaappaus etusivun mobiilinäkymästä ja arvioivan laskurin eräästä vaiheesta

Kuvassa 54 vasemmalla puolella näkyy sovelluksen etusivu mobiilinäkymästä, joka on rakennettu tarjoamaan käyttäjäystävällinen kokemus. Kuvan 54 oikealle puolella on yksi arviointityökalun vaiheista mobiilinäkymässä. Valintapainikkeita on pyritty helpottamaan tuomalla niitä pienemmälle alueelle silloin, kun käytössä on mobiililaite.

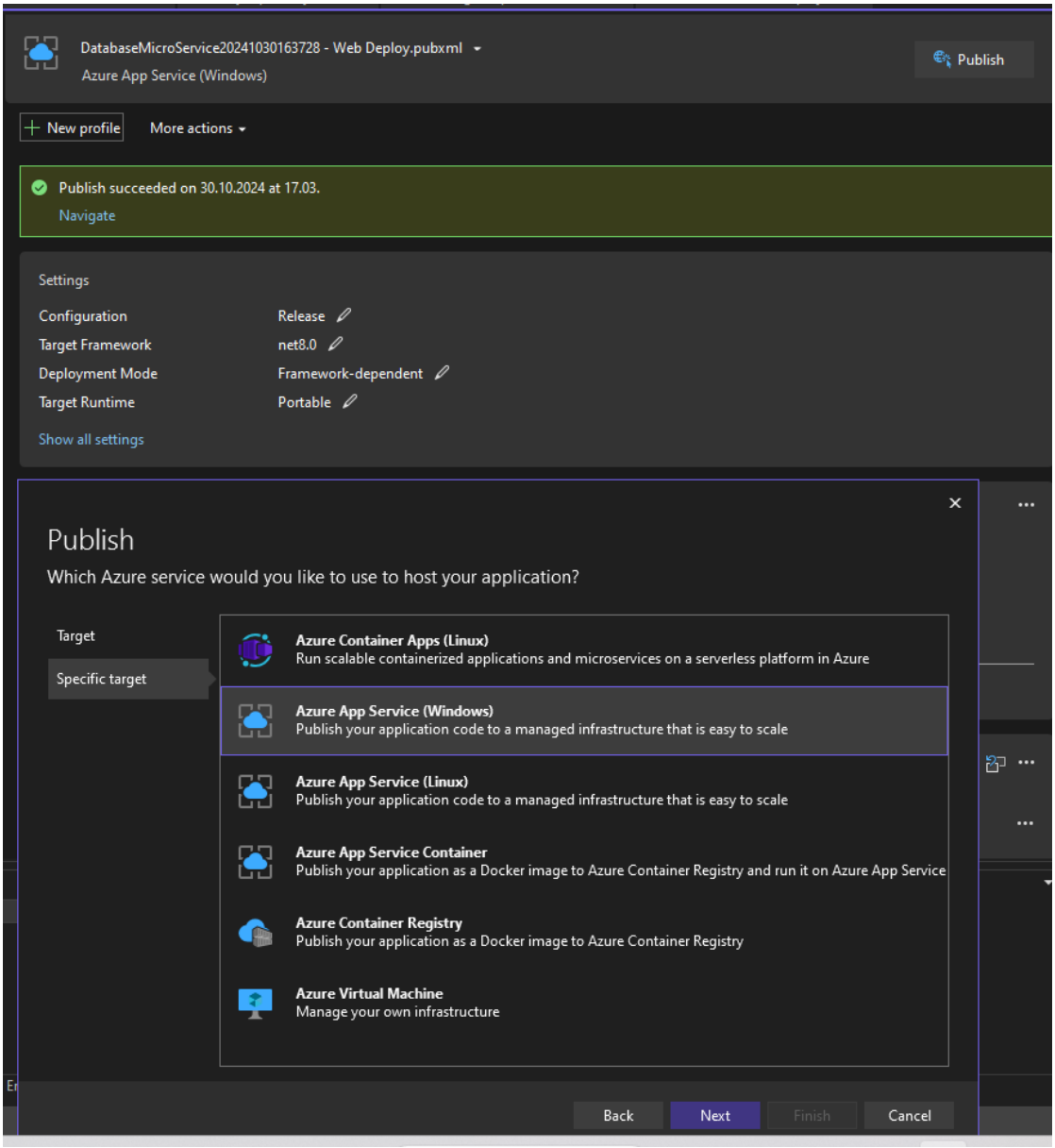
Arviointityökalun avulla voidaan arvioida omaa kulutusta käymällä läpi kyselyn tapaisen valintakaavion. Tässä työkalussa valitaan ensin vuosi, koska pörssisähkön hinta vaihtelee paljon ja sitten kiinteä hinta, johon verrata pörssihintaa. Sen jälkeen valitaan talotyyppi, asunnon pinta-ala, asukkaiden määrä, työmuoto ja lattialämmitys. Laskurissa voidaan talotyypin valinnan mukaan valita myös lämmitysmuoto. Myös sähköautoilu on otettu laskurissa huomioon ja käyttäjä voi valita taloudessa olevien sähköautojen määrän, sekä valita niiden kulutuksen mukaan laskelmaan. Sähkölämmitteinen sauna, takan lämmitys, sekä aurinkopaneelien määrä on myös otettu huomioon, koska ne ovat nykypäivänä oleellisia asioita.

Kuva 55. Mobiilinäkymä arviointityökalun tuloksista.

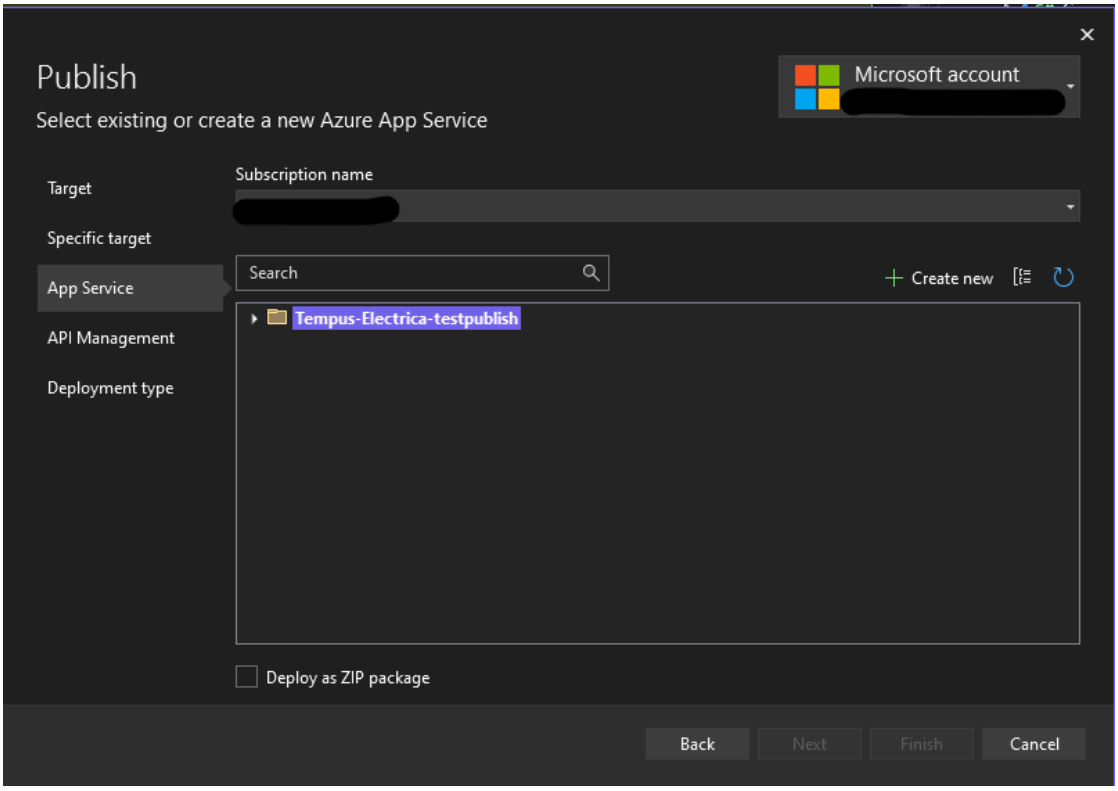
Kuvassa 55 näkyy arviointityökalun laskelman tulokset esitettynä mobiilinäkymässä. Vasemmalla puolella kuvassa näkyvät oleellisimmat tiedot laskutoimituksesta, kuten halvempi vaihtoehto, keskimääräinen hintaero, arvioitu kulutus, sekä kiinteän- ja pörssisähkön hinta. Pylväsdiagrammi on mobiilinäkymässä asetettu näyttämään kuusi kuukautta kerrallaan, jotta se on käyttäjäystävällisempi ja helpommin luettavissa. Samoin kuin kuvissa 52 ja 53, voidaan tarkastella kuukauden hintaa kerrallaan napauttamalla sinistä tai punaista palkkia. Painamalla ”seuraava”-painiketta, pylväsdiagrammi esittää loput vuoden laskelmasta. Arviointityökalun laskelma päästään helposti uudestaan toteuttamaan painamalla ”laske uudelleen”-painiketta.

12.3 Ohjelmiston julkaisun toteutus Azure-ympäristössä ja laadunvarmistus

X-Sähkö-palvelukokonaisuus on julkaistu Azure-pilvipalvelun ympärille. Front- ja back-end-palvelut on julkaistu Azure App Service -palveluun ja salaisuudet, kuten tietokannan yhteysmerkkijono on asetettu Azure Key Vault -palveluun. Sovelluksen GitHub-repositorio sisältää CI (Continuous Integration) -toteutuksen laadunvarmistuksen parantamiseksi ja CD (Continuous Delivery) -toteutuksen automaattisen julkaisun toteuttamiseksi.

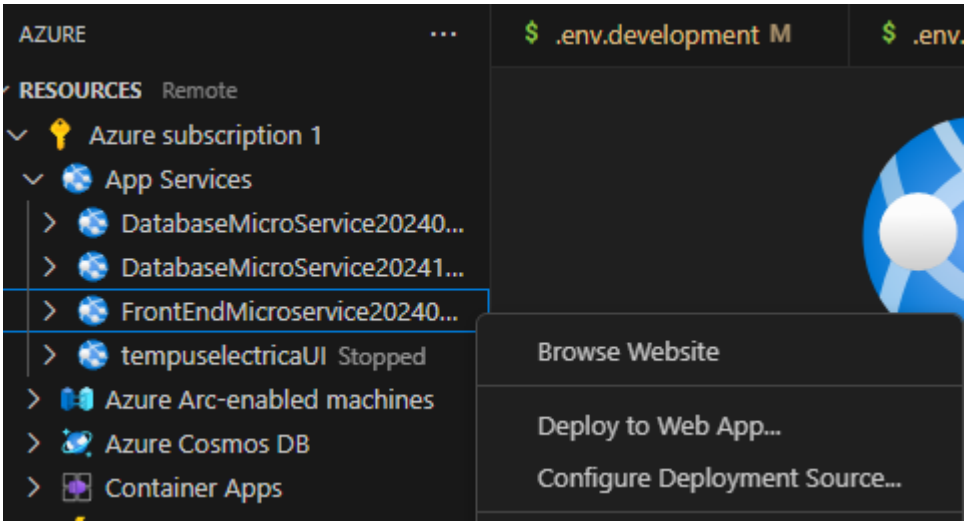


Kuva 56. Back-end-palvelun julkaisu Visual Studio ympäristössä



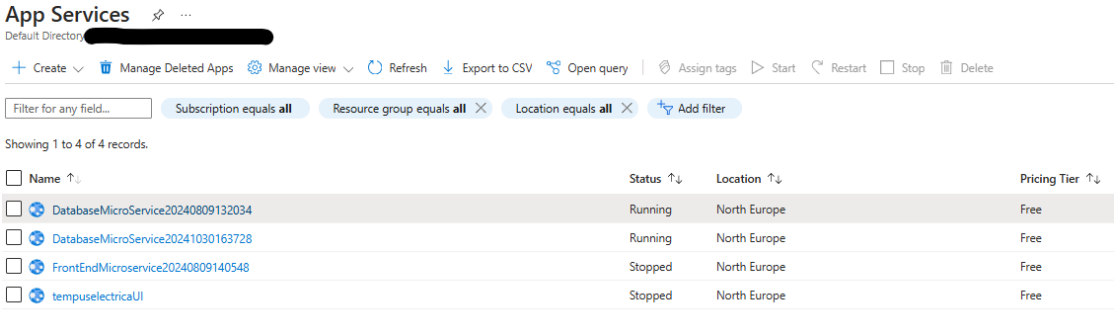
Kuva 57. Olemassa olevan App Service -palvelun valinta

Kuvassa 56 ja 57 esitetään sovelluksen julkaisun vaiheita Visual Studio -ympäristössä. Visual Studio mahdollistaa sovelluksen julkaisun Azure App Service -palveluun. Kuvassa 56 ”Publish” näkymä kuvastaa tilannetta, jossa määritellään, minkälaiseen palveluun sovellus halutaan julkaista. Kuvassa 57 on esillä julkaisun vaihe, jossa valitaan ennalta konfiguroitu App Service -palvelu, johon sovellus halutaan julkaista. Visual Studio myös mahdollistaa täysin uuden App Service -palvelun konfiguroinnin ja luomisen. Kuvan 56 ”Web deploy.pubxml” on valmis julkaisutiedosto, joka sisältää kaikki tarvittavat asetukset sovelluksen julkaisemiseksi. Sen avulla sovellus julkaistaan Azure App Service -palveluun.



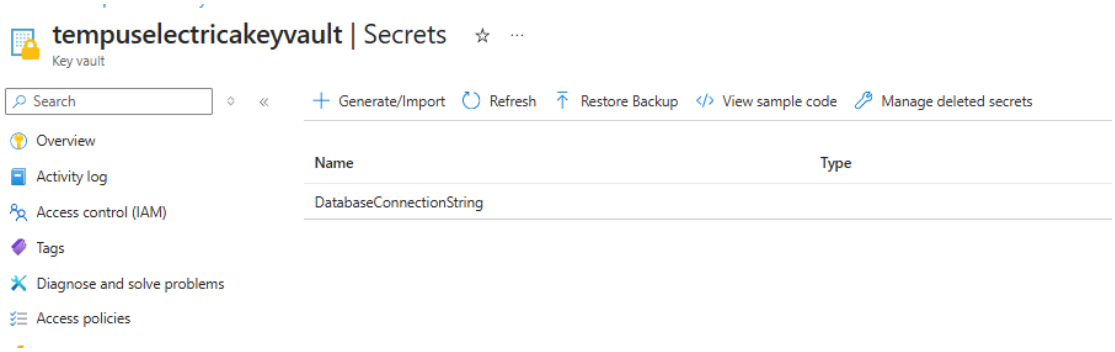
Kuva 58. Front-end-palvelun julkaisu Visual Studio Code -ohjelmassa

Front-end-palvelun julkaisu tapahtui Visual Studio Code -ympäristössä. Kuvassa 58 on esillä Azure App Service laajennus Visual Studio Code -ohjelmassa. Se mahdollistaa projektin julkaisemisen valitsemaansa App Service -palveluun. ”Deploy to Web App”-toiminto ottaa vastaan projektikansion, joka halutaan julkaista. Tähän vaiheeseen asetetaan ”build”-kansio projektista, joka tarkoittaa sovelluksen pakattua kansiota. React-projektista voidaan luoda ”build”-kansio ”npm run build”-komennolla Visual Studio Code -ympäristössä.



Kuva 59. X-Sähkö-palvelun kehitysvaiheen App Service -palvelut Azure Portal -näkymässä

Kuvassa 59 esitetään X-Sähkö-palvelun kehitysvaiheessa käytettyjä App Service -palveluita. Tässä vaiheessa sovellukset ovat siis yleisessä verkossa ja kaikkien saatavilla, mikäli ei ole asettanut pääsyrajoitteita näille palveluille.



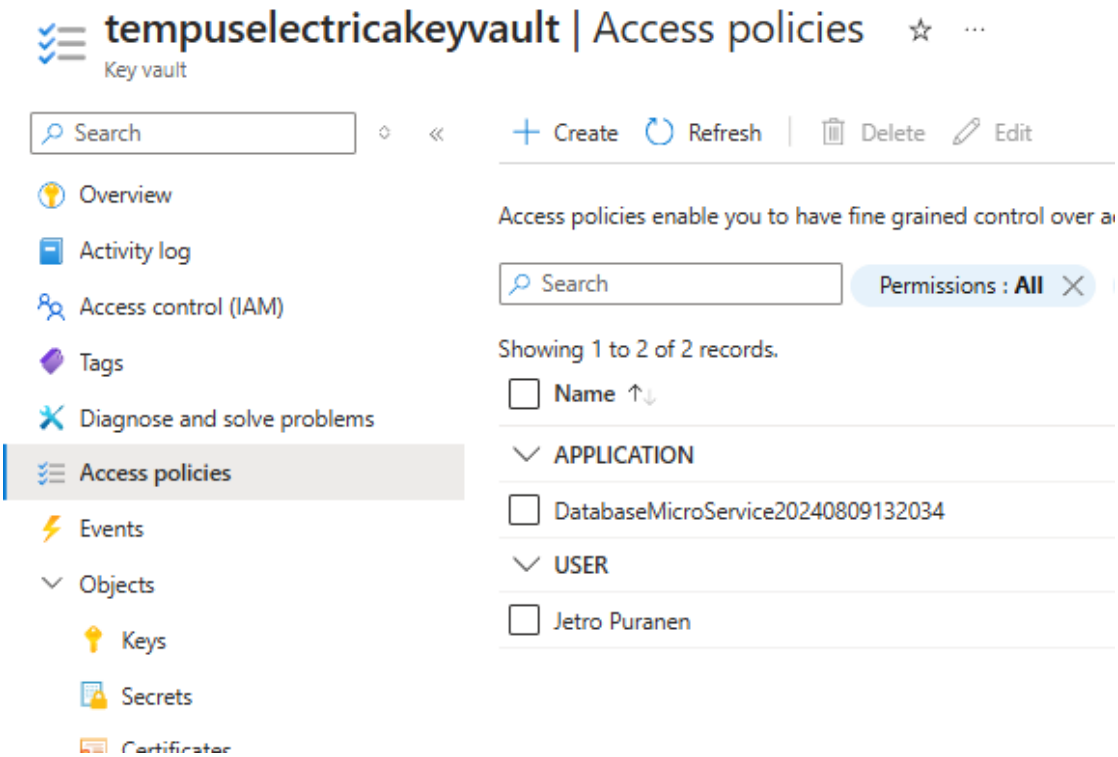
Kuva 60. Key Vault näkymä

Projektin tietokantaratkaisun yhteysmerkkijono on asetettu Azure Key Vault -palveluun. Kuvassa 60 on esillä Key Vault Secrets näkymä, jossa on salaisuuden nimi. Salaisuuden nimelle annettiin arvoksi yhteysmerkkijono ja tämä salaisuus voidaan hakea ohjelmakoodin avulla.

```
0 references
public KeyVaultSecretManager(IConfiguration configuration, ILogger<KeyVaultSecretManager> logger)
{
    _configuration = configuration;
    _logger = logger;

    var keyVaultEndpoint = _configuration["KeyVault:BaseUrl"];

    if (string.IsNullOrWhiteSpace(keyVaultEndpoint))
    {
        throw new ArgumentException("KeyVault:BaseUrl configuration is missing or empty.");
    }

    _secretClient = new SecretClient(new Uri(keyVaultEndpoint), new DefaultAzureCredential());
}

1 reference
public async Task<VaultSecret?> GetSecretAsync()
{
    const string secretName = "DatabaseConnectionString";

    if (_secretsCache.TryGetValue(secretName, out string? cachedSecret))
    {
        return new VaultSecret { DbConnectionString = cachedSecret };
    }

    var fetchedSecret = await FetchAndCacheSecretAsync(secretName);
    return new VaultSecret { DbConnectionString = fetchedSecret };
}
```

Kuva 61. Ohjelmakoodi, joka hakee yhteysmerkkijonon Key Vault palvelusta

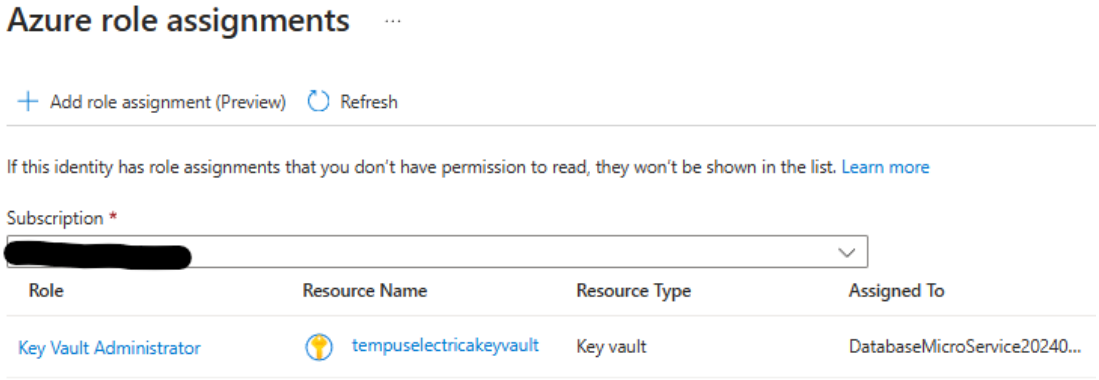
Jotta Key Vault -palvelusta voidaan hakea salaisuuksia, sille pitää konfiguroida käyttöoikeuskäytännöt. Ilman konfigurointia, Key Vault -palvelusta ei ole mahdollista hakea salaisuuksia.



Kuva 62. Key Vault -palvelun käyttöoikeuskäytännöt

Key Vault -ratkaisu käyttää pääsykäytäntöjä (Access Policies) todentamisen ja valtuuksien hallintaan. Kuvassa 62 esitetään Key Vault -palvelun käyttöoikeuskäytännöt. Tässä esimerkissä back-end-palvelun App Servicelle ja sovelluksen kehittäjälle on annettu oikeudet tähän Key Vault -ratkaisuun.

Back-end-palvelun App Service ratkaisulla ei ole oletuksena pääsyoikeuksia muihin resursseihin, kuten Key Vault resurssiin. Jotta tämä App Service saa käyttöoikeuden Key Vault resurssiin, sille pitää luoda oma identiteetti, jolla se voi todentaa itsensä Key Vault -palvelulle.



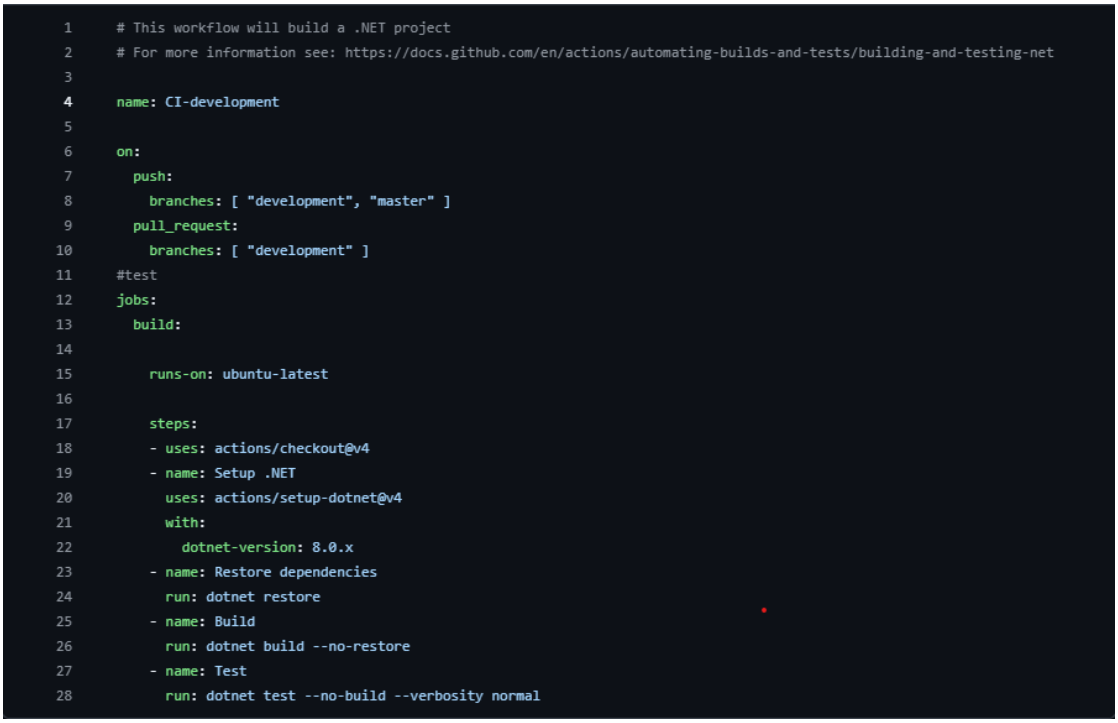
Kuva 63. App Service resurssille asetettu identiteetti

App Service tarvitsee identiteetin ja roolin, jotta se voi hallita ja käyttää Key Vault -palvelun resursseja. Kuvassa 63 back-end-palvelun App Service ratkaisulle on annettu identiteetti ja rooli ”Key Vault Administrator”. Tämä rooli mahdollistaa sen, että App Service voi käyttää ja hallita Key Vault -palvelun salaisuuksia. Ilman tätä konfiguraatiota Key Vault ei voi jakaa salaisuuksia App Service resurssille, koska se ei tiedä ”kuka” App Service on ja mihin se on valtuutettu.

```
[Fact]
0 references
public async Task CalculatePricesAsync_MissingCsvData_EmptyResult()
{
    // Arrange
    var csvFilePath = "nonexistentfile.csv";
    decimal fixedPrice = 0.20m;

    // Act
    var result = await _calculateFingridConsumptionPrice.CalculateTotalConsumptionPricesAsync(csvFilePath, fixedPrice);

    // Assert
    Assert.Equal(0, result.TotalSpotPrice);
    Assert.Equal(0, result.TotalFixedPrice);
    Assert.Equal("Error calculating data, or no data were found", result.CheaperOption);
    Assert.Equal(0, result.EquivalentFixedPrice);
}

[Fact]
0 references
public async Task CalculatePricesAsync_InvalidCsvFormat_EmptyResult()
{
    // Arrange
    var csvFilePath = "invaliddata_20230531_20230601.csv";
    decimal fixedPrice = 0.20m;

    // Mock the CSV reader service to throw CsvReadingException for invalid format
    _csvReaderServiceMock.Setup(service => service.ReadHourlyConsumptionAsync(csvFilePath))
        .ThrowsAsync(new CsvReadingException("Invalid CSV format.", null));

    // Act
    var result = await _calculateFingridConsumptionPrice.CalculateTotalConsumptionPricesAsync(csvFilePath, fixedPrice);

    // Assert
    Assert.Equal(0, result.TotalSpotPrice);
    Assert.Equal(0, result.TotalFixedPrice);
    Assert.Equal("Error calculating data, or no data were found", result.CheaperOption);
    Assert.Equal(0, result.EquivalentFixedPrice);
}
```

Kuva 64. Esimerkki yksikkötesteistä

Kuvassa 64 esitetään kaksi esimerkki testiä, joilla on testattu kulutustiedostolaskurin toimintaa. Nämä testit on toteutettu xUnit-kirjastoa käyttäen, ja niissä tarkastellaan, miten laskuri käyttäytyy tilanteessa, jossa käyttäjä on syöttänyt virheellisen tai puutteellisen kulutustiedoston. Testeillä varmistetaan se, ettei odottamattomia virheitä pääse tapahtumaan ja että virheet voidaan korjata ajoissa, ennen niiden päätymistä tuotantoon.



```yaml
# This workflow will build a .NET project
# For more information see: https://docs.github.com/en/actions/automating-builds-and-tests/building-and-testing-net

name: CI-development

on:
  push:
    branches: [ "development", "master" ]
  pull_request:
    branches: [ "development" ]
#test
jobs:
  build:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v4
    - name: Setup .NET
      uses: actions/setup-dotnet@v4
      with:
        dotnet-version: 8.0.x
    - name: Restore dependencies
      run: dotnet restore
    - name: Build
      run: dotnet build --no-restore
    - name: Test
      run: dotnet test --no-build --verbosity normal
```

Kuva 65. CI-putkilinja.

Kuvassa 65 on kuvankaappaus sovelluksen CI-putkistosta (CI-pipeline), joka on toteutettu GitHub Actions -palvelua käyttäen. Tämä putkisto ajaa kaikki yksikkötestit mukaan lukien kuvassa 64 esitetyt testit, kun sovelluksen kehitys- tai päähaaraan yritetään tehdä muutoksia. Tällä varmistetaan koodin laatu ja toimivuus, sekä virheiden havaitseminen varhaisessa vaiheessa. Jos CI-putkisto löytää virheen ja testit eivät mene läpi, silloin muutoksia ei integroida kehitys- tai päähaaraan ollenkaan.

```yaml
name: Production CD

on:
  workflow_run:
    workflows: ["CI-development"]
    types:
      - completed
    branches:
      - master
  push:
    branches:
      - master

env:
  AZURE_WEBAPP_NAME: tempuselectriciaprod
  DOTNET_VERSION: '8.0.x'
```

Kuva 66. CD-logiikka, joka suorittaa CI-toteutuksen

```yaml
- name: Deploy to Azure Web App
  id: deploy-to-webapp
  uses: azure/webapps-deploy@v2
  with:
    app-name: ${{ env.AZURE_WEBAPP_NAME }}
    publish-profile: ${{ secrets.AZURE_WEBAPP_PUBLISH_PROFILE }}
    package: ${{ github.workspace }}/myapp
```

Kuva 67. CD-logiikka, joka julkaisee sovelluksen App Service -palveluun

Sovelluksen CD-putkisto on myös rakennettu hyödyntäen Github Actions -palvelua. Sen tehtävä on rakentaa julkaistava sovellusversio ja julkaista se olemassa olevaan App Service -palveluun. CD-logiikka ajetaan aina, kun päähaaraan (master) lisätään koodimuutoksia. Kuvassa 66 on osa CD-logiikasta, joka ajaa ensin CI-toteutuksen ennen kuin se siirtyy rakennus vaiheeseen. Mikäli CI-toteutus ei mene onnistuneesti läpi, sovelluksen julkaistavaa versiota ei rakenneta ja julkaista. Kuvassa 67 esitetään CD-logiikkaa, joka lopulta julkaisee sovelluksen App Service -palveluun. Se käyttää Kuvassa 66 esiintyvän ”AZURE_WEBAPP_NAME” arvon, eli App Servicen nimeä ja Github Secrets -palveluun asetettua julkaisutiedostoa sovelluksen julkaisuun. CD-logiikka mahdollistaa nopean ja automattisen julkaisun App Service -palveluun.

13 PÄÄTÄNTÖ

Raportin kirjoittaminen on ollut pitkä prosessi, mutta onneksi aihe on alkanut tulla tutuksi sovelluksen rakentamisen aikana. Aihe-ehdotus tuli ohjaajalta, ja se olikin luonnollinen valinta syventävän harjoittelujakson jälkeen, jossa rakennettiin X-Sähkö-sovellus. Tämän raportin myötä asioita on jouduttu kertaamaan, ja siten on karttunut paljon enemmän tietoa ohjelmistokehityksestä ja sen prosesseista. Aiheeseen perehtyminen on varmasti ollut kannattavaa, koska näitä moderneja teknologioita vaaditaan alan työelämässä nykyään enemmän kuin koskaan ennen.

Tämän työn myötä on opittu yhä lisää sovelluksen rakentamisen työvaiheista, työkaluista sekä tekniikoista. Itse sovellukseen jäi vielä kehittämisideoita, esimerkiksi tulevaisuudessa käyttäjien kulutustietoja voisi hyödyntää arvioivan laskurin kehittämiseen kyselyiden avulla. Sovellus jättääkin tuleville opiskelijoille varmasti paljon hyödyllistä oppimateriaalia ja ohjelmistokehitystehtäviä, kuten koodin refaktorointia, testaamista ja ominaisuuksien lisäämistä.

Olemme tyytyväisiä lopputulokseen ja siihen, että työn kaikki vaiheet suunnittelusta toteutukseen, katselmointiin ja julkaisuun asti sujuivat suunnitellun aikataulun mukaisesti. Ohjelmointikokemus, tiedon kartuttaminen ja tekniikoiden opettelu on vahvistunut paljon käytännön tekemisen myötä.

LÄHTEET

About Azure Key Vault. 2024. Microsoft Learn. WWW-dokumentti. Päivitetty 12.9.2024. Saatavissa: https://learn.microsoft.com/en-us/azure/key-vault/general/overview [viitattu 12.11.2024].

About rulesets s.a. GitHub Docs. WWW-dokumentti. Saatavissa: https://docs.github.com/en/repositories/configuring-branches-and-merges-in-your-repository/managing-rulesets/about-rulesets [viitattu 12.11.2024].

ABTasty s.a. Continuous Integration and Delivery (CI/CD) Explained. WWW-dokumentti. Saatavissa: https://www.abtasty.com/resources/ci-cd/ [viitattu 13.11.2024].

Amazon s.a. What is Full Stack Development? WWW-dokumentti. Saatavissa: https://aws.amazon.com/what-is/full-stack-development [viitattu 29.10.2024].

Andtfolk, L. 2023. Mitä on mikropalveluarkkitehtuuri? Consultor. Blogi. Päivitetty 18.10.2023. Saatavissa: https://www.consultor.fi/mita-on-mikropalveluarkkitehtuuri/ [viitattu 15.10.2024].

Anwar, W. 2023. A Guide for Building Software with Clean Architecture. Ezzylearning. WWW-dokumentti. Päivitetty 3.1.2023. Saatavissa: https://www.ezzylearning.net/tutorial/a-guide-for-building-software-with-clean-architecture [viitattu 7.10.2024].

Application Insights overview. 2024. Microsoft Learn. WWW-dokumentti. Päivitetty 16.11.2024. Saatavissa: https://learn.microsoft.com/en-us/azure/azure-monitor/app/app-insights-overview [viitattu 13.11.2024].

App Service documentation s.a. Microsoft Learn. WWW-dokumentti. Saatavissa: https://learn.microsoft.com/en-us/azure/app-service/ [viitattu 11.11.2024].

App Service overview. 2024. Microsoft Learn. WWW-dokumentti. Päivitetty 13.9.2024. Saatavissa: https://learn.microsoft.com/en-us/azure/app-service/overview [viitattu 11.11.2024].

Azure Monitor overview. 2024. Microsoft Learn. WWW-dokumentti. Päivitetty 11.9.2024. Saatavissa: https://learn.microsoft.com/en-us/azure/azure-monitor/overview [viitattu 13.11.2024].

Azure Key Vault basic concepts. 2024. Microsoft Learn. WWW-dokumentti. Päivitetty 7.8.2024. Saatavissa: https://learn.microsoft.com/en-us/azure/key-vault/general/basic-concepts [viitattu 12.11.2024].

Bitloops s.a. Introduction to Clean Architecture. WWW-dokumentti. Saatavissa: https://bitloops.com/docs/bitloops-language/learning/software-architecture/clean-architecture [viitattu 11.10.2024].

Choudhary, M. 2024. What are Microservices? How does Microservices architecture work? Middleware. WWW-dokumentti. Päivitetty 25.4.2024. Saatavissa: https://middleware.io/blog/microservices-architecture/ [viitattu 15.10.2024].

Cloudflare s.a. What is the cloud. WWW-dokumentti. Saatavissa: https://www.cloudflare.com/learning/cloud/what-is-the-cloud/ [viitattu 2.11.2024].

Codeacademy s.a. Back-End Web Architecture. WWW-dokumentti. Saatavissa: https://www.codecademy.com/article/back-end-architecture [viitattu 29.10.2024].

Codeacedemy. 2021. What Is Front End. Blogi. Päivitetty 9.3.2021. Saatavissa: https://www.codecademy.com/resources/blog/what-is-front-end/ [viitattu 29.10.2024].

Code Maze. 2024. Health Checks in ASP.NET Core. WWW-dokumentti. Päivitetty 30.10.2024. Saatavissa: https://code-maze.com/health-checks-aspnetcore/ [viitattu 8.11.2024].

Condron, G. & Gutsch, J. 2024. Health Checks in ASP.NET Core. Microsoft. WWW-dokumentti. Päivitetty 23.7.2024. Saatavissa: https://learn.microsoft.com/en-us/aspnet/core/host-and-deploy/health-checks?view=aspnetcore-8.0 [viitattu 8.11.2024].

Coursera. 2024a. Front-End vs. Back-End Developer: Understanding the Differences. WWW-dokumentti. Päivitetty 1.2.2024. Saatavissa: https://www.coursera.org/articles/front-end-vs-back-end [viitattu 29.10.2024].

Coursera. 2024b. What Does a Back-End Developer Do? WWW-dokumentti. Päivitetty 3.4.2024. Saatavissa: https://www.coursera.org/articles/back-end-developer [viitattu 29.10.2024].

Deshpande, C. 2024. The Best Guide to Know What Is React. Simplilearn. WWW-dokumentti. Päivitetty 25.7.2024. Saatavissa: https://www.simplilearn.com/tutorials/reactjs-tutorial/what-is-reactjs [viitattu 17.10.2024].

Devmountain s.a. Git vs. Github: What's the Difference? WWW-dokumentti. Saatavissa: https://devmountain.com/blog/git-vs-github-whats-the-difference/ [viitattu 1.11.2024].

Diachenko, S. 2023. Git vs GitHub. GitKraken. Blogi. Päivitetty 11.1.2023. Saatavissa: https://www.gitkraken.com/blog/git-vs-github [viitattu 1.11.2024].

DuMez, K. 2024. Pull request approval permissions and rules in GitHub. Graphite. WWW-dokumentti. Päivitetty 1.8.2024. Saatavissa: https://graphite.dev/guides/pull-request-approval-permissions-rules-github [viitattu 12.11.2024].

Emizentech. 2024. React With TypeScript or JavaScript: Which Is Better? WWW-dokumentti. Päivitetty 9.10.2024. Saatavissa: https://www.emizentech.com/blog/react-with-typescript-or-javascript.html [viitattu 17.10.2024].

Entity Framework Core. 2024. Microsoft. WWW-dokumentti. Päivitetty 12.11.2024. Saatavissa: https://learn.microsoft.com/en-us/ef/core/ [viitattu: 8.10.2024].

Geeksforgeeks. 2022. Version Control Systems. WWW-dokumentti. Päivitetty 29.6.2022. Saatavissa: https://www.geeksforgeeks.org/version-control-systems/ [viitattu 22.10.2024].

GitHub. 2022. What is Version Control? WWW-dokumentti. Päivitetty 1.8.2022. Saatavissa: https://github.com/resources/articles/software-development/what-is-version-control [viitattu 22.10.2024].

GitLab s.a. What is CI/CD? WWW-dokumentti. Saatavissa: https://about.gitlab.com/topics/ci-cd/ [viitattu 13.11.2024].

Git Process s.a. NeITO Playbook. WWW-dokumentti. Saatavissa: https://playbook.neoito.com/source-control/gitprocess/ [viitattu 28.10.2024].

Habr. 2023. Raw SQL vs Entity Framework Core: Which is Right for Your Application? WWW-dokumentti. Päivitetty 24.4.2023. Saatavissa: https://habr.com/en/sandbox/190738/ [viitattu 31.10.2024].

Heller, M. 2024. What is GitHub? InfoWorld. WWW-dokumentti. Päivitetty 6.9.2024. Saatavissa: https://www.infoworld.com/article/2266566/what-is-github-more-than-git-version-control-in-the-cloud.html [viitattu 28.10.2024].

Herbert, D. 2023. What is React.js? HubSpot. Blogi. Päivitetty 13.11.2023. Saatavissa: https://blog.hubspot.com/website/react-js [viitattu 17.10.2024].

Hyvärinen, J. 2019. Mikropalveluarkkitehtuuri – milloin ja milloin ei? Twoday. WWW-dokumentti. Päivitetty 17.10.2019. Saatavissa: https://www.twoday.fi/blogi/blogi/mikropalveluarkkitehtuuri-milloin-ja-milloin-ei [viitattu 15.10.2024].

Jovanovic, M. 2023. Health Checks In ASP.NET Core For Monitoring Your Applications. MJ Tech. WWW-dokumentti. Päivitetty 29.4.2023. Saatavissa: https://www.milanjovanovic.tech/blog/health-checks-in-asp-net-core [viitattu 8.11.2024].

Jovanovic, M. 2022. How To Approach Clean Architecture Folder Structure. Päivitetty 24.9.2022. WWW-dokumentti. Saatavissa: https://www.milanjovanovic.tech/blog/clean-architecture-folder-structure [viitattu 11.10.2024].

Kagga, J. 2018. Understanding React Components. Medium. WWW-dokumentti. Päivitetty 14.5.2018. Saatavissa: https://medium.com/the-andela-way/understanding-react-components-37f841c1f3bb [viitattu 17.10.2024].

Kozon, T. 2023. Unveiling the Core Principles of Clean Architecture. Boring Owl. Blogi. Päivitetty 30.6.2023. Saatavissa: https://boringowl.io/en/blog/unveiling-the-core-principles-of-clean-architecture [viitattu 7.10.2024].

Lukasiewicz, P. 2021. .NET Core vs. .NET Framework – Which is a Better Choice for Your Next App? CSHARK. Blogi. Päivitetty 19.7.2021. Saatavissa: https://www.cshark.com/net-core-vs-net-framework-which-is-a-better-choice-for-your-next-app/ [viitattu 8.10.2024].

Martin, R. 2012. The Clean Architecture. Blogi. Päivitetty: 13.8.2012. Saatavissa: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html [viitattu 11.10.2024].

Melnyk, M. 2021. CI/CD Workflow: How to implement QA Automation. WWW-dokumentti. Päivitetty: 9.8.2021. Saatavissa: https://u-tor.com/topic/ci-cd-workflow-and-qa [viitattu 13.11.2024].

Microsoft. 2022. An introduction to NuGet. WWW-dokumentti. Päivitetty 10.12.2022. Saatavissa: https://learn.microsoft.com/en-us/nuget/what-is-nuget [viitattu 1.11.2024].

Microsoft. 2024. Elinkaaren UKK -.NET ja .NET Core. WWW-dokumentti. Saatavissa: https://learn.microsoft.com/fi-fi/lifecycle/faq/dotnet-core [viitattu 7.10.2024].

Microsoft s.a. What is Azure? WWW-dokumentti. Saatavissa: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-azure/ [viitattu 10.11.2024].

MongoDB s.a. Full Stack Development Explained. WWW-dokumentti. Saatavissa: https://www.mongodb.com/resources/basics/full-stack-development [viitattu 31.10.2024].

Muhammad, M. B. 2024. Understanding Clean Architecture. Medium. WWW-dokumentti. Päivitetty 24.1.2024. Saatavissa: https://levelup.gitconnected.com/clean-architecture-86c4f03e4771 [viitattu 17.10.2024].

Nobledesktop. 2023. What is Git and Why Should You Use it? WWW-dokumentti. Päivitetty 24.7.2023. Saatavissa: https://www.nobledesktop.com/learn/git/what-is-git [viitattu 28.10.2024].

Patel, P. 2023. Clean Architecture in .NET Core: A Complete Overview. Cmarix. Blogi. Päivitetty 21.4.2023. Saatavissa: https://www.cmarix.com/blog/clean-architecture-net-core/ [viitattu 7.10.2024].

Pokhrel, P. 2024. What are the differences between CSS, HTML and JavaScript? KeenToDesign. WWW-dokumentti. Päivitetty 13.8.2024. Saatavissa: https://www.keentodesign.com.au/difference-between-css-html-and-javascript/ [viitattu 29.10.2024].

Prasad, J. 2024. .NET Core vs .NET Framework: Which Beats the Other in App Development? WWW-dokumentti. Päivitetty 20.8.2024. Saatavissa: https://radixweb.com/blog/net-core-vs-net-framework [viitattu 7.10.2024].

Red Hat. 2023. What are microservices? WWW-dokumentti. Päivitetty 1.5.2023. Saatavissa: https://www.redhat.com/en/topics/microservices/what-are-microservices [viitattu 15.10.2024].

Sakovich, N. 2024. Difference Between .NET and C#. Sam-Solutions. Blogi. Päivitetty 8.4.2024. Saatavissa: https://www.sam-solutions.com/blog/dot-net-vs-c/ [viitattu 31.10.2024].

Schults, C. 2024. An introduction to unit testing in C#. Tricentis. WWW-dokumentti. Päivitetty 1.4.2024. Saatavissa: https://www.tricentis.com/learn/an-introduction-to-unit-testing-in-c-sharp [viitattu 4.11.2024].

Sharma, T. 2024. WHAT IS TYPESCRIPT IN REACT? Global Tech Council. WWW-dokumentti. Päivitetty 3.4.2024. Saatavissa: https://www.globaltechcouncil.org/react/typescript-in-react/ [viitattu 17.10.2024].

Singh, A. 2023. A Deep Dive into Clean Architecture and Solid Principles. Medium. WWW-dokumentti. Päivitetty 30.10.2023. Saatavissa: https://medium.com/@unaware_harry/a-deep-dive-into-clean-architecture-and-solid-principles-dcdcec5db48a [viitattu 7.10.2024].

Smith, A. 2023. What is .NET Core and Everything You Need to Know About It. Blogi. Päivitetty 4.1.2023. Saatavissa: https://www.hiddenbrains.com/blog/what-is-net-core-and-everything-you-need-to-know-about-it.html [viitattu 31.10.2024].

Staael, M. 2024. Clean Architecture. The Lost Engineer's Guide. Blogi. Päivitetty 8.10.2024. Saatavissa: https://staael.com/blog/clean-architecture [viitattu 17.10.2024].

Stackify. 2024. .NET Core vs. .NET Framework: How to Pick a .NET Runtime for an Application. WWW-dokumentti. Päivitetty 3.12.2024. Saatavissa: https://stackify.com/net-core-vs-net-framework/ [viitattu 7.10.2024].

Steele, C. 2020. What is A Full Stack Developer? How does It Compare To Back-End or Front-End Developer? Blogi. Päivitetty 1.6.2020. Saatavissa: https://blog.udemy.com/what-is-a-full-stack-developer-how-does-it-compare-to-back-end-or-front-end-developers/ [viitattu 31.10.2024].

Sirotka, A. 2022. What is Git and Why Use It? WWW-dokumentti. Päivitetty 31.1.2022. Saatavissa: https://flatlogic.com/blog/what-is-git-why-use-it/ [viitattu 28.10.2024].

Tozzi, C. 2024. When not to use microservices: 4 challenges to consider. TechTarget. WWW-dokumentti. Saatavissa: https://www.techtarget.com/searchapparchitecture/tip/When-not-to-use-microservices-Challenges-to-consider [viitattu: 15.10.2024].

Triggering a workflow s.a. GitHub Docs. WWW-dokumentti. Saatavissa: https://docs.github.com/en/actions/writing-workflows/choosing-when-your-workflow-runs/triggering-a-workflow [viitattu 12.11.2024].

Versionhallinta ja Git s.a. OS LevelUP Koodarit. WWW-dokumentti. Saatavissa: https://oslevelupkoodarit.github.io/materials/versionhallinta-ja-git.html [viitattu 22.10.2024].

Vinugayathri, X s.a. Why Should You Use xUnit? A Unit Testing Framework For .NET. Clarion Technologies. WWW-dokumentti. Saatavissa: https://www.clariontech.com/blog/why-should-you-use-xunit-a-unit-testing-framework-for-.net [viitattu 4.11.2024].