HAMK
Hämeen ammatti-
korkeakoulu

Tuotannon monitorointi -moduuli hybridisovellukseen mobiililaitteelle

Ammattikorkeakoulututkinnon opinnäytetyö
Tieto- ja viestintätekniikka, insinööri (AMK)
Kevät 2025
Ville-Valtteri Korppila

HAMK
Hämeen ammatti-
korkeakoulu

Tiivistelmä

Koulutus	Tieto- ja viestintätekniikka	
Tekijä	Ville-Valtteri Korppila	Vuosi 2025
Työn nimi	Tuotannon monitorointi -moduuli hybridisovellukseen mobiililaitteelle	
Ohjaaja	Toni Laitinen	

Työn tavoitteena oli toteuttaa mobiililaitteella toimivaan hybridisovellukseen uusi moduuli tuotannon monitorointiin. Valmiissa työssä tuli näkyä tuotantolaitteiden sen hetkiset työvaiheet eri valualustoilla, työvaiheisiin käytetty aika ja niiden aikataulutettu kesto sekä sen hetkinen valmistumisprosentti.

Työtä varten olemassa olevaan käyttöliittymään toteutettiin uusi moduuli Vue 3 -kehyksellä. Käyttöliittymään tarvittavat tiedot haettiin palvelimelta, joka toimii .NET-ympäristössä. Käyttöliittymän ja palvelimen väliseen tiedonsiirtoon käytettiin REST-rajapintaa, sekä SignalR-kirjaston tarjoamaa reaaliaikaista tiedonsiirtoa. Työn toteutusta demonstroidaan työkalujen ja kirjastojen dokumentaatiosta löytyvien esimerkkien avulla.

Lopputuloksena valmistui ensimmäinen versio Tuotannon monitorointi -moduulista, joka lukee tarvittavat tiedot reaaliaikaisesti palvelimelta. Moduuli jäi kuitenkin edelleen kehitysvaiheeseen. Jatkokehitystä varten moduulin vastaanottamat tiedot tulisi jäsennellä vielä järkevämpään muotoon ja laitteiden yksilölliset tiedot lisätä mukaan tiedonsiirtoon. Käyttöliittymän koodi tulisi jäsennellä pienemmiksi komponenteiksi helpottaakseen luettavuutta ja muokattavuutta.

Avainsanat	Hybridisovellus, käyttöliittymä, palvelin, SignalR, tiedonsiirto
Sivut	39 sivua

HAMK
Hämeen ammatti-
korkeakoulu

Abstract

DP	Information and Communication Technology	
Author	Ville-Valtteri Korppila	Year 2025
Subject	Production Monitoring Module for Hybrid Application in Mobile Devices	
Supervisor	Toni Laitinen	

This thesis aimed to produce a new module for monitoring production within a hybrid application which runs in mobile device. The finished module had to show production machines' current work phases in each casting bed, time used for work phases, scheduled duration and current progression percentage.

The new module was created with a Vue 3 framework to the existing user interface for the thesis. The required data for the user interface was fetched from the server, which operates in .NET environment. SignalR library's real-time data transfer capabilities and REST application interface were used for the data transfer between the user interface and the server. The process implemented in the thesis is demonstrated with the help of the examples from the documentations of the libraries and tools used.

The first version of the module, which reads the data from the server in real-time was produced as a result. However, the module remains in development. For further development, the data received by the module should be restructured into a better form and the machine-specific information should be added to the data transfer. The user interface code should also be refactored into smaller components to improve readability and modifiability.

Keywords	Hybrid application, user interface, server, SignalR, data transfer
Pages	39 pages

Sisällys

Kuvat

1 Johdanto

Työssä toteutetaan Tuotannon monitorointi -moduuli hybridisovellukseen mobiililaitteelle. Moduulin tarkoituksena on toimia lisäominaisuutena tuotannonohjausjärjestelmän yhteydessä. Moduuli toteutetaan osaksi olemassa olevaa mobiilikäyttöliittymää. Monitorointi-näkymässä tulee näkyä laitteiden sen hetkiset työvaiheet, työvaiheen valmistumisprosentti, työhön käytetty aika ja työhön arvioitu aika. Näkymän tiedot tuodaan palvelimelta hyödyntäen reaaliaikaista tiedonsiirtoa. Palvelin mukailee CLEAN-arkkitehtuuria ja lukee tiedot tietokannasta hyödyntäen olio-relaatiokartoitusta eli ORM-tekniikkaa. Koska työ toteutetaan hybridisovelluksena, toteutus muistuttaa selainsovellusta ja vastaava toteutus voitaisiinkin toteuttaa myös täysin selainpohjaisena sovelluksena.

Luvussa 2 esitellään työssä käytössä olevat työkalut sekä menetelmät ja referoidaan niihin liittyvää dokumentaatiota. Työ on toteutettu käyttöliittymän osalta Quasar- ja Vue 3 -kehyksillä. Palvelin on toteutettu .NET ympäristössä Entity-kehystä käyttäen. Luvussa 3 käydään läpi vaihtoehtoisia työkaluja toteuttaa vastaavanlainen totetus sekä avataan menetelmiä yleisesti. Luvussa 4 käydään läpi työn vaiheita ja sitä, kuinka luvun 2 työkaluja ja menetelmiä on käytetty toteutuksessa. Työn vaiheita käydään läpi dokumentaatiosta löytyvien esimerkkikoodien avulla. Luvussa 5 tehdään yhteenveto työn toteutuksesta ja käydään läpi jatkokehitystarpeita.

2 Työssä käytettävät työkalut ja menetelmät

Tässä osiossa esitellään työssä käytettyjä työkaluja ja menetelmiä ja referoidaan niihin liittyvää dokumentaatiota ja kirjallisuutta. Osiossa esitellään työn toteutuksen kannalta tärkeimmät kehykset, kirjastot, sekä menetelmät. Työ toteutetaan osana isompaa kokonaisuutta, jonka vuoksi valitut menetelmät ja työkalut ovat jo määritelty aiemmin.

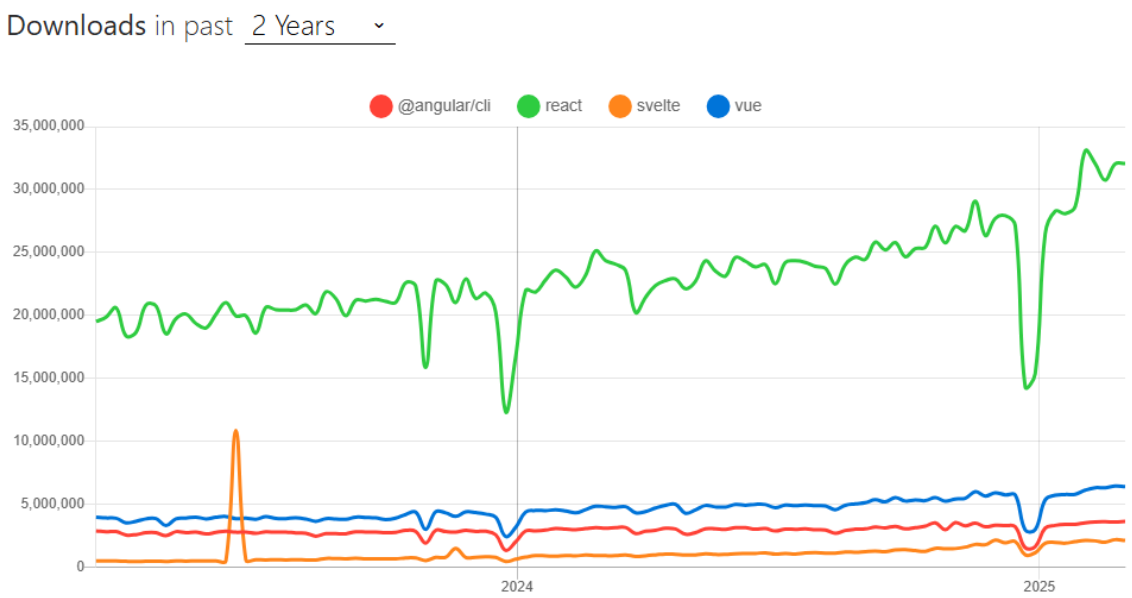
Käyttöliittymänä toimivaa Vue-kehystä pidetään nopeasti omaksuttavana ja Quasar-kehyksen tarjoamilla valmiilla komponenteilla voidaan luoda hyvin nopeasti uusia sovelluksia. Hybridisovellus mahdollistaa mobiililaitteen ominaisuuksien käytön, kuten tehdas käyttöön soveltuvan tabletin tarjoaman viivakoodin skannauksen samalla pitäen kehityksen nopeana.

Palvelinpuolen kehitysympäristönä toimiva .NET-ympäristö toimii saumattomasti Windows-palvelimien kanssa. Ympäristö tarjoaa hyvän skaalautuvuuden ja tietoturvaominaisuudet. Nämä ovat vaikuttaneet ympäristön valinnassa.

2.1 Vue 3

Vue on alun perin Evan You:n vuonna 2014 luoma JavaScript -kehys, jonka kehityksestä nykyään vastaa täyspäiväisesti työskentelevä tiimi sekä joukko vapaaehtoisia (VueJS, n.d.-a). Vue helpottaa selaimella toimivien sovellusten luomista tarjoten suuren määrän ominaisuuksia, joita yleisesti käytetään selainpohjaisten sovellusten kehityksessä (VueJS, n.d.-b). Vue perustuu avoimeen lähdekoodiin, ja se on yhteisövetoinen projekti. Vue:lla on yli 1,5 miljoonaa käyttäjää ympäri maailmaa ja se on tuotantokäytössä muun muassa NASAlla, Applella, Googlella ja Microsoftilla (VueJS, n.d.-a). Kuvasta 1 nähdään, että Vue on npm -paketinhallintaohjelman latausten perusteella tällä hetkellä toiseksi suosituin JavaScript-kehys. Facebookin kehittämä React on latausten perusteella selkeästi suosituin JavaScript-kehys.

Kuva 1. JavaScript-kehyksien lataukset npm-paketinhallintaohjelman kautta viimeisen kahden vuoden aikana (Npm trends, n.d.).



Vuen uusin merkittävä versio on Vue 3 ja se on ollut helmikuun seitsemännestä päivästä vuonna 2022 lähtien yleisesti käytössä. Vue 2 on joulukuussa vuonna 2023 tullut elinkaaren loppuun eli siihen ei enää päivitetä uusia ominaisuuksia. Vue 2 perustui OptionsAPI-tyyliin kirjoittaa koodia. Vue 3:n yhteydessä alun perin esiteltiin uusi CompositionAPI-tyyli. CompositionAPI kuitenkin myöhemmin lisättiin myös Vue 2:een saataville versioon 2.7. (VueJS, n.d.-c)

OptionsAPI on perinteinen tapa luoda .vue -tiedostoja. Siinä tiedoston rakenne on hyvin tarkkaan määritelty, ja kaikki komponentin elinkaaren tietyllä hetkellä tapahtuvat funktiot tulee kirjoittaa samaan osioon koodia. CompositionAPI mahdollistaa vapaamman tiedostorakenteen, jolloin koodi voidaan organisoida kontekstin perusteella. OptionsAPI:n tiukemmat raamit saattavat auttaa pienemmissä komponenteissa pitämään tiedostorakenteen paremmin organisoituna kuin CompositionAPI:ssa, mutta Composition API:lla tehdessä tulisikin nojata JavaScriptin yleisiin hyväksi todettuihin tapoihin organisoida koodia. Tyylien tiedostorakenne-eroja on havainnollistettu alla olevassa kuvassa 2, jossa sama komponentti on kirjoitettu OptionsAPI:lla ja CompositionAPI:lla. Kuvasta voidaan todeta, että CompositionAPI:lla voidaan paremmin organisoida samaan aihealueeseen liittyvät koodit, jolloin tiedostosta tulee selkeämmin luettava. Tämä korostuu etenkin komponenttien koon kasvaessa. (VueJS, n.d.-d)

Kuva 2. Sama komponentti kirjoitettuna OptionsAPI:lla ja CompositionAPI:lla värikoodattuna aihealueittain (VueJS, n.d.-d).

Tiedostorakenteen lisäksi CompositionAPI mahdollistaa selkeämmän logiikan uudelleenkäytön. OptionsAPI:ssa logiikan uudelleenkäytössä hyödynnetään ominaisuutta nimeltä *mixins*. Mixinsin haittapuolena on, että jos tiedostossa on useampi mixins käytössä, ominaisuuden alkuperäistä sijaintia on vaikea jäljittää. Tämän lisäksi erilliset mixins-tiedostot voivat keskenään aiheuttaa törmäyksiä nimiavaruudessa. Näidenkin osalta organisointihaasteet korostuvat sovelluksen kasvaessa. CompositionAPI tarjoaa tähän vaihtoehtona ominaisuuden nimeltä *composables*. Composables mahdollistaa tilallisen

logiikan pilkkomisen omaksi tiedostoksi ja sen ottamisen uudelleenkäyttöön eripuolilla koodia (VueJS, n.d.-e). On hyvä huomioida, että tilattoman logiikan pilkkomiseen ei ole tarvetta käyttää composables-ominaisuutta. Se voidaan toteuttaa normaalina JavaScript-tiedostona.

Toistaiseksi VueJS aikoo ylläpitää sekä OptionsAPI:a, että CompositionAPI:a. OptionsAPI on edelleen laajasti käytössä ja sopii hyvin pienempiin projekteihin. CompositionAPI:n edut tulevat paremmin esille vasta suuremmissa projekteissa. Kyseisiä tyylejä voidaan myös käyttää samanaikaisesti projektissa, mutta se ei ole suotavaa sillä se kuormittaa koodin ylläpitoa. Vue 2 -migraatiosta Vue 3:een voi olla kuitenkin tarve ylläpitää hetkellisesti molempia tyylejä. (VueJS, n.d.-e).

2.2 Quasar

Quasar on Vue-pohjainen kehys. Se perustuu Vuen tavoin avoimeen lähdekoodiin ja sen on alun perin kehittänyt Razvan Stoenescu vuonna 2015. Stoenesculla oli tarve työkalulle, joka mahdollistaisi yhden lähdekoodin kaikille alustoille. Sopivaa työkalua ei ollut saatavilla, joten Stoenescu päätti kehittää tätä varten Quasarin. Quasarin tavoitteena on tarjota kattava joukko valmiita yleisesti käytettyjä komponentteja sovelluskehitykseen ja auttaa kehittäjiä luomaan nopeasti sovelluksia, joita voidaan ajaa selaimessa, työpöydällä ja mobiililaitteilla. (Stoenescu, 2019)

Quasar mahdollistaa mobiilisovellusten kehityksen hyödyntäen Cordovaa ja Capacitoria. Tässä työssä käytetään Quasarin tarjoamaa Cordovaa, josta kerrotaan seuraavassa kappaleessa tarkemmin. Capacitoria ei käsitellä tässä työssä. (Quasar, n.d.-a)

Cordova on mobiilisovelluskehityskehys, jonka on alun perin luonut Nitobi. Adobe Systems osti Nitobi:n vuonna 2011 ja myöhemmin julkaisi ohjelmasta avoimen lähdekoodin version nimellä Apache Cordova. (Quasar n.d.-b)

Cordova mahdollistaa mobiilisovellusten kehityksen yleisillä selainpohjaisten sovellusten kehityskielillä, kuten HTML5:llä, CSS3:lla ja JavaScriptilla. Cordovan avulla voidaan luoda mobiililaitteelle selainpohjainen sovellus, joka kuitenkin ulkoisesti näyttää kuin se olisi natiivisovellus. Se ei kuitenkaan ole täysin natiivisovellus mutta ei myöskään täysin selainpohjainen sovellus vaan näiden yhdistelmä. Cordova tarjoaa myös joukon

laajenuksia, joita käyttämällä pystytään sovelluksessa hyödyntämään mobiililaitteen sensoreita ja ominaisuuksia, kuten kameraa tai akun varausta. (Cordova, n.d.)

2.3 .NET

.NET on Microsoftin ylläpitämä avoimen lähdekoodin kehitysympäristö, jolla voidaan luoda laajasti erilaisia sovelluksia. Ympäristö tukee useampaa eri kieltä, joista C# on yleisin.

Ympäristö voidaan jakaa kuuteen komponenttiin: Suorituspalveluun (*runtime*), kirjastoihin (*libraries*), kääntäjään (*compiler*), ohjelmistokehityspakettiin (*sdk*) ja sovelluskehitysympäristöihin (*app stacks*). Suorituspalvelu ajaa sovelluksen koodia, kirjastot tarjoavat joukon apufunktioita ja ominaisuuksia, kääntäjä kääntää kirjoitetun koodin ajettavaan muotoon, ohjelmistokehityspaketti mahdollistaa sovellusten rakentamisen sekä tarkkailun, ja sovelluskehitysympäristöt mahdollistavat sovellusten kirjoittamisen.

.NET-ympäristöstä on olemassa eri variaatioita, joista kaksi yleisintä ovat .NET Framework ja .NET Core. .NET Core on näistä uudempi ja tarjoaa mahdollisuuden käyttää eri alustoja. (Microsoft, 2024-a)

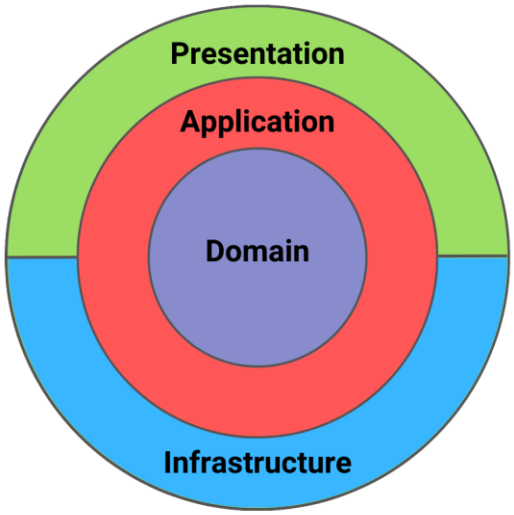
ASP.NET Core on .NET-ympäristön sovelluskehitysympäristö, jolla voidaan toteuttaa fullstack-selainpohjaisia sovelluksia, perinteisiä REST-rajapintoja, reaaliaikaisen tiedonsiirron palveluita sekä tekoälysovelluksia (Microsoft, n.d-b). Se on hyvä yrityssovelluksiin, sillä se skaalautuu isoihin projekteihin ja tarjoaa kattavat tietoturvaominaisuudet. Se on kuitenkin melko raskas ympäristö pienempiin projekteihin. (WAC, 2024)

Entity-kehys on Microsoftin luoma kehys ASP.NET-ympäristöön, jolla voidaan hakea dataa relaatiotietokannasta ja muuntaa se oliopohjaiseksi. Kehyksen ansiosta kehittäjän ei tarvitse tehdä suoria SQL-kielellä tehtyjä hakuja kantaan, vaan voi tehdä kyselyt C#-kielellä. (Microsoft 2022-c)

2.4 CLEAN-arkkitehtuuri

CLEAN-arkkitehtuuri on tapa rakentaa ohjelmistoa. Sen ideana on pitää ydinkoodi muuttumattomana, jolloin voidaan tarvittaessa helposti vaihtaa käyttöliittymä tai tietokanta toiseen järjestelmään. Tämän saavuttamiseksi tulisi välttää suoria riippuvuuksia ohjelmiston eri osien välillä (Khan, n.d.). CLEAN-arkkitehtuuri voidaan jakaa kuvassa 3 esitettyihin neljään kerrokseen: *Domain*, *Application*, *Infrastructure* ja *Presentation*.

Kuva 3. CLEAN-arkkitehtuurin rakenne havainnollistettuna (Jovanovic, 2022).



Domain sisältää yleisesti tärkeimmän sisällön, kuten Entity-luokkien määrittelyn. *Application* sisältää liiketoimintalogiikan, kuten palvelut (*services*) ja rajapinnat (*interfaces*) palvelujen sekä kontrollerien (*controllers*) välillä. *Infrastructure* sisältää migraation kantaan, kuten kontekstien (*contexts*) konfiguroinnin, ja *Presentation* sisältää rajapinnan käyttäjille tai muihin sovelluksiin, kuten kontrollerit. (Khan, n.d.)

2.5 Rajapinnat

Ohjelmoinnissa rajapinnalla tarkoitetaan sovelluksien välistä tiedonsiirtorajapintaa. Siitä käytetään yleisesti lyhennettää API, joka tulee englannin sanoista *Application Programming Interface*. Tässä työssä API:lla viitataan yleisesti palvelimen tarjoamaan rajapintaan, jonka avulla käyttöliittymällä voidaan hakea ja muokata tietokannan tietoja.

Palvelimen ja käyttöliittymän väliseen rajapintaan käytetään sekä REST:iä, että SignalR:ä. REST:llä haetaan tiedot, kun käyttöliittymä avataan ensimmäisen kerran. Tämän jälkeen palvelin lähettää reaaliaikaista tietoa käyttöliittymälle SignalR:n avulla.Tämän lisäksi on olemassa tietokannan ja palvelimen välinen rajapinta. Näiden väliseen tiedonsiirtoon käytetään Entity-kehystä.

2.5.1 REST API

REST API tunnetaan myös nimellä RESTful API. Se on tapa toteuttaa rajapinta sovellusten välillä, ja sen on alun perin määritellyt Dr. Roy Fielding vuonna 2000. REST-rajapinnan perusperiaate on mahdollistaa toisen sovelluksen tai palvelun ottaa HTTP-pyynnöin yhteyttä palvelimeen, joka tekee pyynnön perusteella halutun toimenpiteen. Yleisimmät toimenpiteet ovat resurssin luonti, lukeminen, päivittäminen tai poisto ja näistä käytetäänkin lyhennettä CRUD, joka tulee toimenpiteiden englanninkielisistä käännöksistä *Create*, *Read*, *Update* ja *Delete*. REST-rajapinnan tila on muuttumaton, eli jokaisen pyynnön tulisi sisältää aina palvelimen kannalta tarpeelliset tiedot riippumatta siitä, onko pyyntöjä lähetetty aiemmin tai ei. (IBM, n.d.)

REST-rajapinta on yleinen tiedonsiirtomenetelmä ja toimii hyvin moniin eri sovelluksiin. Se ei kuitenkaan toimi kovin hyvin reaaliaikaiseen tiedonsiirtoon, sillä sovelluksen tai palvelun tulee lähettää HTTP-pyyntö palvelimelle, joka palauttaa tiedon. Jotta REST-rajapinnalla päästäisiin lähelle reaaliaikaista tiedonsiirtoa, tulisi sovelluksen tai palvelun lähettää HTTP-pyyntö tietyin intervallein. Tämä aiheuttaa kuitenkin turhaa kuormitusta verkkoon.

2.5.2 SignalR

SignalR on Microsoftin luoma kirjasto ASP.NET-ympäristöön. Se mahdollistaa reaaliaikaisen tiedonsiirron palvelimen ja sovelluksen tai palvelun välillä. REST-rajapinnan tiedonsiirrosta poiketen sovelluksen ei tarvitse ottaa ensin yhteyttä palvelimeen, vaan palvelin tarkkailee tietokantaan tulevia muutoksia ja tietojen päivittyessä palvelin lähettää automaattisesti tiedon SignalR-keskuksen (*hub*) kautta sovellukselle eli vastaanottajalle (*client*). SignalR tukee kolmea eri tekniikkaa tiedonsiirtoon, joista se valitsee tilanteen mukaan parhaiten soveltuvan. Nämä kolme tekniikkaa ovat WebSockets-protokolla, palvelimelta lähetettävät tapahtumat (*Server-Sent Events*) ja pitkät HTTP-kutsut (*long polling*). (Microsoft, 2024-d)

SignalR vaatii keskuksen määrittelyn palvelimelle, sekä vastaanottajan määrittelyn käyttöliittymälle. Palvelin puolella SignalR on sisällytetty ASP.NET Coreen, joten sitä ei tarvitse erikseen asentaa. Vastaanottajaksi on tarjolla valmiita kirjastoja JavaScriptille, .NET:lle, Javalle ja Swiftille. (Microsoft, 2024-d)

3 Vaihtoehtoiset tavat

Tässä osiossa käydään lyhyesti läpi vaihtoehtoisia tapoja toteuttaa vastaavanlainen työ. Työ voidaan jakaa kolmeen fyysiseen osaan, mobiilikäyttöliittymään, palvelimeen ja tietokantaan sekä näiden osien välisiin rajapintoihin. Tietokantaa ei varsinaisesti käsitellä tässä työssä muuten kuin palvelimen ja tietokannan välisen rajapinnan kautta.

Osiossa käydään läpi mobiilikehitykseen liittyvät eri tavat toteuttaa sovelluksia, sekä mahdolliset kielet ja kehykset, joilla eri toteutukset voidaan luoda. Palvelinpuolella esitellään yleiset ohjelmointikielet palvelimien tekemiseen. Tämän lisäksi osiossa käydään läpi palvelimen ja käyttöliittymän väliseen rajapintaan tarjolla olevia protokollia ja menetelmiä.

3.1 Mobiilikehitys

Mobiilisovelluksia voidaan luoda joko natiivi-, hybridi- tai web-sovelluksena. Natiivisovellus tarkoittaa, että sovellus on toteutettu käyttäen mobiililaitteen käyttöjärjestelmän sallimaa kieltä.

Natiivisovellus tarjoaa mahdollisuuden parempaan optimointiin kuin muut sovellustyypit. Tämä näkyy etenkin massiivisissa sovelluksissa. Sen haasteena on kuitenkin hitaampi kehitys, sillä sovellus toteutetaan alustakohtaisesti ja eri alustalle siirrettäessä sovellus joudutaan luomaan käytännössä uudestaan.

Web-sovellus on puhtaasti selaimessa toimiva sovellus. Sen etuna on, että se on helposti saavutettavissa kaikilta laitteilta, jolloin kehittäjän ei tarvitse luoda kuin yksi toteutus. Se ei myöskään vaadi erillistä latausta laitteelle. Selainpohjainen sovellus ei pysty kuitenkaan hyödyntämään alustan tarjoamia ominaisuuksia.

Hybridisovellus on näiden välimalli. Se on käytännössä web-sovellus, joka on upotettu natiiviin sovellukseen. Siinä yhdistyy web-sovelluksen tavoin mahdollisuus luoda vain yksi toteutus sekä natiivisovelluksen tavoin mahdollisuus käyttää alustan tarjoamia ominaisuuksia, kuten akun varausta tai viivakoodin skannausta. (Amazon, n.d.)

3.1.1 Natiivisovellus

Googlen Android-sovelluksia on yleisesti luotu natiivina Java-, Kotlin- ja C++ -kielillä. Android tarjoaa sovellusten luomiseen Android Studio -kehitysympäristön ja suosittelee käyttämään ensisijaisesti Kotlinia. Kotlin muistuttaa Javaa, mutta tarjoaa parannuksia, kuten helpotusta vakaan koodin kirjoitukseen. (Android, n.d.)

Vastaavasti IOS-sovelluksiin on käytetty muun muassa Swift-, Objective-C- ja C++ -kieltä (Apple, n.d.). Apple tarjoaa sovellusten luomiseen Xcode-kehitysympäristön. On huomioitava, että Xcode on saatavilla vain Mac App Storesta ja vaatii Macin, eikä ole saatavilla Windowsille tai Linuxille. Xcode suosittelee käyttämään Swiftiä, joka on yleinen ohjelmointikieli Applen laitteille.

3.1.2 Hybridisovellus

Hybridisovelluksen luomiseen voidaan käyttää muun muassa Apache Cordova-, Ionic- ja React Native -kehyksiä. React Native ja Ionic ovat nykyään Apache Cordovaan verrattuna selkeästi suositumpia. Luvussa 2.2.1 on käsitelty työssä käytettävää Apache Cordovaa tarkemmin.

Ionic tarjoaa mahdollisuuden luoda hybridisovelluksia yleisimmillä JavaScript-pohjaisilla kehyksillä kuten Angularilla, Reactilla ja Vuella (Ionic, n.d. -a). Se on avoimen lähdekoodin projekti, jonka on luonut samanniminen yritys. Yrityksen on alun perin perustanut Max Lynch ja Ben Sperry vuonna 2012 (Ionic, n.d. -b). React Native on vuonna 2015 Metan julkaisema kehys, jolla voi luoda hybridisovelluksia React-kehyksellä (React Native, n-d.).

3.1.3 Natiivit usean alustan sovellukset

Sovelluksilta kaivataan nykyään samaan aikaan entistä parempaa suorituskykyä, mutta samalla niitä täytyisi pystyä kehittämään aiempaa nopeammin. Tämän vuoksi viime vuosina on alkanut yleistymään natiivien usean alustan sovellusten kehitykseen soveltuvat kehykset.

Flutter on Googlen vuonna 2017 julkaisema kehys, jolla voidaan luoda natiiveja sovelluksia käyttäen olio-ohjelmointikieltä nimeltä Dart. Toisin kuin hybridisovellukset, Flutter kääntää koodin suoraan natiiviksi. Tämä mahdollistaa yhden toteutuksen useammalle alustalle mutta kuitenkin pitäen sovelluksen täysin natiivina poistaen JavaScriptin aiheuttamat suoritusvaikeudet (Flutter, n.d.). Vastaavasti Kotlin Multiplatform on Kotlin-säätiön vuonna 2023 julkaisema kehys natiivien sovellusten luomiseen useammalle alustalle Kotlin-kielellä. Myös Microsoft on luonut usean alustan natiivisovellusten luomiseen kehyksen nimeltä .NET MAUI. Kyseinen kehys on julkaistu vuonna 2022 ja se käyttää C# -kieltä. (JetBrains, n.d.)

3.2 Palvelimet

Palvelimet voivat toimia käyttöliittymän ja tietokannan välissä. Näiden tehtävänä on jäsennellä tietokannan tiedot käyttöliittymän tarvitsemaan muotoon ja suojata tietokantaa mahdollisilta haitallisilta kyselyiltä. Myös mahdolliset salassa pidettävät kaavat ja logiikka tulee sijoittaa turvallisuussyistä palvelimelle eikä käyttöliittymän lähdekoodiin. Palvelimet toteutetaan yleensä olio-ohjelmointikielillä, kuten Javalla, C#:lla, Pythonilla, Rustilla, Golangilla tai PHP:lla. Lisäksi on olemassa JavaScript pohjainen NodeJS-kehys. (WAC, 2024)

Näistä kielistä selkeästi suosituimpia ovat Java, C# ja Python. Lisäksi NodeJS-kehys on kasvattanut suosiotaan. Rust ja Golang ovat selkeästi vähemmän suosittuja. PHP on edelleen laajalti käytössä, mutta sitä voidaan pitää jossain määrin vanhentuneena tekniikkana.

3.2.1 Tällä hetkellä suositut kielet

Työssä palvelin on toteutettu C# -kielellä ASP.NET Core -kehyksellä, josta on kerrottu tarkemmin luvussa 2.3.1. Se voidaan laskea mukaan suosittuihin olio-ohjelmointikieliin. Etenkin ASP.NET Core -kehyksen myötä .NET-ympäristöön tullut mahdollisuus käyttää muita alustoja kuin Windowsia on lisännyt C# -kielen suosiota.

Java on hyvin suosittu ohjelmointikieli yrityssovelluksissa. Se tarjoaa hyvät tietoturvaominaisuudet ja vakaan alustan isojen projektien hallitsemiseen. Javasta on olemassa useita eri kehyksiä, joista tunnetuin on Spring-kehys. Java on kielenä kuitenkin melko monimutkainen, ja sillä kehittäminen on hidasta. Se on kuitenkin hyvin vakaa ja toimii kaikissa ympäristöissä. (WAC, 2024)

Python on yleisesti hyvin suosittu ja laajasti käytössä oleva ohjelmointikieli. Sen suosiota selittää erityisesti tekoäly- ja tiedonkäsittelysovellukset, joita tehdään ensisijaisesti Pythonilla. Palvelinpuolella Pythonia voidaan hyödyntää Django- ja Flask-kehyksillä. Flask soveltuu vain pieniin toteutuksiin. Django on raskaampi kehys, mutta tarjoaa valmiina monia tarpeellisia työkaluja, kuten käyttäjienhallinnan ja ylläpidon. Django soveltuu niin yrityssovelluksiin kuin myös pienempiin projekteihin, sillä Python on kielenä helppo oppia. Pythonin heikkoutena on kuitenkin huonompi suorituskyky verrattuna muihin kieliin. (WAC, 2024)

NodeJS on JavaScript-pohjainen kehys palvelimen luontiin. JavaScript on normaalisti huono valinta palvelinpuolen kieleksi, sillä se ei tarjoa muuttujien tyyppien määrittelyä. Se on myös turvallisuudeltaan ja laadultaan riskialtis. NodeJS tarjoaa kuitenkin työkaluja taklaamaan normaalit JavaScriptin ongelmat. Pienempiin projekteihin NodeJS on hyvä valinta, jolloin kehittäjä voi toteuttaa selainpohjaisen sovelluksen käyttäen JavaScriptia sekä käyttöliittymässä että palvelinpuolella. NodeJS:lla on nopea toteuttaa palvelin, mutta kehittäjän tulisi kiinnittää erityistä huomiota tietoturvaan ja koodin laatuun. (WAC, 2024)

3.2.2 Vähemmän suositut kielet

Golang on Googlen luoma palvelinpuolen kieli. Se on hyvin suorituskykyinen ympäristö ja soveltuu isoille käyttäjämäärille sekä pilvipalveluihin. Golang ei ole kuitenkaan yhtä suosittu kuin Java, Python, C# tai JavaScript. Tämä voi aiheuttaa haasteita, kuten löytää sopivia valmiita kirjastoja tai ratkaisuja ongelmiin. (WAC, 2024)

Rust on Mozilla-säätiön kehittämä ohjelmointikieli. Se on myös hyvin suorituskykyinen ympäristö ja tarjoaa hyviä tietoturvaominaisuuksia. Se onkin hyvä valinta, jos ei haluta tinkiä luotettavuudesta yhtään. Golangin tavoin se ei ole kuitenkaan yhtä suosittu. Sen suosiota rajoittaa myös melko korkea oppimiskynnys. (WAC, 2024)

PHP on historiassa ollut hyvin suosittu kieli toteuttaa palvelimia. Se on kuitenkin melko hidas verrattuna uudempiin totetuksiin, ja siihen liittyy paljon tietoturvauhkia, jotka tulee ottaa huomioon kehitettäessä. (WAC, 2024)

3.3 Rajapinnat

Rajapintoja voidaan tarkastella tässä työssä kahdessa eri osassa: palvelimen ja tietokannan välinen rajapinta sekä palvelimen ja käyttöliittymän välinen rajapinta. Näihin on tarjolla monia eri ratkaisuja. Alla on esitettynä muutamia, joilla voidaan saavuttaa sama lopputulos käyttäen muita teknologioita kuin tässä työssä on käytetty.

Palvelimen ja tietokannan välisessä rajapinnassa käydään läpi ORM-työkaluja ja niiden eroja suoriin SQL-kyselyihin. Palvelimen ja käyttöliittymän välisessä rajapinnassa käsitellään eri menetelmiä luoda rajapinta sekä reaaliaikaiseen tiedonsiirtoon liittyviä protokollia.

3.3.1 Palvelimen ja tietokannan väliset rajapinnat

Relaatiotietokantaa muokataan SQL-kyselyillä. Palvelimelta voidaan tehdä suoraan SQL-kyselyitä kantaan, mutta monissa olio-ohjelmointikielissä on myös tarjolla olio-relaatiokartoitus-työkaluja, eli ORM (*Object Relational Mapping*) -työkaluja. ORM-työkalut ovat yleensä olio-ohjelmointiin tottuneelle kehittäjälle helpompia käyttää kuin suorat SQL-kyselyt. Eri kielien tarjoamat ORM-työkalut kuitenkin eroavat toisistaan ja sisältävät kehyskohtaisia komentoja. Monimutkaisissa kyselyissä ORM-työkalut ovat myös monesti suoria SQL-kyselyitä hitaampia, ja tällaisissa tilanteissa voikin olla tarvetta edelleen käyttää suoria SQL-kyselyitä, jotta suorituskyky pysyy riittävänä. (Ihechikara, 2022)

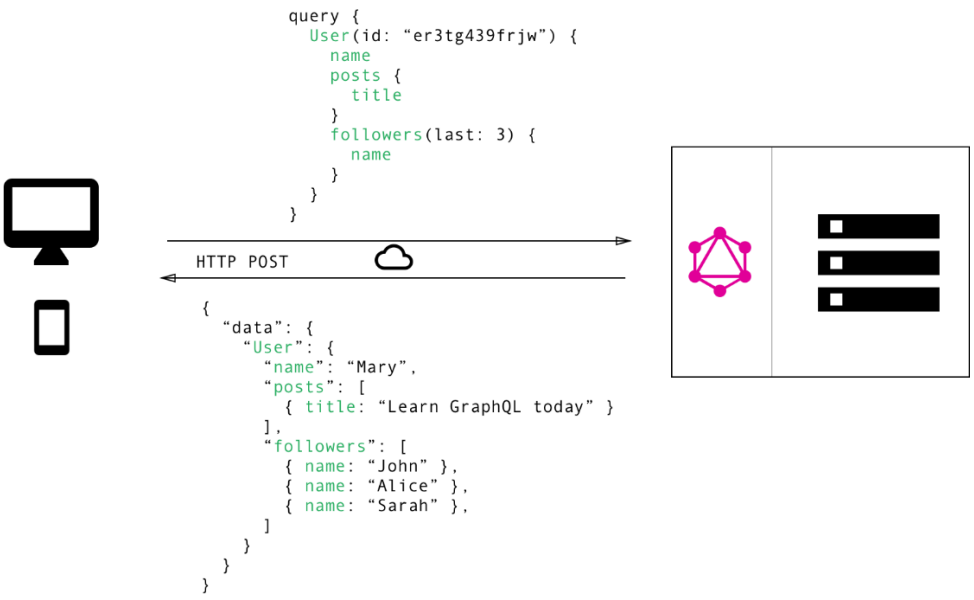
Työssä on käytetty .NET-ympäristön Entity-kehystä, joka tarjoaa ORM-työkalun. Tästä on kerrottu tarkemmin luvussa 2.3.2. Javalta löytyy Hibernate-niminen ORM-työkalu. Pythonin Django-kehykseen on sisäänrakennettu ORM-työkalu, ja PHP-kielelle löytyy Laravel-kehys, joka sisältää Eloquent-nimisen ORM-työkalun. (Ihechikara, 2022)

3.3.2 Palvelimen ja käyttöliittymän väliset rajapinnat

Palvelimen ja käyttöliittymän väliseen rajapintaan voidaan toteutuksesta riippumatta käyttää aina REST-rajapintaa, josta on kerrottu luvussa 2.4.1 tarkemmin. REST:n sijaan rajapintana voidaan käyttää myös GraphGL:ää. GraphGL on REST:n tavoin menetelmä luoda rajapinta, ja sitä voidaan käyttää kaikkien yleisien tiedonsiirtoprotokollien yhteydessä. Sen ajatuksena on tarjota kaikki saatavat kyselyt yhdellä kertaa mutta siten, että käyttöliittymä määrittelee vain sen, mitä haluaa hakea käyttöliittymältä. GraphGL:n dokumentaatiosta löytyy esimerkkikuvat 4 ja 5, joissa havainnollistetaan REST:n ja GraphGL:n eroja. Siinä on esitetty saman tiedon hakemista kolmella REST GET -kutsulla, jotka voitaisiin hakea yhdellä GraphGL POST -kutsulla. REST GET -kutsut voitaisiin myös ketjuttaa yhdeksi GET-kutsuksi, mutta silloin saatettaisiin päätyä tilanteeseen, että kutsuissa palautetaankin ylimääräistä tietoa. GraphGL:n etuna tietoliikenne saadaan optimoitua, koska kyselyssä palautetaan vain halutut tiedot. (GraphGL, n.d.)

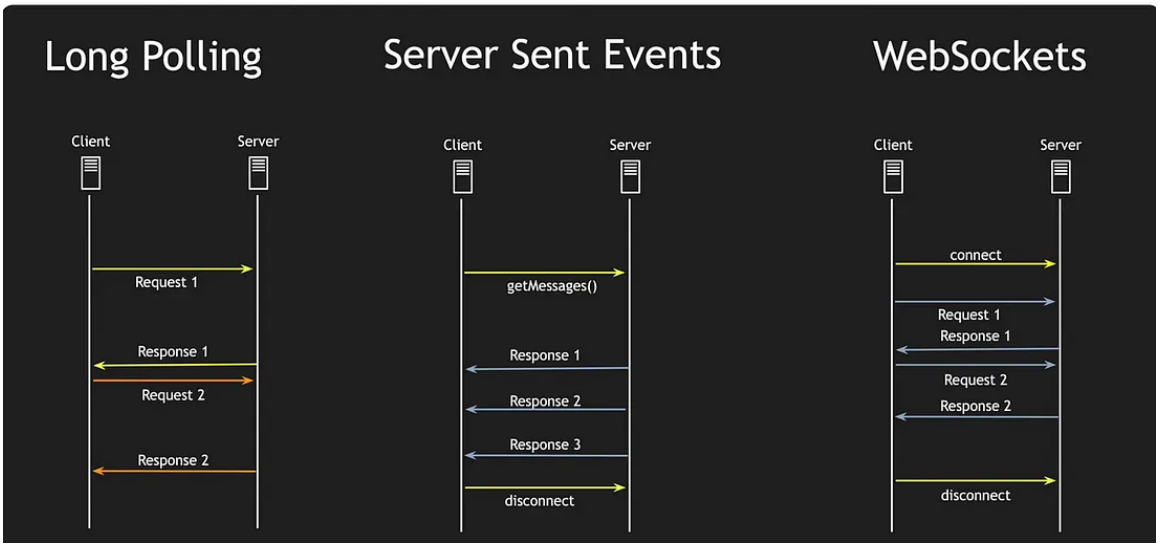
Kuva 4. GraphGL:n dokumentaatiossa REST API esitettynä (GraphGL, n.d.).

Kuva 5. GraphGL:n dokumentaatiossa GraphGL esitettynä (GraphGL, n.d.).



Reaaliaikaisen tiedonsiirron kaltaiseen lopputulokseen voidaan päästä myös normaaleilla HTTP-kutsuilla, jos käyttöliittymään luodaan ajastin, joka hakee uudet tiedot tietyin intervallein. Tästä tavasta käytetään myös nimeä *short polling*. Tämä ei yleensä ole kuitenkaan järkevin ratkaisu, sillä se aiheuttaa vääjäämättä ylimääräistä kuormitusta tiedonsiirtoon. Yleisesti käytössä olevat menetelmät reaaliaikaiseen tiedonsiirtoon ovat pitkät HTTP-kutsut, WebSockets-protokolla ja palvelimelta lähtevät tapahtumat. Pitkät HTTP-kutsut toimivat kuten normaalit niin sanotut lyhyet HTTP-kutsut, mutta palvelin jääkin pitämään kyselyä siihen asti auki, kunnes tiedot päivittyvät, ja lähettää sen jälkeen vastauksen takaisin. Tämän jälkeen käyttöliittymä voi lähettää uuden kutsun, joka jää taas odottamaan tietojen päivittymistä. WebSocket luo ensin yhteyden palvelimen ja käyttöliittymän välille, jonka jälkeen se mahdollistaa kaksisuuntaisen tiedonsiirron näiden välillä, kunnes jompikumpi sulkee yhteyden. Palvelimelta lähtevät tapahtumat perustuvat siihen, että palvelin lähettää yksisuuntaisena vastauksen käyttöliittymälle aina, kun palvelimen tiedot päivittyvät, jolloin käyttöliittymä kuntelee vain palvelimelta tulevia tapahtumia (Pubnub, 2023). Pitkien HTTP-kutsujen, WebSocket-protokollan ja palvelimelta lähtevien tapahtumien toimintaperiaatteita on esitetty kuvassa 6.

Kuva 6. Reaaliaikaisten tiedonsiirtomenetelmien toimintaperiaatteet (Asharsaleem, 2024).



Työssä käytetty SignalR on Microsoftin luoma kirjasto .NET-ympäristöön ja sitä voidaan käyttää kaikkiin selainpohjaisiin käyttöliittymiin. Sen toiminnasta on kerrottu tarkemmin luvussa 2.4.2. SignalR-kirjastoa voidaan käyttää myös natiiveihin mobiilisovelluksiin, sillä se tukee yhteyttä Java- ja Swift-pohjaisiin sovelluksiin. SignalR vaatii kuitenkin palvelimeksi aina .NET-ympäristön. (Microsoft, 2025-i)

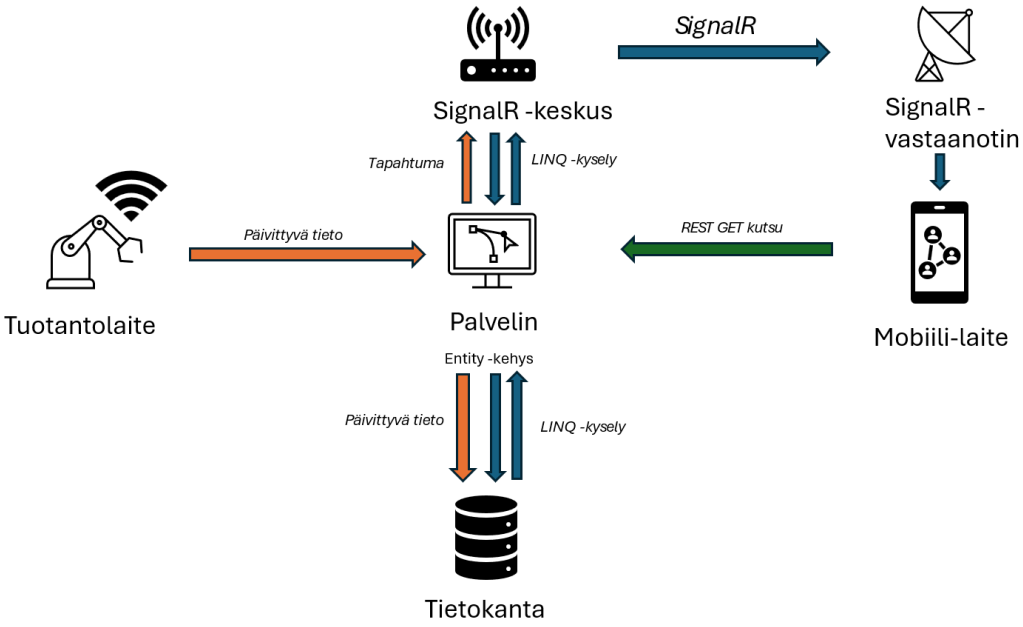
Socket.IO on myös työkalu reaaliaikaiseen tiedonsiirtoon. Se hyödyntää tiedonsiirtoon WebSocket-protokollaa, WebTransport-protokollaa, sekä pitkiä HTTP-kutsuja. WebTransport-protokolla on uudempi versio WebSocket-protokollasta. Se hyödyntää tiedonsiirrossa HTTP/3:a ja QUIC-protokollaa eikä perinteistä TCP-protokollaa (Mozilla, n.d.). Socket.IO:n etuna on sen laaja soveltuvuus eri kielille. Palvelinpuolella sitä voidaan käyttää Java-, NodeJS-, Python-, Golang- ja Rust -kielillä. Käyttöliittymäpuolella sitä voidaan käyttää JavaScript-, Java-, Swift-, Dart-, Python-, .NET-, Rust-, Kotlin-, PHP-, Golang- ja C++ -kielillä. Käytännössä se siis soveltuu kaikkiin muihin esitettyihin vaihtoehtoihin paitsi tilanteeseen, joissa palvelin on toteutettuna PHP:llä tai .NET-ympäristössä. (Socket.IO, n.d.)

Tämän lisäksi on olemassa monia muita vastaavanlaisia valmiita työkaluja reaaliaikaiseen tiedonsiirtoon. Näistä esimerkkeinä on muun muassa Java-ympäristössä toimivat Apache Kafka- ja Apache Flink -kehykset (Obregon, 2023).

4 Toteutus

Työssä toteutetaan Tuotannon monitorointi -moduuli mobiilikäyttöliittymään. Kun moduulin näkymä avataan ensimmäisen kerran, käyttöliittymä hakee tiedot palvelimelta REST-rajapinnan GET-kutsulla. Tämän jälkeen käyttöliittymän ja palvelimen välille avataan reaaliaikaiseen tiedonsiirtoon SignalR-yhteys. Tuotantolaite päivittää tietoja palvelimen kautta tietokantaan. Tiedon päivityksen yhteydessä SignalR-keskukselle lähetetään tapahtuma, jolla SignalR-keskus saa tiedon, että tietokantaan on tullut uutta tietoa. SignalR-keskus lähettää LINQ-kyselyn tietokantaan, hakee sieltä uudet tiedot ja lähettää ne SignalR-vastaanottimeen mobiilikäyttöliittymällä. Tuotantolaitteen ja tietokannan välinen tiedonsiirto on rajattu tämän työn ulkopuolelle. Kuvassa 7 on esitetty haluttu lopputulos tiedonsiirrosta. Tuotantolaitteen siirtämä tieto on esitetty oranseilla nuolilla, ja SignalR:n kautta liikkuva tieto on esitetty sinisillä nuolilla. Tämän lisäksi ensimmäinen tiedonsiirto suoritetaan REST-rajapintaa käyttäen, ja se on esitetty vihreällä. Palvelimen puolella GET-kutsu käyttää samaa LINQ-kyselyä tiedonhakuun tietokannasta kuin SignalR.

Kuva 7. Toteutuksen tiedon kulku tuotantolaitteelta mobiililaitteelle.



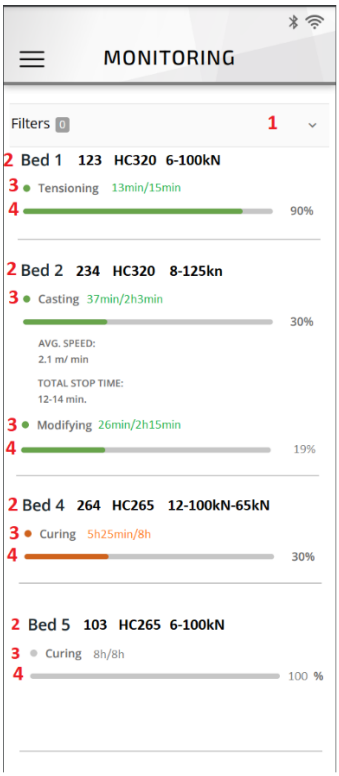
Mobiilikäyttöliittymä on toteutettu Quasar-kehyksellä ja Cordova-laajennuksella. Käyttöliittymän lähdekoodi on kirjoitettu Vue 3:lla ja sisältää pääasiassa OptionsAPI:ia,

mutta myös CompositionAPI:ia. Työssä toteutettava moduulinäkymä tehdään CompositionAPI-tyylillä. Palvelin on toteutettu .NET Core -kehitysympäristöllä C# -kielellä. Palvelin mukailee CLEAN-arkkitehtuuria ja hakee tiedot tietokannasta Entity-kehystä hyödyntäen.

4.1 Käyttöliittymän näkymän hahmottelu

Työn toteutus aloitetaan hahmottelemalla näkymä käyttöliittymään annetun luonnoksen (kuva 8) perusteella. Luonnoksessa näkyy joukko valualustoja, joilla on eri työvaiheet menossa. Työvaiheista halutaan nähdä edistymisaste sekä työhön käytetty aika ja se, onko se linjassa suunniteltuun aikatauluun. Nämä kaikki tiedot ovat jo saatavilla tietokannasta, jolloin työstettäväksi jää vain tietojen tuominen käyttöliittymään ja niiden esittäminen halutussa muodossa.

Kuva 8. Annettu luonnos halutusta näkymästä (Elematic Oyj, 2025).



Jotta käyttöliittymään saadaan lisättyä uusi näkymä valikkoon, täytyy koodiin lisätä näkymää varten *route* eli reitti. Quasar käyttää reititykseen Vue Router -kirjastoa, josta työssä on käytössä versio 4. Vue Routerilla voidaan määritellä sovelluksen reitit ja ottaa ne helposti käyttöön eri puolilla sovellusta. Reitit määritellään normaalisti Route.js-tiedostossa, jonne luodaan ”routes” -muuttuja, johon määritellään kaikki halutut reitit ja mihin komponentteihin reitti on liitetty (VueRouter, n.d.). Reiteillä voi olla myös alakomponentteja eli *children*, mutta toistaiseksi Tuotannon monitorointi -moduulia varten riittää yksi näkymä, ja tälle voidaan luodaan uusi reitti päätteestä ”/monitoring”, johon liitetään uusi komponentti. Normaalisti reittien lisäksi tulee määritellä reititin (*router*). Työ toteutetaan kuitenkin olemassa olevaan käyttöliittymään, johon reititin on jo määriteltynä. Reitittimen ja reittien määritellystä on esitetty esimerkkikoodi kuvassa 9.

Kuva 9. Näyttöleike Vue Routerin esimerkkikoodista reitittimen ja reittien määrittelystä (VueRouter, n.d.).

```js
import { createMemoryHistory, createRouter } from 'vue-router'

import HomeView from './HomeView.vue'
import AboutView from './AboutView.vue'

const routes = [
  { path: '/', component: HomeView },
  { path: '/about', component: AboutView },
]

const router = createRouter({
  history: createMemoryHistory(),
  routes,
})
```

Komponentin sisällöksi riittää alustavasti pelkästään yksi HTML-elementti. Moduulia varten täytyy vielä lisätä uusi kuvake etusivulle, johon on liitetty uusi reitti hyödyntäen Vue Routerin tarjoamaa <RouterLink /> -komponenttia. Navigoinnin luomisesta <RouterLink /> -komponenteilla on esitetty esimerkki kuvassa 10. Linkin jälkeen etusivulta saadaan auki luotu komponentti, joka toistaiseksi sisältää vain yhden HTML-elementin.

Kuva 10. Näyttöleike VueRouterin esimerkistä <RouterLink /> -komponentin käyttöön (VueRouter, n.d.).

```vue
<template>
  <h1>Hello App!</h1>
  <p>
    <strong>Current route path:</strong> {{ $route.fullPath }}
  </p>
  <nav>
    <RouterLink to="/">Go to Home</RouterLink>
    <RouterLink to="/about">Go to About</RouterLink>
  </nav>
  <main>
    <RouterView />
  </main>
</template>
```

Luodaan seuraavaksi uuteen komponenttiin sisältöä. Kuvan 8 luonnos voidaan jakaa karkeasti kolmeen osaan. Ensimmäisessä osassa on koko sivun rakenne, jonka määrittely on jo tehty valmiiksi sovelluksen baselayout.vue-tiedostossa. Tässä on määriteltynä, kuinka valikko näkyy ja kuinka näkymä skaalautuu eri laitteilla. Toisessa osassa on filteröintikomponentti, joka voidaan luoda hyödyntämällä Quasarin *expansion-item*-komponenttia (Quasar, n.d-b). Se mahdollistaa valikon avaamisen ja pienentämisen klikkaamalla. Kolmannessa osassa on lista valualustoista ja niiden työvaiheista. Tämä osio vaihtelee dynaamisesti luettujen tietojen perusteella.

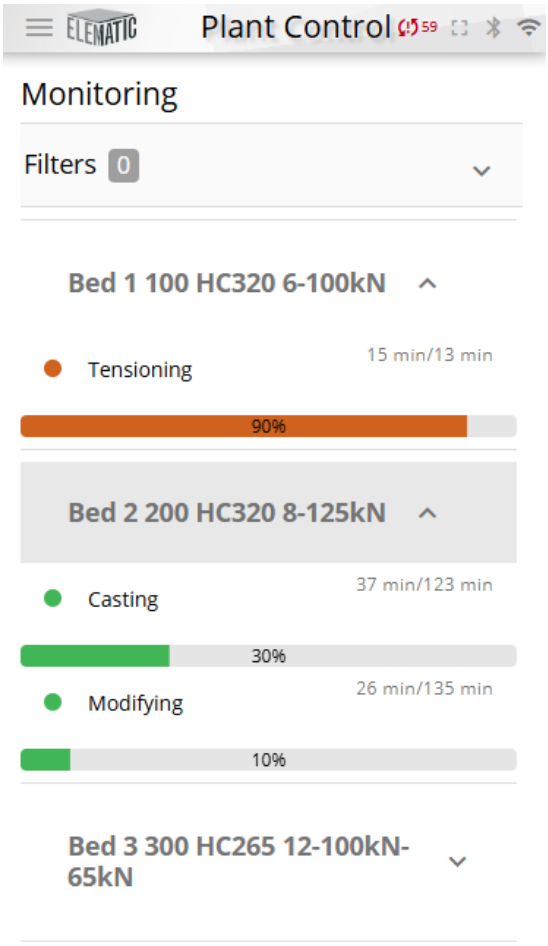
Lista valualustoista voidaan pilkkoa vielä pienempiin osiin. Yksi valualusta voi sisältää useamman työvaiheen samanaikaisesti. Jokaisella valualustalla on otsikko, jossa on määritelty valualustan numero ja sillä menossa oleva tuote. Tämän alla on lista työvaiheista. Yhdellä työvaiheella on otsikorivi, jossa näkyy menossa oleva työvaihe, sen tila, siihen käytetty aika ja siihen arvioitu aika. Otsikkorivin alla näytetään työvaiheen valmistumisaste. Valmistumisasteen esittämistä varten Quasar tarjoaa valmiin *linear-progress*-komponentin (Quasar, n.d-c). Lisäksi työvaiheella voidaan näyttää tuotantolaitteen tarjoamia mahdollisia lisätietoja.

Koska palvelimen rajapintoja ei ole vielä luotu, täytyy uutta näkymää testata luomalla testidataa. Testidatan olisi hyvä jäljitellä mahdollisimman tarkkaan palvelimelta tulevaisuudessa luettavaa dataa, jotta komponentin logiikkaa ei tarvitse muuttaa

jälkikäteen. Testidata on toistaiseksi luotu pelkkänä JSON-tiedostona, joka ladataan suoraan komponentille ilman erillisiä kutsuja.

Kun testidata on luotu, voidaan komponentin logiikassa ottaa testidata käyttöön ja liittää data luotuihin HTML-elementteihin. Käyttöliittymän osalta Tuotannon monitorointi -moduulin ensimmäinen versio on nyt valmis ja se on esitetty kuvassa 11.

Kuva 11. Moduulin ensimmäinen versio selaimen mobiililaite (*mobile device*) -näkymällä.



4.2 Entities-määrittely palvelimelle

Toteutetaan seuraavana palvelimen ja tietokannan välinen rajapinta. Määrittelyä varten täytyy halutut tietokannan taulukot luoda Entity-luokkina palvelimelle. Entity-luokkaan määritellään lisäksi navigaatio muihin Entity-luokkiin. Navigaatiota varten kannassa ei

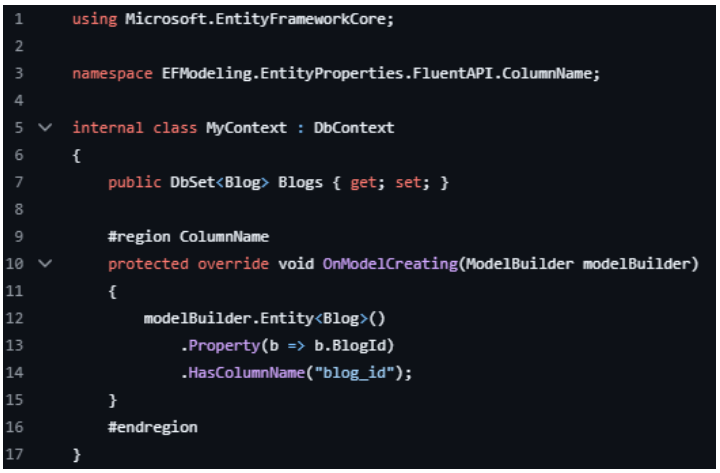
tarvitse olla valmiita viiteavaimia taulukoiden välillä luotuna, vaan ne voidaan tarvittaessa luoda keinotekoisesti Entities-kehyksen avulla. Entity-luokkien väliset liitokset ovat ”yksi moneen”, ”moni moneen” tai ”yksi yhteen”. Yksi moneen -liitos tarkoittaa, että luokan A olio voi olla liitettynä useaan luokan B olioon, mutta B-luokan olioon voi olla liitettynä vain yksi A-luokan olio. Vastaavasti moni moneen -liitos mahdollistaa, että luokan A oliolla voi olla liitettynä monta B-luokan oliota ja B-luokan oliolla voi olla monta A-luokan oliota. Yksi yhteen -liitos taas rajoittaa, että luokan A oliolla voi olla vain yksi B-luokan olio ja B-luokan oliolla vain yksi A-luokan olio. Entity-luokassa Navigaation tyypiksi tulee liitettävä luokka tai liitettävän luokan kokoelma eli ”ICollection<T>”-tyyppi (Microsoft, 2023-e). Entity-kehyksen dokumenteista löytyy esimerkkikoodia, jota on esitetty alla kuvissa 12 ja 13. Kuvasta 12 voidaan huomata, että luokan Post määrittelyyn on liitetty yksi Blog-olio. Vastaavasti kuvasta 13 voidaan nähdä, että Blog-luokkaan voidaan liittää useampi Post-olio.

Kuva 12. Näyttöleike Entity-kehyksen dokumentaatiosta, jossa esitettynä Post-luokka (Microsoft, 2023-e).

Kuva 13. Näyttöleike Entity-kehyksen dokumentaatiosta, jossa esitettynä Blog-luokka (Microsoft, 2023-e).

Entity-luokkien määrittelyn jälkeen siirrytään Contextin luontiin. Siinä määritellään, mitkä palvelimen Entity-luokat ja niiden ominaisuudet eli *properties* ovat yhdistettynä mihinkin tietokannan taulukkoon ja niiden sarakkeisiin. Contextissa määritellään myös navigaatioiden liitokset (Microsoft, 2023-f). Contextin määrittelyyn käytetään Entity-kehyksen ModelBuilder-työkalua, joka tunnetaan myös FluentAPI:na. Se mahdollistaa Entity-luokkien konfiguroinnin ilman luokkien muokkausta. Contextin määrittelystä löytyy Entity-kehyksen dokumentaatiosta esimerkkikoodia, joita on esitelty alla kuvissa 14 ja 15. Kuvassa 14 on esitettynä ominaisuuksien yhdistämistä tietokannan taulukon sarakkeeseen. Kuvassa 15 on esitettynä navigaatioiden määrittelyä. Taulukoiden välille voidaan luoda liitos, vaikka kannassa näiden välillä ei olisikaan vierasavainta (*foreign key*).

Kuva 14. Näyttöleike Entity-kehyksen Github-projektin tiedostosta ColumnName.cs (Github, n.d.-a).



```
using Microsoft.EntityFrameworkCore;

namespace EFModeling.EntityProperties.FluentAPI.ColumnName;

internal class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    #region ColumnName
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .Property(b => b.BlogId)
            .HasColumnName("blog_id");
    }
    #endregion
}
```

Kuva 15. Näyttöleike Entity-kehyksen Github-projektin tiedostosta NoForeignKey.cs (Github, n.d.-b).

```
1   using System.Collections.Generic;
2   using Microsoft.EntityFrameworkCore;
3
4   namespace EFModeling.Relationships.FluentAPI.NoForeignKey;
5
6   #region NoForeignKey
7   internal class MyContext : DbContext
8   {
9       public DbSet<Blog> Blogs { get; set; }
10      public DbSet<Post> Posts { get; set; }
11
12      protected override void OnModelCreating(ModelBuilder modelBuilder)
13      {
14          modelBuilder.Entity<Post>()
15              .HasOne(p => p.Blog)
16              .WithMany(b => b.Posts);
17      }
18  }
19
```

4.3 Kyselyiden tekeminen tietokantaan

Kun Entity-luokat ja Context ovat määritelty, voidaan tietokantaan alkaa tekemään kyselyitä eli *Query*jä. Entity-kehys tarjoaa mahdollisuuden LINQ:iin. LINQ on tekniikka, jolla voidaan suorittaa tietokantakyselyitä C# -kielellä. Sen nimi tulee englannin kielen sanoista *Language Integrated Query*. LINQ:n ansiosta kehittäjän ei tarvitse kirjoittaa tietokannan käyttämiä SQL-kyselyitä, vaan voi tehdä samat kyselyt käyttäen kehykseen integroituja komentoja C# -kielellä (Microsoft, 2021-g). Entity-kehyksen dokumentaatiosta löytyy esimerkkikoodia kyselyjen tekemiseen LINQ:llä. Kuvasta 16 on esitettynä yksinkertainen yhden blogin haku tietokannasta hyödyntäen ”.SingleAsync()”-funktiota.

Kuva 16. Näyttöleike Entity-kehyksen dokumentaatiosta, jossa esitetty yhden tiedon hakeminen blogin id:n perusteella (Microsoft, 2021-j).

```
C#                                                          Copy

using (var context = new BloggingContext())
{
    var blog = await context.Blogs
        .SingleAsync(b => b.BlogId == 1);
}
```

Kyselyt tulisi määritellä siten, että yksi kysely toteuttaa aina yhden selkeän tehtävän taulukkoon, jotta vältetään liian suurien kyselyjen tekeminen. Suuret kyselyt kuormittavat tietokantaa ja voivat hidastaa tiedonsiirtoa. Etenkin, jos Entity-luokkien välillä on paljon riippuvuuksia, joita haetaan samassa LINQ-kyselyssä, kyselyistä muodostuu herkästi hyvin raskaita. LINQ-kyselyt muunnetaan SQL-kielelle ja kyselyn sisältämät ”.Include()”-metodit muunnetaan SQL:n komennoksi ”JOIN”. Jos LINQ-kysely sisältää useita ”.Include()”-lausekkeita, niistä muodostuu rinnakkaisia ”JOIN”-komentoja, jotka voivat kasvattaa hakutuloksia ekspotentiaalisesti. Tästä on esimerkkinä kuvat 17 ja 18. Kuva 17 sisältää kaksi ”.Include()”-metodia, jotka muunnetaan kuvan 18 muotoiseen SQL-kyselyyn. Entity-kehyksen dokumentaatiossa on kerrottu, että jos kuvien blogilla on 10 kpl viestejä ja 10 kpl avustajia, tulee kuvien kyselyllä yhtä blogia varten jo 100 osumaa. (Microsoft, 2024-h)

Kuva 17. Näyttöleike Entity-kehyksen dokumentaatiosta, jossa esitetty blogin ja siihen liitettyjen viestien ja avustajien haku LINQ-kyselyllä (Microsoft, 2024-h).

```c#
var blogs = await ctx.Blogs
    .Include(b => b.Posts)
    .Include(b => b.Contributors)
    .ToListAsync();
```

Kuva 18. Näyttöleike Entity-kehyksen dokumentaatiosta, jossa esitetty kuvan 17 LINQ-kysely SQL-kielelle käännettynä (Microsoft, 2024-h).

```SQL
SELECT [b].[Id], [b].[Name], [p].[Id], [p].[BlogId], [p].[Title], [c].[Id], [c].[BlogId], [c].[FirstName],
FROM [Blogs] AS [b]
LEFT JOIN [Posts] AS [p] ON [b].[Id] = [p].[BlogId]
LEFT JOIN [Contributors] AS [c] ON [b].[Id] = [c].[BlogId]
ORDER BY [b].[Id], [p].[Id]
```

Toteutusta varten riittää, että luodaan tarvittavat kyselyt, joilla halutut tiedot haetaan tietokannasta. Tietoja ei tarvitse päivittää tietokantaan, sillä tuotantolaitteen ja palvelimen välinen tiedonsiirto on rajattu toteutuksen ulkopuolelle.

4.4 REST API -pääte

Toteutusta varten luodaan ensin normaali REST-rajapinta palvelimelle. REST:n periaatteena on tarjota käyttöliittymälle valmiita päätteitä, joilla käyttöliittymä voi suorittaa toimintoja palvelimelle. Työtä varten toteutetaan GET-pääte, jonka tehtävänä on tuoda halutut valualustat ja niiden työvaiheet, kun käyttöliittymän moduuli avataan ensimmäisen kerran.

Rajapintaa varten palvelimelle täytyy luoda *ViewModel*, *Controller*, *Interface* ja *Service*. Kontrollerin (*Controller*) tehtävä on luoda ja käsitellä REST:n käyttämät päätteet, eli *endpoints*. REST-päätteet ovat aina joku toiminto, jonka kontrolleri suorittaa. Kun HTTP-pyyntö lähetetään määriteltyyn osoitteeseen, ottaa kontrolleri sen vastaan ja lähettää tiedon eteenpäin rajapinnan (*Interface)* kautta palvelulle (*Service)*. Palvelu kutsuu tarvittavia kyselyitä ja palauttaa kyselyiden palauttamat halutut tiedot kontrollerille. Näkymämallit (*ViewModel*) määrittelevät minkälaisessa muodossa kontrollerin vastaanottamat ja lähettämät tiedot tulisivat olla.

4.4.1 Palvelu (*service*)

Palvelun tehtävänä on kutsua haluttuja kyselyitä ja lähettää tiedot eteenpäin. Kyselyt halutaan pitää lähtökohtaisesti yksinkertaisina ja tiedot suoraan tietokannan tai Entity-luokan palauttamassa muodossa. Tietoja voi olla tarvetta jäsennellä, suorittaa laskentaa, yhdistää useampien eri kyselyiden tietoja tai muuten muokata tietoja. Nämä kaikki suoritetaan ensisijaisesti palvelussa.

Tätä toteutusta varten riittää, että luodaan yksi palvelu. Sen tehtävänä on suorittaa tarvittavat luodut kyselyt kantaan ja palauttaa tiedot jäsenneltynä. Kontrolleri kutsuu palvelua rajapinnan kautta.

4.4.2 Rajapinta (*Interface*)

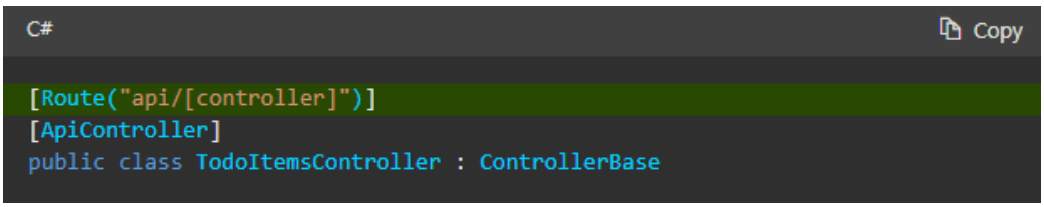
C# -olio-ohjelmointikieli tarjoaa viitetyypin nimeltä *interface*, eli rajapinta. Luokalle voidaan luoda useampi rajapinta. Nämä toimivat abstraktina kerroksena luokalle (Microsoft, 2023-k). Tässä työssä rajapinta luodaan palvelulle ja kontrolleri kutsuu rajapintaa palvelun sijaan.

Logiikan kasvaessa useampi kontrolleri voi käyttää samaa palvelua. Myöhemmin voi tulla tarvetta tehdä muutoksia yhteen kontrollereista tai lisätä uusi kontrolleri käyttämään palvelua. Näillä kontrollereilla voi olla tarvetta käyttää palvelua hieman eri tavoin. Rajapinnan avulla palvelu voidaan pitää muuttumattomana ja kontrollereiden väliset erot voidaan toteuttaa rajapinnan avulla. Toteutusta varten rajapinta on kuitenkin toistaiseksi vain yksi kerros, jonka ainoa tehtävä on olla apuna tulevaisuuden muutoksissa ja lisäyksissä.

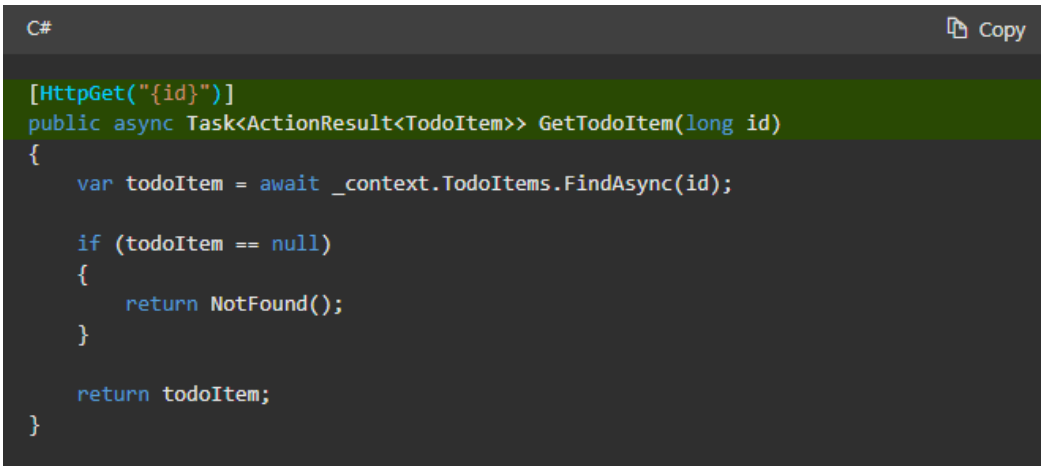
4.4.3 Kontrolleri (*Controller*)

Kontrollerin tehtävänä on tarjota REST:ä varten pääte. Se ottaa vastaan päätteeseen tehdyn kyselyn, validoi sen ja palauttaa sen perusteella vastauksensa. Työtä varten luodaan HTTP GET -toiminto, jota kutsuttaessa haetaan kaikki valualustat ja niiden työvaiheet ja tiedot. Uusi kontrolleri perii valmiin ControllerBase-luokan. Tämän lisäksi kontrollerille määritellään reitti, josta kaikki päätteet löytyvät. Kontrollerin luonnista löytyy esimerkkikoodia ASP.NET:n dokumentaatiosta, jota on esitetty kuvissa 19 ja 20. Kuvassa 19 luodaan uusi kontrolleriluokka ja määritellään reitti. Kuvassa 20 luodaan varsinainen pääte. Esimerkissä on luotu GET-pyyntö, osoitteesta ”/api/todoitem/id”. Osoitteessa oleva muuttuja id poimitaan ja lähetetään palvelulle suoritettavaksi. Palvelu palauttaa tiedon muuttujaan, ja jos muuttuja on tyhjä eli id:tä ei löydy, se palauttaa tiedon kyselyn lähettäjälle, että tietoa ei löytynyt. Jos id sen sijaan löytyy, lähetetään kyselyn lähettäjälle palvelun palauttama tieto. (Microsoft, 2025-l)

Kuva 19. Näyttöleike C#:n dokumentaation esimerkkikoodista uuden kontrollerin luonnista (Microsoft, 2025-l).



Kuva 20. Näyttöleike C#:n dokumentaation esimerkkikoodista REST GET -päätteen luonnista (Microsoft, 2025-l).



Vastaavasti työtä varten toteutetaan GET-kysely, joka kutsuu luodun palvelun rajapintaa. Jos tietoja ei löydy, käyttäjälle palautetaan 404 HTTP -status. Muussa tapauksessa palautetaan 200 HTTP -status sekä palvelun palauttama tieto validoituna näkymämallin mukaiseen muotoon.

4.4.4 Näkymämalli (*ViewModel*)

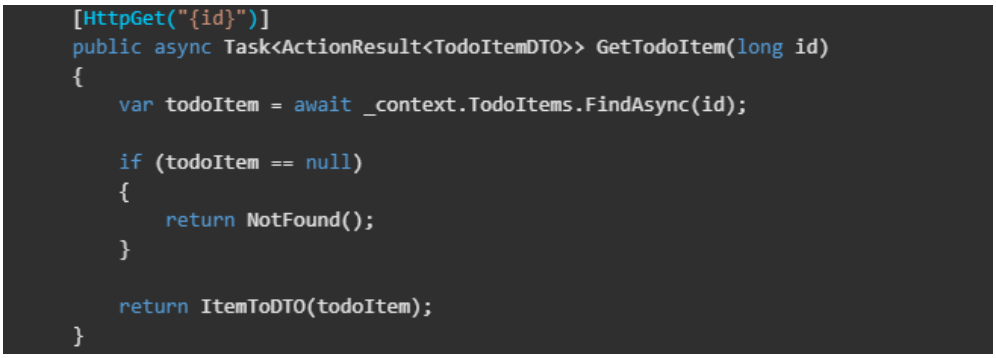
Näkymämallin tehtävä on validoida tiedot haluttuun formaattiin. Tällä vältetään se, ettei tiedonsiirtoon pääse virheellistä tietoa, ja samalla suojellaan logiikassa käsiteltäviä mahdollisesti salaisia tietoja. Näkymämallin muotoon muunnetusta oliosta käytetään

yleensä lyhennettä DTO, joka tulee englannin kielen sanoista *Data Transfer Object* eli tiedonsiirto-olio.

Aiemmassa luvussa esitetty esimerkki GET-päätteestä voidaan muuttaa käyttämään näkymämallia. Kuvassa 21 luodaan näkymämalli normaalin luokan luonnin tapaan ja kuvassa 22 se otetaan GET-päätteeseen käyttöön määrittelemällä palautettavaksi tyypiksi kyseinen tiedonsiirto-olio ja luomalla uusi tiedonsiirto-olio palvelun palauttamasta muuttujasta.

Kuva 21. Näyttöleike C#:n dokumentaation esimerkkikoodista näkymämallin luonnista (Microsoft, 2025-l).

Kuva 22. Näyttöleike C#:n dokumentaation esimerkkikoodista GET-päätteen luonnista näkymämallin kanssa (Microsoft, 2025-l).



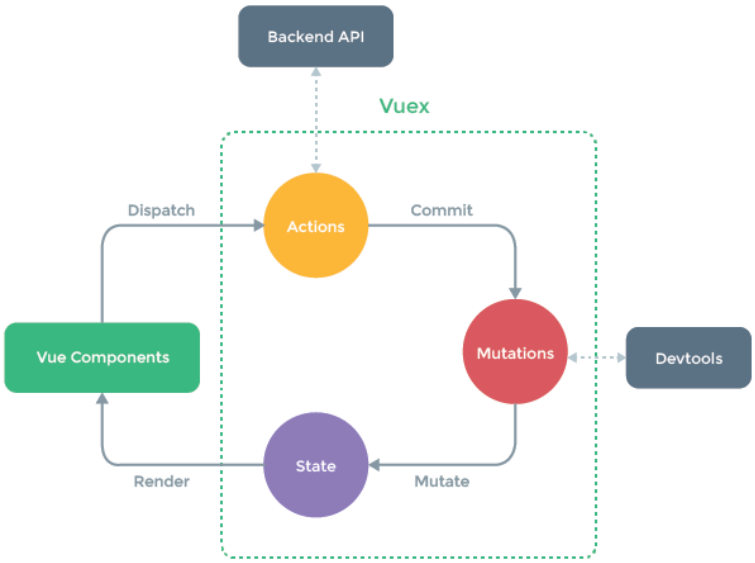
Esimerkkien tapaan työssä luodaan uusille palautettaville tiedoille tarvittavat näkymämallit ja kontrollerissa tiedot muunnetaan tiedonsiirto-olioiksi. Täten vastaanottaja ei näe Entity-luokkien määrittelyä vaan ainoastaan tiedonsiirto-olion muodon.

4.5 Käyttöliittymän Store ja REST API:n tietojen haku

Normaalisti tilanhallinta toimii komponentin elinkaaren ajan. Sovelluksen kasvaessa tilanhallinta voidaan haluta irrottaa komponentilta erilliseksi. Tätä varten on olemassa erilaisia tilanhallintakirjastoja. Työssä on käytössä tilanhallintakirjasto nimeltä Vuex ja sen versio 4. Nykyään tilanhallintakirjastoksi Vuen virallisessa dokumentaatiossa suositellaan Piniaa. Vuex versio 4:ää edelleen ylläpidetään, eikä työtä varten ole tarvetta lähteä päivittämään kirjastoa uudempaan. (Vuex n.d-a)

Vuex-tilanhallinnan toiminnan logiikkaa on esitetty Vuexin dokumentaatiosta löytyvällä kaaviolla kuvassa 23. Kuvasta nähdään, kuinka Vuexin toiminnoilla (*Actions*) haetaan tiedot palvelimen rajapinnasta, jonka jälkeen API:lta saatu vastaus lähetetään toimminnosta muunnettavaksi (*Mutations*), jossa se käsitellään ja asetetaan uudeksi tilaksi (*state*). Kun tila muuttuu, se renderöityy komponentille uudestaan, jolloin komponentin tila päivittyy käyttöliittymässä. (Vuex n.d-a)

Kuva 23. Näyttöleike Vuexin dokumentaation kaaviosta, jossa esitetty Vuex-kirjaston toimintaperiaate (Vuex, n.d.-a).



Työtä varten moduulille luodaan oma kansio storelle ja sinne määritellään omat tiedostot toiminnoille, muunnoksille ja tilalle. Vuexin dokumentaatiosta löytyy esimerkki storen perustamisesta. Tämä on esitetty kuvassa 24. Kuvassa näkyy tilan muuttuja, jonka lähtöarvo on 0. Tilaa voidaan muuttaa muunnoksilla, joista kuvassa on annettu esimerkkinä ”increment (state)” -muunnos, joka kasvattaa tilan suuruutta yhdellä. Muunnosta kutsutaan samannimisellä toiminnolla (Vuex, n.d.-b). Esimerkistä poiketen työtä varten kirjaston toiminnalisuudet on pilkottu omiin tiedostoihin helpottaakseen luettavuutta.

Kuva 24. Näyttöleike Vuexin dokumentaation storen luomisesta (Vuex, n.d.-b).

CompositionAPI:lla kirjoitettaessa store tulee ottaa ensin käyttöön komponentin sisällä Vuex:in tarjoamalla ”useStore()”-hookilla. Tämän jälkeen komponentissa voidaan lukea storen tilaan määrittämällä se ”computed”-muuttujaksi, eli automaattisesti päivittyväksi muuttujaksi, jota ei pysty muuttamaan. Jos tilaa halutaan muuttaa, voidaan kutsua joko suoraan storen muunnosta käyttämällä komentoa ”store.commit()” tai kutsumalla toimintoa komennolla ”store.dispatch()”. API:lta haettaessa tietoa, halutaan käyttää toimintoja, eli tiedot päivitetään kutsumalla ”store.dispatch()”-komentoa, joka hakee tiedot API:lta ja tämän jälkeen kutsuu muunnosta, joka muuntaa tilan. Kun tila muuttuu, ”computed”-muuttuja automaattisesti päivittyy.

Kun store on luotu, voidaan testidata poistaa ja laittaa komponentti lukemaan storen tilaa. Koska käytössä on CompositionAPI ja <script setup> -tag, komponentin alkuun voidaan laittaa ”store.dispatch()”-komento, joka hakee palvelimen rajapinnasta tiedot. Kyseinen koodi suoritetaan kerran, kun komponentti luodaan. Jotta käyttöliittymä saadaan päivittämään tietoja, täytyy seuraavaksi luoda SignalR-yhteys.

4.6 SignalR-yhteyden luominen

SignalR-yhteyttä varten täytyy palvelimelle luoda ensin keskus ja konfiguroida yhteys. Keskus luodaan luomalla uusi luokka ja se perii SignalR:n Hub-luokan. Luotuun luokkaan lisätään haluttuja tehtäviä (*task*), joilla voidaan lähettää tietoa vastaanottajille. Kuvassa 25 on esitelty SignalR-kirjaston dokumentaatiosta löytyvä esimerkkikoodi keskuksen luomiseen. Siinä on luotu yksinkertainen keskus, jossa on yksi tehtävä viestien lähettämiseen kaikille vastaanottajille (*clients*). (Microsoft, 16.11.2023-m)

Kuva 25. Näyttöleike SignalR-kirjaston esimerkkikoodista keskuksen luontiin (Microsoft, 16.11.2023-m).

C# Copy

```
using Microsoft.AspNetCore.SignalR;

namespace SignalRChat.Hubs
{
    public class ChatHub : Hub
    {
        public async Task SendMessage(string user, string message)
        {
            await Clients.All.SendAsync("ReceiveMessage", user, message);
        }
    }
}
```

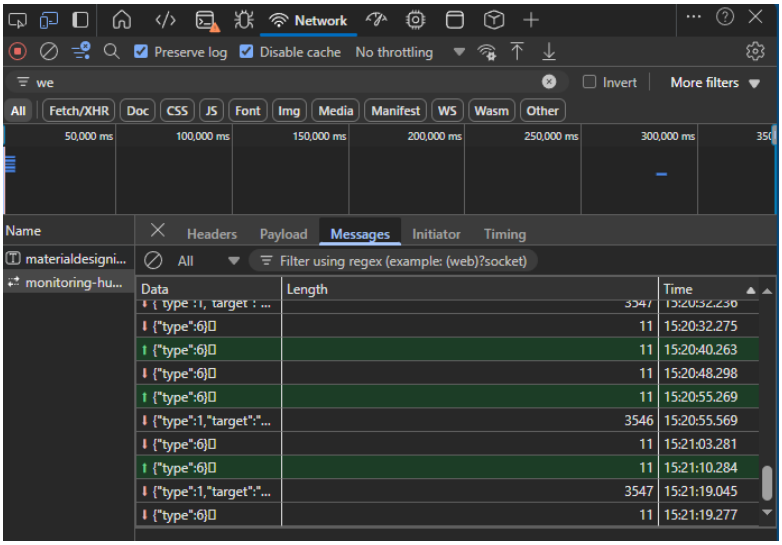
Jotta luotu keskus saadaan käyttöön, täytyy SignalR-palvelu alustaa projektin Program.cs-tiedostossa komennolla ”builder.Services.AddSignalR();”. Konfiguroinnissa määritellään lisäksi luotu keskus ja sen reitti komennolla ”app.MapHub<NimiHub>(”/reitti”)”. Tämän jälkeen keskus on toimintakunnossa. Jotta keskus lähettää tietoja vastaanottajalle, täytyy sitä lisäksi erikseen kutsua. Keskuksen tehtäviä voidaan kutsua eri puolelta palvelinta IHubContext-palvelulla. Käyttämällä IHubContext-instanssia, voidaan keskuksen tehtäviä kutsua vastaavasti kuin kuvassa 25 on esitettynä ”await SendAsync()”-metodi. (Microsoft, 18.6.2024-n)

Työtä varten halutaan lähettää tehtävä vastaanottajalle aina, kun laite päivittää tietoa tietokantaan. Tieto lähetetään samassa tiedonsiirto-oliomuodossa, kuin REST:n kautta tehdessä.

Käyttöliittymäpuolella täytyy määritellä SignalR-vastaanotin, joka keskustelee luodun keskuksen kanssa. SignalR:stä on olemassa valmis kirjasto JavaScriptille nimeltä ”@microsoft/signalr”. Työtä varten se on jo valmiiksi asennettuna, sillä käyttöliittymässä on jo toinen keskus käytössä. Vastaanotin tulee ottaa käyttöön moduulinäkymän avautuessa. Se voidaan ottaa käyttöön kirjaston tarjoamalla ”signalR.HubConnectionBuilder().withUrl().configureLogging(signalR.LogLevel.Information) .build();” -funktiolla, joka asetetaan muuttujaan ”connection”. Kun ”connection”-muuttuja on määritelty, voidaan yhteys vastaanottimen ja keskuksen välillä avata komennolla ”connection.on()”. Yhteyden avaus määritellään komponentin kiinnittyessä (*mount*), funktion ”onMounted()” sisällä, jotta yhteys on ehditty varmasti ensin luoda. ”connection.on()”-funktio ottaa vastaan kaksi muuttujaa, joihin ensimmäiseen annetaan keskuksen nimi stringinä ja toiseen haluttu funktio, jonka yhteys suorittaa aina uuden viestin saapuessa. Suoritettavaan funktioon halutaan laittaa aiemmin luodun storen mutaatio, jotta voimme päivittää tilan. Tämä voidaan tehdä ”store.commit(”store/commitFunction”, updatedValue)” -komennolla.

Tämän jälkeen yhteyden toiminta voidaan todentaa selaimen kehitystyökalujen avulla. Tiedonsiirto (*network*) -välilehdeltä nähdään käyttöliittymän luovan WebSocket-protokollayhteyden. Yhteyttä seuraamalla nähdään, että käyttöliittymä vastaanottaa viestejä keskukselta, jotka sisältävät halutun tiedon (kuva 26). Tästä voidaan todeta, että käyttöliittymä toimii ja tiedonsiirto toimii käyttöliittymän ja palvelimen välillä.

Kuva 26. Käyttöliittymän muodostama WebSocket-yhteys.



5 Yhteenveto

Työn toteutuksena valmistui ensimmäinen versio moduulista, joka lukee tuotantolaitteen päivittämää tietoa reaaliajassa ja näyttää sen mobiililaitteessa. Kun hybridisovelluksen näkymä avataan, lähettää se GET-kutsun palvelimelle, joka palauttaa sen hetkiset tiedot näkymälle. Samanaikaisesti muodostetaan SignalR:n kautta WebSocket-yhteys palvelimeen, jonka kautta näkymä saa uudet tiedot tuotantolaitteen päivittäessä tiedot tietokantaan. Toteutus lukee valualustoilla tapahtuvat työvaiheet, niiden tilan ja niihin käytetyn ajan. Tuotantolaitteiden yksilöllisiä tietoja ei vielä lueta näkymään. Näytettäviä valualustoja voidaan halutessaan filtteröidä tai piilottaa.

Toteutus vaatii vielä siistimistä ja jatkokehitystä. Käyttöliittymäpuolella moduuli on yhtenä tiedostona, joka voitaisiin refaktoroida pienempiin komponentteihin. Palvelinpuolella SignalR-yhteydestä puuttuu vielä autentikointi. Tämän lisäksi luotu GET-kutsu vaatii vielä jäsentelyä. Kutsussa tuodaan tällä hetkellä kaikki mahdollinen tieto mukana, mutta voi olla käytännön kannalta järkevämpi suodattaa turhia tietoja DTO:sta pois. Myös osa GET-kutsun tiedoista on vielä käyttöliittymän kannalta väärässä muodossa.

Toteutuksesta voidaan luoda .apk-tiedosto, joka mahdollistaa hybridisovelluksen ajamisen selaimen sijasta natiivisti mobiililaitteella. .apk:n luonti tapahtuu komennolla ”quasar build -m android”. Moduuli on kuitenkin kokonaisuudessaan edelleen keskeneräinen, ja hybridisovellus jää vielä kehitysvaiheeseen, jonka vuoksi .apk:ta ei ole tarvetta vielä luoda.

Lähteet

Amazon. (n.d.). *What's the Difference Between Web Apps, Native Apps, and Hybrid Apps?* Haettu 28.3.2025 osoitteesta https://aws.amazon.com/compare/the-difference-between-web-apps-native-apps-and-hybrid-apps/

Android. (n.d.). Android's *Kotlin-first approach.* Haettu 28.3.2025 osoitteesta https://developer.android.com/kotlin/first

Apple. (n.d.). *Swift.* Haettu 28.3.2025 osoitteesta https://developer.apple.com/swift/

Asharsaleem. (n.d.). *Long Polling vs Server-Sent Events vs WebSockets: A Comprehensive Guide.* [kuva] https://medium.com/@asharsaleem4/long-polling-vs-server-sent-events-vs-websockets-a-comprehensive-guide-fb27c8e610d0

Cordova. (n.d.). *Overview.* Haettu 16.3.2025 osoitteesta https://cordova.apache.org/docs/en/12.x-2025.01/guide/overview/index.html

Elematic Oyj. (10.2.2025). Annettu luonnos halutusta näkymästä [kuva].

Flutter. (n.d.). *Flutter for React Native developers.* Haettu 28.3.2025 osoitteesta https://docs.flutter.dev/get-started/flutter-for/react-native-devs

Github. (n.d.-a). *ColumnName.cs* [näyttöleike] https://github.com/dotnet/EntityFramework.Docs/blob/main/samples/core/Modeling/EntityProperties/FluentAPI/ColumnName.cs

Github. (n.d.-b). *NoForeignKey.cs* [näyttöleike] https://github.com/dotnet/EntityFramework.Docs/blob/main/samples/core/Modeling/Relationships/FluentAPI/NoForeignKey.cs

GraphQL. (n.d.). *GraphQL is the better REST.* Haettu 8.9.2025 osoitteesta https://www.howtographql.com/basics/1-graphql-is-the-better-rest/

IBM. (n.d.). *What is a REST API?* Haettu 19.3.2025 osoitteesta https://www.ibm.com/think/topics/rest-apis

Ihechikara, A. (21.10.2022). *What is an ORM – The Meaning of Object Relational Mapping Database Tools.* https://www.freecodecamp.org/news/what-is-an-orm-the-meaning-of-object-relational-mapping-database-tools/

Ionic. (n.d.-a). *Introduction to Ionic.* Haettu 28.3.2025 osoitteesta https://ionicframework.com/docs

Ionic. (n.d.-b). *Hi, we're Ionic.* Haettu 28.3.2025 osoitteesta https://ionic.io/about

JetBrains. (n.d.). *The Six Most Popular Cross-Platform App Development Frameworks.* Haettu 28.3.2025 osoitteesta https://www.jetbrains.com/help/kotlin-multiplatform-dev/cross-platform-frameworks.html

Jovanovic, M. (24.9.2022). *How to Approach CLEAN architecture folder structure.* https://www.milanjovanovic.tech/blog/clean-architecture-folder-structure

Khan, S. M. A. (n.d.). *Clean Architecture In ASP.NET Core Web API.* Haettu 19.3.2025 osoitteesta https://www.csharp.com/article/clean-architecture-in-asp-net-core-web-api/

Microsoft. (1.10.2024-a). *Introduction to .NET.* https://learn.microsoft.com/en-us/dotnet/core/introduction

Microsoft. (n.d-b). *ASP.NET Core.* Haettu 19.3.2025 osoitteesta https://dotnet.microsoft.com/en-us/apps/aspnet

Microsoft. (22.7.2022-c). *Entity Framework.* https://learn.microsoft.com/en-us/aspnet/entity-framework

Microsoft. (12.2.2024-d). *Overview of ASP.NET Core SignalR.* https://learn.microsoft.com/en-us/aspnet/core/signalr/introduction?view=aspnetcore-9.0

Microsoft. (30.3.2023-e). *Introduction to relationships.* https://learn.microsoft.com/en-us/ef/core/modeling/relationships

Microsoft. (28.3.2023-f). *Creating and Configuring a Model.* https://learn.microsoft.com/en-us/ef/core/modeling/

Microsoft. (12.1.2021-g). *How Queries Work.* https://learn.microsoft.com/en-us/ef/core/querying/how-query-works

Microsoft. (25.1.2024-h). *Single vs. Split Queries.* https://learn.microsoft.com/en-us/ef/core/querying/single-split-queries

Microsoft. (25.3.2025-i). *ASP.NET Core SignalR clients.* https://learn.microsoft.com/en-us/aspnet/core/signalr/client-features?view=aspnetcore-9.0

Microsoft. (11.3.2021-j). *Querying Data.* https://learn.microsoft.com/en-us/ef/core/querying/

Microsoft. (18.3.2023-k). *Interfaces – define behavior for multiple types*. https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/types/interfaces

Microsoft. (17.2.2025-l). *Tutorial: Create a controller-based web API with ASP.NET Core.* https://learn.microsoft.com/en-us/aspnet/core/tutorials/first-web-api?view=aspnetcore-9.0&tabs=visual-studio

Microsoft. (16.11.2023-m). *Tutorial: Get started with ASP.NET Core SignalR.* https://learn.microsoft.com/en-us/aspnet/core/tutorials/signalr?view=aspnetcore-9.0&tabs=visual-studio

Microsoft. (18.6.2024-n). *Send messages from outside a hub.* https://learn.microsoft.com/en-us/aspnet/core/signalr/hubcontext?view=aspnetcore-9.0

Mozilla. (n.d.). *WebTransport API.* Haettu 30.3.2025 osoitteesta https://developer.mozilla.org/en-US/docs/Web/API/WebTransport_API

Npm trends. (n.d.). Haettu 15.3.2025 osoitteesta https://npmtrends.com/@angular/cli-vs-react-vs-svelte-vs-vue

Obregon, A. (2.12.2023). *Real-Time Data Processing with Java: Frameworks and Strategies.* https://medium.com/@AlexanderObregon/real-time-data-processing-with-java-frameworks-and-strategies-04622fbda6d7

Pubnub. (26.9.2023). *What is Long Polling?* https://www.pubnub.com/guides/long-polling/

Quasar. (n.d.-a). *Pick a Quasar Flavour.* Haettu 16.3.2025 osoitteesta https://quasar.dev/start/pick-quasar-flavour

Quasar. (n.d.-b). *Expansion Item.* Haettu 23.3.2025 osoitteesta https://quasar.dev/vue-components/expansion-item

React Native. (n.d.). *React Native.* Haettu 28.3.2025 osoitteesta https://reactnative.dev/

Stoenescu, R. (3.6.2019). *Why every Vue developer should be excited by Quasar 1.0.* Medium. https://medium.com/quasar-framework/quasar-1-0-4bc696d60c1b

Socket.IO. (n.d.). *Introduction.* Haettu 30.3.2025 osoitteesta https://socket.io/docs/v4/

VueJS. (n.d.-a). *Frequently Asked Questions*. Haettu 15.3.2025 osoitteesta

https://vuejs.org/about/faq.html

VueJS. (n.d.-b). *Introduction.* Haettu 15.3.2025 osoitteesta https://vuejs.org/guide/introduction

VueJS. (n.d.-c). *Vue 2 end of life.* Haettu 15.3.2025 osoitteesta https://v2.vuejs.org/eol/

VueJS. (n.d.-d). *CompositionAPI faq.* Haettu 15.3.2025 osoitteesta https://vuejs.org/guide/extras/composition-api-faq.html

VueJS. (n.d.-e). *Composables: vs mixins.* Haettu 15.3.2025 osoitteesta https://vuejs.org/guide/reusability/composables.html#vs-mixins

VueRouter. (n.d.). *Getting started.* Haettu 23.3.2025 osoitteesta https://router.vuejs.org/guide/

Vuex. (n.d.-a) *What is Vuex?* Haettu 5.4.2025 osoitteesta https://vuex.vuejs.org/

Vuex. (n.d.-b) *Actions*. Haettu 5.4.2025 osoitteesta https://vuex.vuejs.org/guide/actions.html

WAC. (13.11.2024). *Top 11 Backend Programming Languages in 2025.* https://webandcrafts.com/blog/backend-languages