Haaga-Helia
ammattikorkeakoulu Oy

Sisäilmaa monitoroivan sovelluksen toteutus web-pohjaisena sovelluksena ASP.NET Core:lla

Anna Gunda

Opinnäytetyö

Tietojenkäsittelyn
koulutusohjelma

2021

Tiivistelmä

Haaga-Helia
ammattikorkeakoulu Oy

Tekijä

Anna Gunda

Koulutusohjelma

Tietojenkäsittelyn koulutusohjelma

Opinnäytetyön nimi
Sisäilmaa monitoroivan sovelluksen toteutus web-pohjaisena sovelluksena ASP.NET Core:lla

Sivu- ja liitesivumäärä

54

Opinnäytetyöprojektin puitteissa toteutettiin sisäilman olosuhteita monitoroiva web-pohjainen sovellus. Projektin tuloksena syntyi ratkaisu, joka olosuhteiden monitoroinnin lisäksi mahdollistaa hälytysten luomisen ja lähettää käyttäjille notifikaatiot, mikäli huoneistossa havaitaan poikkeukselliset lämpötilan tai kosteusasteen lukemat, sekä ohjaa huoneistossa asetetun ilmalämpöpumpun toimintaa. Sovellus koostuu kolmesta loogisesta kokonaisuudesta: sisäilman monitoroinnin moduulista, hälytysmoduulista ja ilmalämpöpumpun automaattisen ohjauksen moduulista.

Opinnäytetyön tyypiksi valittiin toiminnallinen opinnäytetyö. Opinnäytetyöprojektin päätavoitteena oli konkreettinen tuotos eli varsinainen sovellus. Toimivan sovelluksen tuottamisen lisäksi opinnäytetyöprojektin tavoitteina oli syventää ammatillista osaamista ja oppia käyttämään valittuja teknologioita, harjoitella kokonaisuuden rakentamista sovelluksen liiketoiminta-, tietokantaoperaatiot- ja näkymäkerrokset toisistaan erottaen sekä saada syvempi ymmärrys palvelujen integroinneista ja yhteistoimivuudesta API-rajapinnan kautta.

Projektin teknisessä toteutuksessa käytettiin opinnäytetyön kirjoittamisen hetkellä laajasti käytössä olleita Microsoftin teknologioita. Sovellus toteutettiin C#-kielellä ja ajoympäristöksi valittiin ASP.NET Core. ASP.NET Core-komponenteista sovelluksessa laajasti käytettiin ASP.NET MVC -kehystä ja Entity Framework Core -olio-relaatio-mallinnus -työkalua (Object Relational Mapper). Käyttöliittymät generoidaan käyttäen Razor -näkymämoottoria. Selainpään käyttöliittymä hyödyntää HTML, CSS ja JavaScript -teknologioita.

Opinnäytetyön tarkoituksena ei ollut suorittaa markkinatutkimus olemassa olevista tarpeisiin soveltuvista sovellusratkaisuista eikä verrata keskenään sovelluksen kehityksessä tarvittavia teknologioita. Projekti on rajattu henkilökohtaisiin tarpeisiin soveltuvan web-sovelluksen toteutukseen valittuja teknologioita käyttäen.

Asiasanat

ASP.NET Core, Entity Framework Core, MVC, web-sovellus, sisäilman monitorointi

Sisällys

1 Johdanto

Opinnäytetyöprojektin puitteissa toteutettiin sisäilman olosuhteita monitoroiva web-pohjainen sovellus. Projektin tuloksena syntyi ratkaisu, joka olosuhteiden monitoroinnin lisäksi lähettää käyttäjille notifikaatiot, mikäli huoneistossa havaitaan poikkeukselliset lämpötilan tai kosteusasteen lukemat, sekä ohjaa huoneistossa asetetun ilmalämpöpumpun toimintaa.

Opinnäytetyöprojektin aikana keskityttiin sopivien teknologien valintaan ja sovelluksen tekniseen toteutukseen. Toteutettu sovellus koostuu kolmesta loogisesta kokonaisuudesta: sisäilman monitoroinnin moduulista, hälytysmoduulista ja ilmalämpöpumpun automaattisen ohjauksen moduulista. Sisäilman monitoroinnin moduulissa viimeisimpien mittaustulosten esittämisen lisäksi päätettiin toteuttaa mahdollisuus seurata lämpötilan ja kosteuden seitsemän vuorokauden muutostrendiä.

Projektin teknisessä toteutuksessa käytettiin opinnäytetyön kirjoittamisen hetkellä laajasti käytössä olleita Microsoftin teknologioita. Käytettyjä teknologioita käsitellään erikseen tämän opinnäytetyön sisällössä.

Henkilökohtainen tarve sisäilman olosuhteita monitoroivalle sovellukselle nousi arkipäiväisestä ongelmasta. Ilman kylmetessä ja sisäilman kosteusasteen laskiessa perheemme yöunet jatkuvasti keskeytyivät lapsemme nenäverenvuotoihin. Pitkän seurantajakson jälkeen havaittiin yhteys nenäverenvuotojen ja matalan ilmankosteuden välillä.

Ratkaisuna ongelmaan ostettiin ilmankostutin, joka ruvettiin laittamaan iltaisin lapsen huoneeseen. Ongelma ei kuitenkaan ratkennut ilmankostuttimen hankkimisella, sillä usein ilmankostutin unohtui pitkäksi aikaa päälle ja huoneilman kosteus pääsi nousemaan liian korkealle. Seuraavana toimenpiteenä jokaisen huoneeseen asennettiin RuuviTag -anturi, joka kirjasi sisäilman olosuhteiden lukemat tietokantaan minuutin välein. Tehtyyn ratkaisuun tarvittiin räätälöity sovellus, joka pääsisi näyttämään tietokantaan kerättyä dataa ja suorittamaan datan pohjalta tarvittavat toimenpiteet käyttäjien toiveiden mukaisesti. Tästä lähti liikkeelle omiin tarpeisiin räätälöidyn sovelluksen suunnittelu ja kehitys. Sovellukseen päätettiin integroida myös huoneistossa asetettu ilmalämpöpumppu, joka mahdollistaisi sisäilman lämpötilan automaattisen korjauksen asetettujen tavoitelukemien mukaisesti.

Opinnäytetyöprojektin puitteissa lähdettiin tutkimaan mahdollisia huonoon sisäilman laatuun liittyviä ongelmia kirjallisuudesta ja verkkolähteistä, minkä kautta saatiin todisteita sopivien olosuhteiden ylläpitämisen tärkeydelle. Luettuihin lähteisiin pohjautuen todettiin, että sisäilman laatu vaikuttaa ihmisen hyvinvointiin ja elämänlaatuun. Optimaaliset olosuhteet edistävät ihmisen hyvinvointia. Liian matalat tai liian korkeat lämpötilat sekä liian kuiva tai liian kostea ilma voivat aiheuttaa tai pahentaa ihmisen terveysongelmia.

Sisäilman kuivuus aiheuttaa hengitysteiden, limakalvojen ja ihon ärsytystä sekä lisää limakalvojen tulehdusriskiä. Liian kostea sisäilma voi puolestaan aiheuttaa home- ja kosteusvaurioita huoneistoissa ja sillä aiheuttaa vakavia haittoja ihmisen terveydelle ja hyvinvoinnille. Väsymysoireet, allergia- ja astmaoireet voivat myös johtua huonosta sisäilman laadusta. (Hengitysliitto.)

Huoneilman kuivuudesta johtuen, hengitysteiden värekarvojen liikkeet voivat hidastua, minkä seurauksena ilman poistuminen hengitysteistä heikkenee ja limakalvojen kyky vastustaa tulehduksia vähenee. Oireiden ilmestyessä tai pahetessa talviaikaan suositellaan kostuttamaan huoneilmaa tai laskemaan lämpötilaa huoneessa. (Valvira 2016, 12.)

Pitkäkestoiset liian korkeat kosteuspitoisuudet asuintiloissa nostavat huomattavasti kosteusvaurioiden riskiä. Suomessa joka vuosi kymmeniä tuhansia ihmisiä saavat sisätilojen kosteusvaurioista aiheutuvia oireita ja noin 800 ihmistä sairastuu astmaan kosteusvaurioiden takia. (Lampi & Pekkanen 2008, 23.)

Saatuamme tarpeeksi tieteellisiä todisteita optimaalisen lämpötilan ja kosteusasteen ylläpitämisen tärkeydestä, ruvettiin suunnittelemaan ja kehittämään sovellusta ja kirjoittamaan opinnäytetyötä. Opinnäytetyön tyypiksi valittiin toiminnallinen opinnäytetyö. Opinnäytetyöprojektin päätavoitteena oli konkreettinen tuotos eli varsinainen sovellus.

Toimivan sovelluksen tuottamisen lisäksi opinnäytetyöprojektin päätavoitteina oli syventää ammatillista osaamista ja oppia käyttämään valittuja teknologioita, harjoitella kokonaisuuden rakentamista sovelluksen liiketoiminta-, tietokantaoperaatiot- ja näkymäkerrokset toisistaan erottaen sekä saada syvempi ymmärrys eri toimittajien palvelujen integroinneista ja yhteistoimivuudesta API-rajapinnan kautta.

Opinnäytetyön tarkoituksena ei ollut suorittaa markkinatutkimus olemassa olevista tarpeisiin soveltuvista sovellusratkaisuista eikä verrata keskenään sovelluksen kehityksessä tarvittavia teknologioita. Projekti on rajattu henkilökohtaisiin tarpeisiin soveltuvan web-sovelluksen toteutukseen valittuja teknologioita käyttäen. Projekti ei kata

teknistä osuutta, joka vastaa RuuviTag -antureiden asennuksesta ja integroinnista sekä antureiden lukemien tallennuksesta tietokantaan. Opinnäytetyöprojektin ulkopuolelle on rajattu myös toteutetun sovelluksen testaus ja käyttöönotto.

2 Käytetyt teknologiat

Tässä luvussa tutustutaan opinnäytetyön keskeisimpiin käsitteisiin ja käsitellään sovelluksessa käytettyjä teknologioita.

2.1 Käsitteet

.NET Core

.NET Core on ilmainen, avoimen lähdekoodin ohjelmistokehys Windows, Linux sekä MacOS käyttöjärjestelmille. .NET Core:n seuraaja virallisesti kulkee nimellä .NET 5. (Wikipedia.)

ASP.NET Core

ASP.NET Core on Microsoftin alustariippumaton avoimeen lähdekoodiin perustuva Web-sovelluksien ohjelmistokehys ja ajoympäristö. ASP.NET Core ajetaan .NET Core:n päällä. (Smith 2020.)

MVC

MVC on ohjelmistoarkkitehtuuri, jossa sovellus jaetaan kolmeen komponenttikokonaisuuteen: mallit (models), näkymät (views) ja ohjaimet (controllers). Jaon päätarkoituksena on erottaa toisistaan erilaiset loogiset tehtäväryhmät. (Smith 2020.)

MVC on lyhenne englanninkielisistä termeistä "Model", "View", "Controller".

ASP.NET Core MVC

ASP.NET Core MVC on kevyt, avoimeen lähdekoodiin perustuva ohjelmistokehys, joka on optimoitu käytettäväksi ASP.NET Core:n kanssa. ASP.NET Core MVC:lla kehitetään MVC-ohjelmistoarkkitehtuuriin pohjautuvia dynaamisia verkkosivuja. (Smith 2020.)

LINQ

LINQ (Language-Integrated Query) on C# kieleen integroitu kyselykieli, joka mahdollistaa SQL -kaltaisten kyselyiden kirjoittamisen suoraan C# -koodiin (Microsoft 2016).

Entity Framework Core

Entity Framework Core on Microsoft:n kehittämä olio-relaatio-mallinnustyökalu (Object Relational Mapper), joka esittää relaatiotietokannan dataa .NET olioina / entiteetteina (Freeman 2020, 39). Kyselyt kirjoitetaan käyttämällä LINQ:ta.

Väliohjelmisto (middleware)

ASP.NET Core sovelluksessa väliohjelmisto on sovellus tai sovelluksen osa, joka laajentaa ASP.NET Core alustan perusominaisuuksia. Väliohjelmistot istuvat/toimivat ASP.NET Core:n päällä, ja tarjoavat erilaisia palveluja sovelluksille. (Anderson & Smith 2020.)

Request pipeline - ASP.NET

Pyynnön käsittelyketju (Heikniemi 9.4.2021).

ASP.NET Core:ssa "request pipeline" tarkoittaa loogista kokonaisuutta, joka kuvaa sisääntulevan pyynnön käsittelyn eri vaiheita: mm reititys, validointi, malli-bindaus, autentikaatio, käyttöoikeudet, lokitus, virhekäsittely jne (Freeman 2020, 450).

REST

(Representational state transfer). REST on sovellusarkkitehtuuri, joka hyväksikäyttää HTTP protokollaa (Wikipedia).

REST API

REST API on ohjelmointirajapinta, jota on tarkoitus käyttää REST -arkkitehtuuria vastaavalla tavalla (Wikipedia).

Singleton

Singleton on olio-ohjelmoinnin toteutustapa, jonka avulla huolehditaan siitä, että singleton -luokasta voidaan luoda vain yhden instanssin koko ohjelman elinkaaren aikana (Gamma, Helm, Johnson & Vlissides 2005).

2.2 Teknologiavalinnat ja perustelut valinnoille

Sovellus toteutettiin C#-ohjelmointikielellä ja ajoympäristöksi valittiin ASP.NET Core. ASP.NET Core-komponenteista sovelluksessa laajasti käytettiin ASP.NET MVC -kehikkoa ja Entity Framework Core -olio-relaatio-mallinnus -työkalua (Object Relational Mapper).

Sovelluksen toteutukseen haluttiin valita vahvasti tyypitetty kieli, jolla voi tehdä olio-ohjelmointia, ja valittiin C#. C# on olio-ohjelmointiin soveltuva moderni vahvasti tyypitetty ohjelmointikieli, joka on laajassa käytössä ja jatkuvassa kehityksessä.

Ajoympäristön valinnassa tärkeimpiä kriteereitä olivat laaja saatavuus ja laaja käyttö, alustariippumattomuus sekä suorituskyky. ASP.NET Core-ajoympäristö valittiin sen takia, että se täyttää kaikki yllä mainitut kriteerit. (Anderson, Luttin & Roth 2020.)

ASP.NET Core on Microsoftin ylläpitämä, mutta kuitenkin avoimeen lähdekoodiin perustuva ja alustariippumaton ajoympäristö ohjelmille. ASP.NET Core soveltuu myös konteissa ajoon. Sovelluksen tekemisen ja opinnäytetyön kirjoittamisen hetkellä .NET Core on laajasti käytössä oleva alusta, jota usein käytetään yritys-sovellusten tekemisessä. .NET Core:a alusta asti kehitettiin avoimen lähdekoodin keinoin. Tämä tarkoittaa sitä, että kuka tahansa voi tuoda uusia ominaisuuksia sekä C# -kääntäjään, että .NET Core -ajoympäristöön Pull Request:ien kautta. (Anderson, Luttin & Roth 2020.)

.NET Core:n käyttö on todella laaja. Sitä käytetään mobiiliohjelmien kehityksessä, pelien kehityksessä ja erilaisten ohjelmien tekemisessä web-sovelluksissa. .NET Core:n käytön kokemusta voi myöhemmin hyödyntää, jos haluaa siirtyä web-sovellusten tekemisestä vaikka pelien tai mobiiliohjelmien tekemiseen. Siinä tapauksessa uuden kielen ja .NET peruskirjaston käytön opettelua ei tarvitse. ASP.NET Core sisältää laajan tuen Web sovelluskehityksen eri osakokonaisuuksien hallintaan. Se soveltuu sekä dynaamisten Web sovellusten, SPA sovellusten (Single Page Application) sekä API sovellusten kehitykseen. (Anderson, Luttin & Roth 2020.)

.NET Core pyörii kaikilla alustoilla. Web sovellusten kannalta tärkeimmät alustat ovat kuitenkin Windows ja Linux. .NET Core -ohjelmat voidaan kääntää suoraan kohdealustalla pyöriväksi natiivikoodiksi. Näin saadaan paras suorituskyky, eikä interpretaatio tai just-in-time (JIT) kääntäminen hidasta ohjelman käynnistysaikaa tai suorituskykyä. .NET Core:n suorituskykyä on parannettu jokaisessa versiossa. (Toub 2020.)

Samalla ASP.NET Core ajoympäristö on ajoympäristönä robusti ja nopea. Sisäänrakennettu Kestrel web palvelimen avulla voidaan ajaa sovellukset suoraan myös

tuotannossa (käyttäen reverse-proxy tekniikkaa). Näin ollen sekä kehityksen, testauksen ja tuotannon aikana voimme käyttää samaa Web-ajoympäristöä. (Anderson, Luttin & Roth 2020.)

Vaikka suorat tietokantakyselyt ovat myös mahdollisia .NET Core ympäristössä, ottamalla käyttöön Entity Framework Core:n (EF Core) saavutetaan myös riippumattomuus tietokantakerroksesta. Käyttämällä modernia Object Relational Mapper (ORM) tuotetta kuten EF Core, voidaan koodia kirjoittamalla toteuttaa sovelluksen tietokantamallia, riippumatta varsinaisesta tietokantakerroksesta. Tietokantakerrosta voidaan vaihtaa vaikka jokaisessa ajoympäristössä eri tietokantaan. EF Core huolehtii yhteyden muodostuksesta sekä tietojen mappauksesta olio- ja relaatiomaailmoiden välissä.

2.3 ASP.NET Core

ASP.NET Core on Microsoftin alustariippumaton (cross-platform), avoimen lähdekoodin (open-source) kehikko, jolla voidaan kehittää nykyaikaisia pilvipohjaisia web-sovelluksia (Smith 2020.). ASP.NET Core koostuu alustasta (platform), joka on tarkoitettu HTTP-pyyntöjen käsittelyyn, ja kaikista tarvittavista työkaluista web-sovelluksen kehittämiseen. ASP.NET Core alusta (platform) toimii pohjana, jonka päällä sovellukset rakennetaan, ja tarjoaa kaikki ominaisuudet HTTP -pyyntöjen käsittelyyn. (Freeman 2020, 446.)

ASP.NET Core alusta (platform) hoitaa web-sovellusten matalan tason toimintojen toimivuuden ja antaa kehittäjille mahdollisuuden keskittyä ominaisuuksien kehittämiseen loppukäyttäjiä varten. ASP.NET Core:a käyttäessä kehittäjä ei välttämättä joudu käsittelemään platform-tason toimintoja. Siitä huolimatta juuri platform-tason toiminnot toimivat sovelluksen perustana ja takaavat korkeampien tasojen komponenttien, kehysten ja työkalujen toimivuuden (Smith 2020). ASP.NET Core Platform -tason keskeisimmät rakennuspalikat ovat palvelut (services) ja väliohjelmistokomponentit (middleware components), jotka molemmat voi konfiguroida `Startup.cs` -luokassa. (Freeman 2020, 446.)

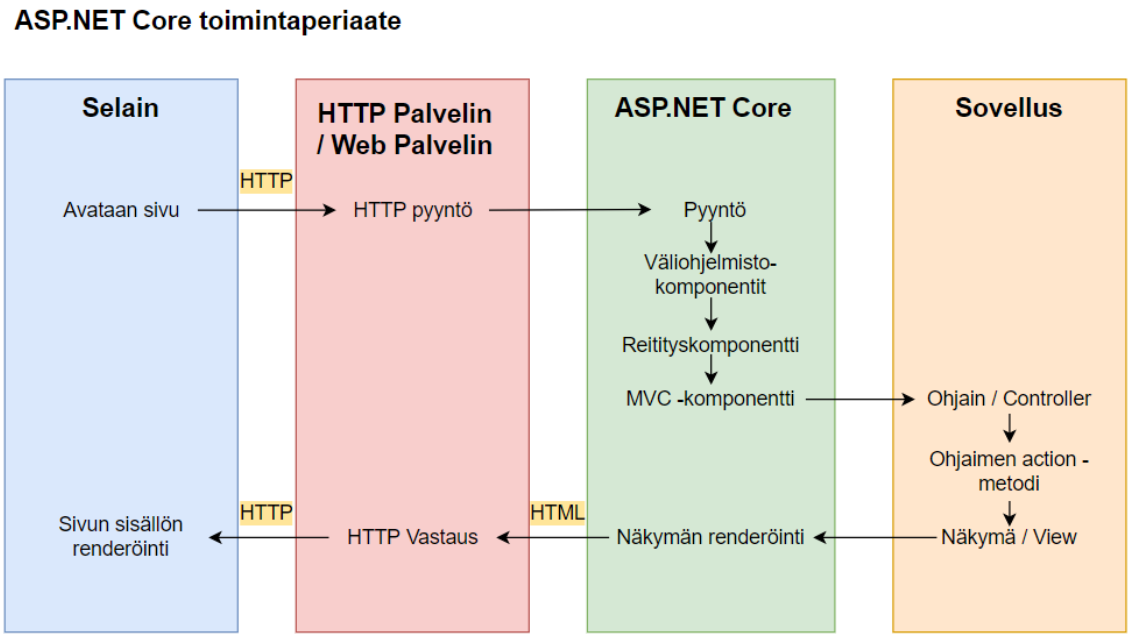
ASP.NET Core:ssa web-sovellusta ja sen ajoympäristöä suurimmalta osin alustetaan ja konfiguroidaan konvention mukaan Startup.cs -tiedostossa. Projektin luomisvaiheessa Startup -luokkaan automaatttisesti lisätään kaksi oletusmetodia: `Configure` -metodi ja `ConfigureServices` -metodi.

`ConfigureServices` – metodissa määritellään palvelut (services), joita sovellus tarvitsee. Palvelut ovat objekteja, jotka ovat jaettu läpi sovelluksen. Niiden käsiksi pääsee

käyttämällä riippuvuusinjektio (dependency injection) -ominaisuutta. Esimerkiksi `ConfigureServices` – metodissa kutsuttu `AddControllersWithViews` -metodi alustaa jaetut objektit, jotka tarvitaan MVC Framework:ia ja Razor-moottoria käyttävissä sovelluksissa. (Freeman 2020, 236.)

ASP.NET Core saa HTTP-pyynnöt ja välittää/kuljettaa ne kutsuputken (request pipeline) kautta, joka koostuu Configure -metodissa rekisteröidyistä väliohjelmistokomponenteista. Jokainen väliohjelmistokomponentti pystyy tutkimaan kutsua, muokata kutsua, luoda vastauksen tai muokata muiden komponenttien luomia vastauksia. (Freeman 2020, 236.)

Configure -metodissa rekisteröidään kaikki väliohjelmistokomponentit (middleware components) joita putki (pipeline) tulee käyttämään. Tyypillisesti projektin luomisvaiheessa pipeline:een lisätään kolme komponenttia: `UseDeveloperExceptionPage`, `UseRouting`, `UseEndpoints`. `UseDeveloperExceptionPage` -metodi lisää väliohjelmistokomponentin, joka vastaa käsittelemättömien poikkeusten tietojen välittämisestä. `UseRouting` -metodi lisää putkeen reitityskomponentin, joka vastaa pyyntöjen käsittelystä. `UseEndpoints` -metodi kuvaa konfiguraation `UseRouting` -metodin lisäämälle komponentille. (Freeman 2020, 453 – 454.) Kuva 1:ssa on esitetty ASP.NET Core toimintaperiaate yksinkertaistettuna.



Kuva 1. ASP.NET Core toimintaperiaate yksinkertaistettuna

ASP.NET Core -sovelluksissa kaikki sisääntulevat pyynnöt joutuvat päätepisteiden (endpoints) käsiteltäviksi. Päätepiste on toiminto (action), joka on C# -kielellä kirjoitettu

metodi. Toiminnot on määritelty ohjaimessa (controller), joka on `Microsoft.AspNetCore.Mvc.Controller` – sisäänrakennetusta perusluokasta periytyvä C# -luokka. (Freeman 2020, 65.)

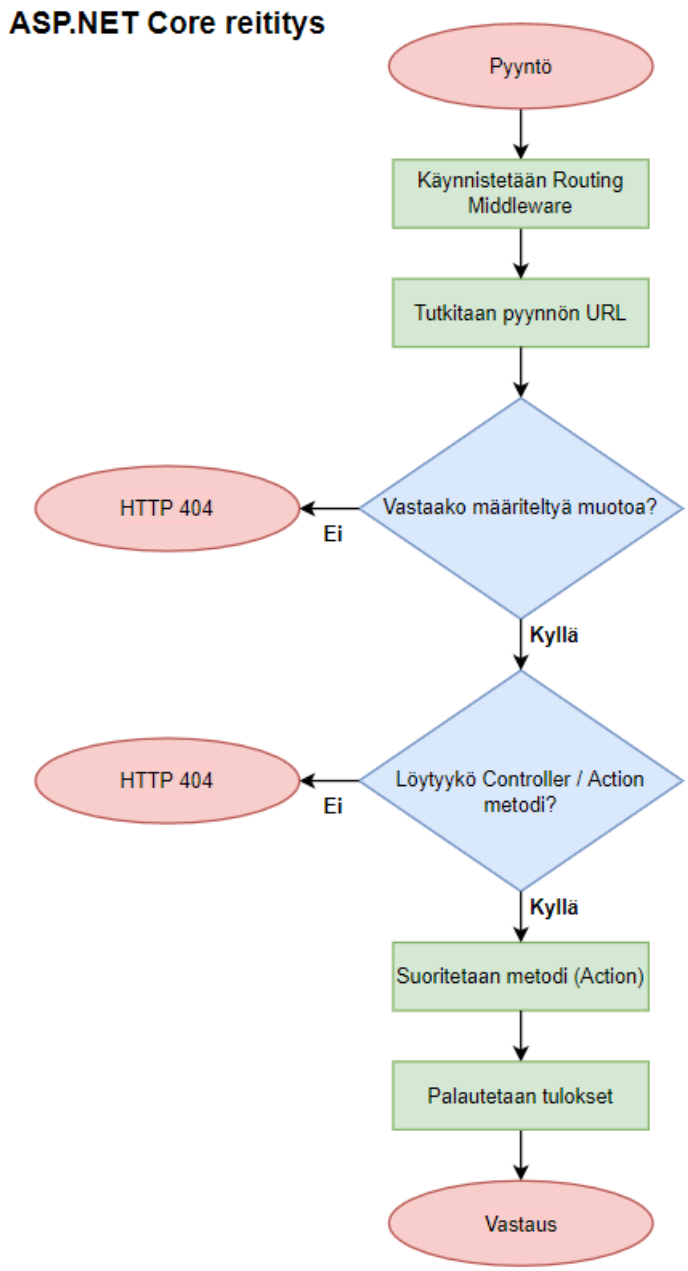
Jokainen ohjaimessa oleva julkinen metodi on toiminto (action), mikä tarkoittaa, että metodia voi määrittää käsittelemään HTTP-pyyntöä. Konvention mukaan ASP.NET Core projekteissa kaikki ohjainluokat (controller class) tallennetaan projektin luomisvaiheessa oletuksena luotuun `Controllers` -nimiseen hakemistoon. Ohjainluokkien konvention mukainen nimeämismalli on varsinainen ohjainluokan nimi, jota seuraa `Controller` -sana. Esimerkiksi `HomeController.cs` -tiedoston nimestä tietää, että se sisältää `Home` -nimisen ohjaimen, joka on ASP.NET Core -projektien oletusohjain. (Freeman 2020, 65.)

ASP.NET Core:n reititys (engl. routing) on vahva URL-mappauskomponentti, joka sallii älykkäiden ja helposti löydettävien url:ien käyttöä sovelluksessa. ASP.NET Core:n reititys mahdollistaa hakukoneoptimoidut url:ien nimeämiskäytännöt. ASP.NET Core:n reitityksen ansiosta url:ien toimivuuteen ei vaikuta hakemistojen ja tiedostojen sijainti palvelimella. Reitit voidaan määrittää käyttämällä reititysmallia. (Smith 2020.)

ASP.NET Core:n reititysjärjestelmä vastaa sen päätepisteen valinnasta, joka tulee käsittelemään HTTP -pyynnön. Reittiä voi ajatella sääntönä, jota sovellus käyttää päättämään, miten käsitellä sisääntulevat pyynnöt. Reitityksestä vastaava väliohjelmisto (routing middleware) prosessoi URL:ia, tutkii reittejä ja löytää päätepisteen, joka tulee käsittelemään pyynnön. Reitityksestä vastaava väliohjelmisto lisätään putkeen (pipeline) `Startup.cs` -tiedoston `Configure` -metodissa käyttämällä kahta erillistä metodia `UseRouting` ja `UseEndpoints`. (Freeman 2020, 494.)

Konventioon perustuva reititys (convention-based routing) määrittää sovelluksen hyväksymät url-muodot. Reityksen perustehtävä on ohjata käyttäjän lähettämä pyyntö url:in perusteella valitulle ohjaimelle (engl. controller) ja ohjaimen toiminnolle (engl. action). Käytännössä se tapahtuu niin, että reititysmoottori parsii url:n, vertaa saatua URL:ia sovelluksessa määriteltyihin URL-formaatteihin ja sopivan formaatin löydettyä kutsuu vastaavan ohjaimen vastaavaa toimintoa (Kuva 2). (Smith 2020.)

Attribuutteihin perustuvassa reitityskäytännössä (attribute routing) on mahdollista määrittää sovelluksen reititystä varustamalla ohjaimet ja toiminnot attribuuteilla, joiden mukaan reititys sovelluksessa tulee tapahtumaan.



Kuva 2. ASP.NET Core reititys

2.4 MVC ohjelmistoarkkitehtuuri yleisesti

Model - View - Controller (MVC) on ohjelmistoarkkitehtuuri, johon nykyään usein pohjaudutaan sovellusten suunnittelussa ja toteutuksessa. MVC - ohjelmistoarkkitehtuurissa sovellus jaetaan kolmeen komponenttikokonaisuuteen: mallit (models), näkymät (views) ja ohjaimet (controllers). Jaon päätarkoituksena on erottaa toisistaan erilaiset loogiset tehtäväryhmät. (Smith 2020.)

Käyttäjän pyynnöt ohjataan ohjaimelle (controller), joka puolestaan käsittelee malleja (model). Mallit esittävät tietokannassa olevaa dataa ja sovelluksen liiketoimintalogiikkaa. Tiedot tallennetaan tietokantaan, haetaan tietokannasta ja käsitellään sovelluksessa mallien kautta. Ohjain valitsee näkymän, joka palvelee käyttäjän pyyntöjä, ja välittää valitulle näkymälle tarvittavan mallin dataa.

Ohjaimet ovat komponentteja, jotka käsittelevät käyttäjän pyyntöjä, sovelluksen malleja ja määrittävät käyttäjälle esitettävä näkymä. Näkymät vastaavat datan esittämisestä käyttäjälle käyttöliittymän kautta. MVC-ohjelmistoarkkitehtuurilla toteutetussa sovelluksessa malli edustaa sovelluksen tilaa sekä sovelluksen koko liiketoimintalogiikkaa ja siihen liittyvät toiminnot. Jos käyttöliittymässä halutaan esittää monimutkaisen mallin dataa, sovelluksessa kannattaa ottaa käyttöön ylimääräinen logiikkakerros: esimerkiksi näkymämalli (ViewModel). (Smith 2020.)

Vahvasti tyypitetyt näkymät eli näkymät, jotka ovat varustettu @model -direktiivillä, käyttävät niin sanottuja näkymän malleja (view model). Näkymän mallin tarkoitus on kuvata tiedot, jotka tulee näyttää valitussa näkymässä. Ohjain luo ja täyttää näkymän mallin instanssit välitetyn mallin perusteella.

Toteuttamalla sovellusta MVC-ohjelmistoarkkitehtuurilla voi helpottaa sovelluksen ylläpitoprosessia. On helpompi kirjoittaa, debuggata ja testata kokonaisuuksia (malli, näkymä tai ohjain), joista jokaisella on oma erillinen tarkoitus ja tehtävä. Vastaavasti on vaikeampi päivittää, testata ja debuggata sovellusta, jonka riippuvuudet ulottuvat kahdelle tai kolmelle ym. kokonaisuudelle. Yleensä käyttöliittymäkerros vaatii muutoksia useammin, kuin sovelluksen liiketoimintalogiikkakerros. Jos näkymä- ja liiketoimintalogiikka ovat yhdistetty samassa oliossa, jokainen käyttöliittymämuutos tulee vaatimaan vastaavasti muutoksia liiketoimintalogiikkaan. Tämmöinen toteutustapa on virhealtis ja vaatii perusteellista liiketoimintalogiikkakerroksen testausta pienenkin käyttöliittymämuutoksen yhteydessä. (Smith 2020.)

Lisäämällä näkymämalli (ViewModel) -abstraktiotason luokat sovelluksen käyttöliittymää voi rakentaa täysin ilman, että käyttää tietokantaa. Siinä tapauksessa tietokannan voi kytkeä sovellukseen myöhemmin ja luomalla yhteyden tietokannan ja ViewModel -luokkien välille käyttöliittymä saa näkyville tietokantadataa.

Käyttöliittymä on helpompi pitää pysyvänä ja muuttumattomana silloin, kun käyttöliittymän toimivuus pohjautuu ViewModel -luokkiin. Käytännössä se tarkoittaa sitä, että käyttöliittymä tulee toimimaan myös siinä tapauksessa, jos tietokanta muuttuu. Tällöin

muutoksia tarvitaan mallitason (Model) ja näkymämallitason (ViewModel) välille (pitää muuttaa mappaus tietokannan ja näkymämallien välillä), mutta muutoksia käyttöliittymään ei tarvitse tehdä.

2.5 ASP.NET Core MVC

ASP.NET Core MVC on kevyt, avoimen lähdekoodin ohjelmistokehys, joka on optimoitu käytettäväksi ASP.NET Core:n kanssa. ASP.NET Core MVC:lla kehitetään MVC-ohjelmistoarkkitehtuuriin pohjautuvia dynaamisia verkkosivuja, joissa on mahdollistettu tarkka huolenaiheiden erottaminen (engl. separation of concerns). (Smith 2020.)

ASP.NET Core MVC koostuu seuraavista komponenteista: reititys (routing), Model binding, mallivalidointi (model validation), riippuvuuksien rekisteröinti (dependency injection), suodattimet (filters), alueet (areas), web-rajapinnat (web APIs), testattavuus (testability), Razor -näkymämoottori (Razor view engine), vahvasti tyypitetyt näkymät (strongly typed views), Tag Helpers, näkymä -komponentit (View Components) (Smith, 2020). ASP.NET Core MVC on rakennettu ASP.NET Core:n reitityksen päälle, josta tarkemmin on kerrottu opinnäytetyön kohdassa ”ASP.NET Core Routing”.

Mallin sitomisen avulla ASP.NET Core MVC rakentaa käyttäjältä saatujen pyyntöparametrien avulla olioita. Käyttäjältä saatujen pyyntöparametrien lähteet ovat lomakkeiden arvot (form values), reittitiedot (route data), url-parametrit (query string parameters), HTTP -otsikot (HTTP -headers). Ohjaimet saavat kaikki tarvittavat tiedot toimintojen parametreina. (Smith 2020.)

Mallin sitominen toimii ”siltana” HTTP-pyynnöstä toiminto -metodiin. Oletuksena mallin sitomisen järjestelmä etsii data-arvot seuraavasta neljästä kohdasta: lomakkeen tiedoista (form data), pyynnön rungosta (request body), reittiin sisältyvistä muuttujista (routing segment variables) ja kyselymerkkijonoista (query strings). Data-arvot etsitään pyynnön rungosta vain niiden ohjainten kohdalla, jotka ovat varustettu ApiController -attribuutilla. Jokaista tietolähdettä tutkitaan vuorotelleen edellä kuvatussa järjestyksessä, kunnes argumentin arvo löytyy. Sopivan data-arvon löydettyä haku loppuu eikä seuraavia kohtia tutkita. Esimerkiksi jos sopiva data-arvo löytyy reittimuuttujasta, kyselymerkkijonoja ei tutkita. (Freeman 2020, 1183.)

Mallin tila pohjautuu kahteen osajärjestelmään: mallin sitomiseen ja mallivalidointiin. Mallin sitomisessa tapahtuvat virhetilat ovat yleensä tietojen muuntamisesta johtuvat virheet. Esimerkiksi integer -tyyppistä tietoa vaativaan kenttään syötetään string -tyyppinen arvo. Mallivalidointi tapahtuu mallin sitomisen jälkeen ja mallivalidointivirheet

johtuvat yleensä siitä, että syötetyt arvot eivät vastaa liiketoimintalogiikan sääntöjä. Mallin sitominen ja mallivalidointi tapahtuvat ennen ohjaimen toiminnon suorittamista. (Larkin 2019.)

Validointisäännöt mallin kentille määritellään validointiattribuuttien avulla. Mallivalidoinnissa voi käyttää joko sisäänrakennettuja validointiattribuutteja tai niitä voi luoda itse (custom validation attribute). Käytetyimpiä sisäänrakennettuja attribuutteja ovat muun muassa `[EmailAddress]`, `[Phone]`, `[Required]`, `[RegularExpression]`, `[StringLength]`.

Web-sovelluksissa mallin tila voidaan tarkistaa ehtolauseella `if (!ModelState.IsValid) {...}`, jossa määritetään myös, miten sovelluksen kuuluu reagoida mallivalidointivirheisiin.

ASP.NET Core MVC käyttää Razor -näkymämoottoria (view engine) näkymien renderöintiin. Razor näkymämoottori generoi HTML -vastaukset prosessoimalla .cshtml -päätteiset tiedostot. Razor -syntaksi mahdollistaa C# -kielellä kirjoitettujen koodipätkien sijoittamisen HTML:ään mahdollistaen tällä tavalla sisällön dynaamisen generoinnin. Razor kääntäjä erottaa staattisen HTML:n C# -koodista. Razor syntaksissa C# -koodit kääritään @ -merkkeihin. Razor -syntaksin ansiosta HTML-sivuilla on mahdollista kirjoittaa tarvittavat silmukat ja ehtolauseet.

Tag Helpers on ASP.NET Core MVC:n sisäänrakennettu ominaisuus, joka mahdollistaa palvelinpään metodien paluuarvojen käyttöä HTML-elementtien luomiseen ja renderöintiin Razor tiedostoissa. Useimmat sisäänrakennetut Tag Helperit ovat suunniteltu käytettäväksi standardien HTML -elementtien kanssa ja tarjoavat elementeille tarkoitetut palvelinpään attribuutit kuten esimerkiksi `<input>` -elementtien kanssa käytettävä `asp-for` -attribuutti. (Anderson 2019.)

ASP.NET Core laajentaa standardia HTML `<a></a>` -ankkuritagia uusien attribuuttien lisäämisellä. ASP.NET Core:ssa kyseinen ominaisuus on nimetty "Anchor Tag Helper":ksi. Konvention mukaisesti `<a></a>` -tagin attribuutteihin lisätään `asp-` -etuliite, minkä kautta `<a></a>` -elementin `href` -attribuutin arvo määritellään `asp-` -attribuuttien arvojen perusteella. (Addie & Kellner 2019.)

Yleiset HTML -rakenteet kuten esimerkiksi sivun staattiset osat (header, footer), skriptit ja tyylitiedostot ovat usein useamman sivun käytössä sovelluksen sisällä. Sovelluksen sisällä toistuvat elementit on mahdollista saada kaikkien näkymien käyttöön ASP.NET Core:n `_Layout.cshtml` -kaavaimen (template) avulla. Jaettu ulkoasu sijoitetaan

oletuksena tiedostoon `Views/Shared/_Layout.cshtml`. `_Layout.cshtml` -kaavain on korkean tason kaavain sovelluksen näkymille. Oletuksena jokaisen layout -näkymän kuuluu kutsua `RenderBody` -metodia. Kutsumalla `RenderBody` -metodia, renderöidään näkymän sisältö. Layout -näkymä voi myös kutsua `RenderSection` -metodia. Metodin kutsu mahdollistaa elementtien sijoittaminen haluttuun paikkaan ja voi määrittää elementit joko pakollisiksi tai vaihtoehtoisiksi. (Smith & Brock 2019.)

2.6 Entity Framework Core ja tietokantamigraatiot

Entity Framework Core (EF Core) on Microsoft:n kehittämä olio-relaatio-mallinnustyökalu (object-relational mapping (ORM)), joka hoitaa relaatiodatan ja oliomallin välisen konversion automaattisesti.

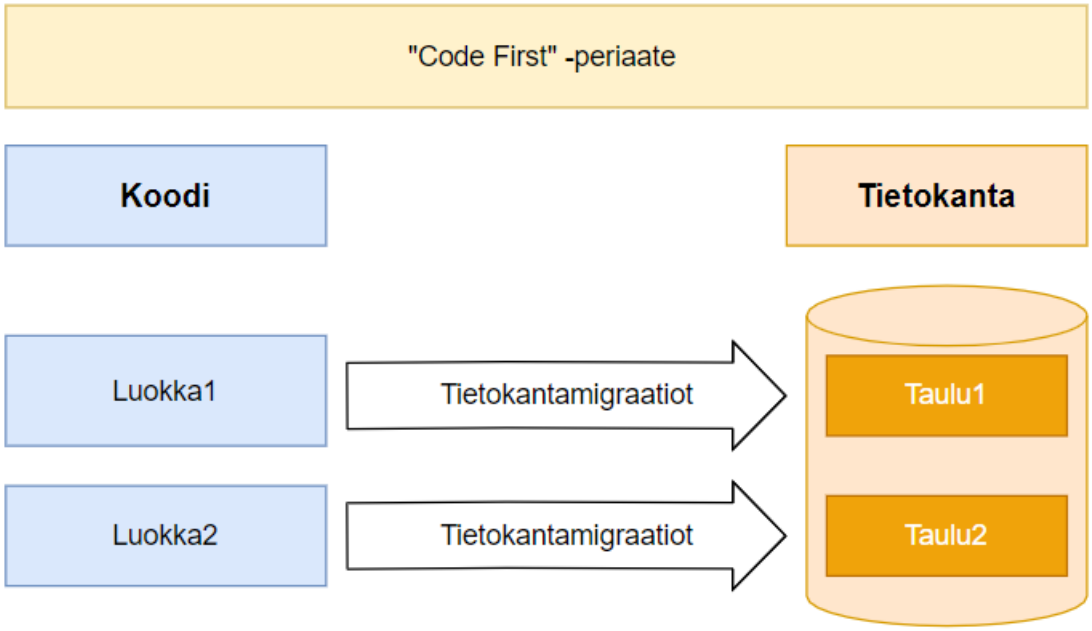
Sovelluksen puolella relaatioiden data saadaan käyttöön entiteettien muodossa EF Core:n avulla. Yhteys toimii myös toisinpäin – sovelluksen puolella entiteetteihin / luokkiin tehdyt muutokset on helppo tallentaa relaatiotietokantaan EF Core:n avulla.

Opinnäytetyön kirjoittamisen hetkellä EF Core oli käytetyin tapa käsitellä tietokantoja ASP.NET Core projekteissa (Freeman 2020, 242).

Tässä projektissa EF Corea käytettiin ”Code First” -tavalla, mikä tarkoittaa sitä, että entiteetit luotiin lähdekoodissa. Näin ollen lähdekoodiin tehdyt entiteetit kuvaavat sovelluksen tietorakennetta. Näiden entiteettien avulla luodaan relaatiotietokantaa.

EF Core antaa tietokantapääsyn `DbContext` -kontekstiluokan kautta. Tietokantayhteyden muodostamiseksi EF Core pitää olla konfiguroitu niin, että EF Core saa käyttöönsä seuraavat tiedot: tietokannan tyyppi, johon tulee kytkeytyä; mikä connection string kuvaa tietokantayhteyden ja mikä kontekstiluokka esittää datan tiedokannassa. Tietokantayhteyksiä konvention mukaisesti konfiguroidaan `Startup` -luokassa. EF Core konfiguroidaan `AddDbContext` -metodin avulla, joka rekisteröi `DbContext` -luokan ja konfiguroi sovelluksen ja tietokannan väliset suhteet. (Freeman 2020, 244.)

Muutokset, jotka tehdään entiteetteihin (luokkiin) sovelluksessa, viedään relaatiotietokantatasolle migraatioiden kautta eli migraatiot päivittävät tietokantaa uudempaan tai vanhempaan versioon. ”Code First” – tavassa migraatiot generoidaan automaattisesti koodin pohjalta, minkä jälkeen migraatioita voi tarkistaa ja tarvittaessa säätää käsin. Sen jälkeen migraatio ajetaan, jolloin tietokantarelaatiot päivittyvät vastaamaan sovelluksen entiteetti-mallia (Kuva 3).



Kuva 3. "Code First" -periaate

Näin ollen olio-relaatio-mallinnus -työkaluna EF Core huolehtii seuraavista tehtävistä projektissa:

- Automaattisesti avaa tietokantayhteydet määriteltyihin tietokantoihin;
- Luo tietokantaan sovelluksen puolella kuvatut entiteetit (luokat). Luokissa kuvatuista properteista automaattisesti muodostetaan taulujen sarakkeet. Tarvittaessa luokkien properteille voi määritellä säännöt tai attribuutit (englanniksi ”data annotation attributes”) kuten esimerkiksi tietotyypit, pakollisuusattribuutit `[Required]`, perusavain- attribuutti `[Key]` sekä `[DatabaseGenerated(DatabaseGeneratedOption.Identity)]` – attribuutti, joka määrittää kentän automaattisesti kasvavaksi id-kentäksi;
- Päivittää tietokantojen entiteetit vastaamaan sovelluksessa määriteltyjä rakenteita ja sääntöjä;
- Automaattisesti sulkee tietokantayhteydet.

(Freeman, 2020.)

2.7 Language-Integrated Query (LINQ)

LINQ (Language-Integrated Query) on termi, jolla kutsutaan eri teknologioita, jotka pohjautuvat kyselykielen integrointiin suoraan C# kieleen. Perinteisesti kehittäjä joutuu opettelemaan omaa kyselykieltä jokaista datalähdettä varten: SQL tietokannat, XML dokumentit yms. LINQ:n tapauksessa kysely kuuluu perus (ensimmäisen luokan) kielirakenteisiin kuten muun muassa luokat (classes), metodit (methods) ja tapahtumat

(events) (Microsoft 2016.). Näin ollen LINQ -kyselyitä voi käyttää datahakuihin kaikista yllä mainituista lähteistä.

LINQ-kyselyt voidaan kirjoittaa kahdella eri tavalla: lambdalausekkeen muodossa (engl. lambda expression) tai kyselylausekkeen muodossa (engl. query expression).

Lambdalausekkeet lisättiin C#-kieleen versiossa 3.0. Lambdalausekkeiden muodossa voidaan kirjoittaa niin sanottuja anonyymimetodeja (delegaatteja). Näitä voidaan välittää toiselle metodille ilman, että lambdalausekkeesta kirjoitettaisiin erikseen oman metodin. Lamdalausekkeilla voidaan määritellä paluuarvon tyyppi: kuten kaikki muutkin metodit C#:ssa delegaattikutsu on vahvasti tyypitetty. Lambdalausekkeiden avulla voi huomattavasti vähentää ohjelmakoodin rivien määrää.

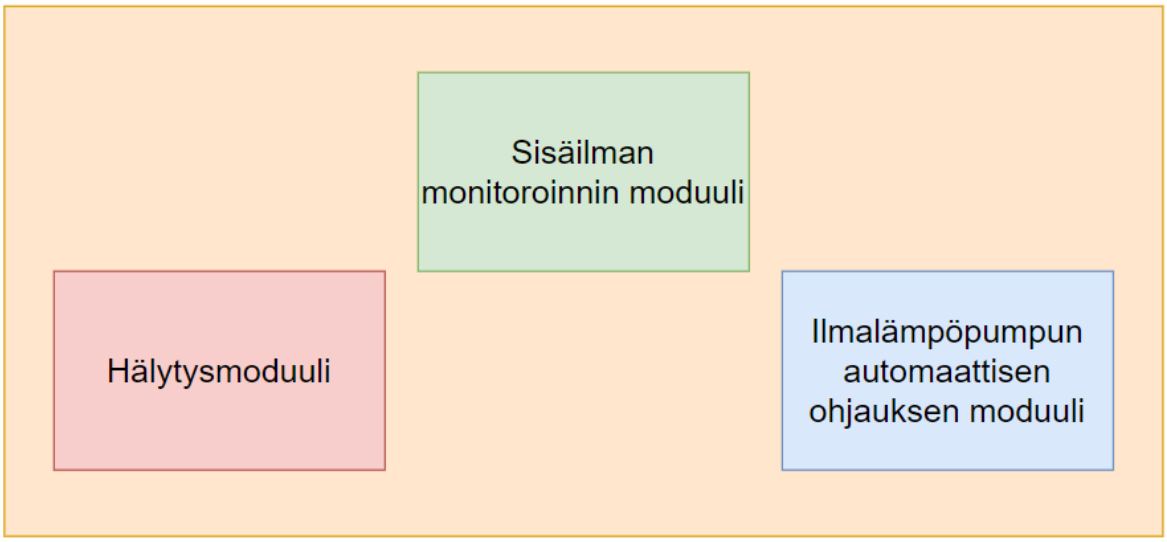
Kyselylausekkeet (query expressions) aina aloitetaan `from` -määreellä ja lopetetaan joko `select` – tai `group` -komennolla. LINQ-kyselyt kyselylausekkeen muodossa muistuttavat syntaksiltaan SQL-kyselyitä, mutta kuitenkin LINQ- ja SQL- kyselyiden välillä on selkeitä eroja. C#-kielessä LINQ-kyselylauseke rakennetaan seuraavassa järjestyksessä: `from -> where -> select`. Vastaava SQL -lause kirjoitetaan eri järjestyksessä: `SELECT -> FROM -> WHERE`. Syynä tähän muun muassa on IntelliSense - IDE:n ominaisuus tarjota kehittäjälle apua alasvetovalikoiden muodossa, joissa kehittäjä pääsee näkemään käsiteltävän olion kenttiä. C#-kielen LINQ-kyselylausekkeissa käytetty järjestys kuvaa paremmin loogisen järjestyksen, jossa toiminnot suoritetaan. Kysely aloitetaan valitsemalla tarvittava kokoelma `from` -määreellä, sitten suodatetaan pois tarpeettomat tiedot `where` -ehdolla ja lopuksi `select` -komennolla kuvataan haluttu lopputulos. (Michaelis 2018, 22054 – 22055.)

3 Sovelluksen kuvaus ja rakenne

Tässä luvussa käsitellään toteutetun sovelluksen rakennetta ja käydään läpi jokaisen kokonaisuuden teknistä toteutusta.

3.1 Sovelluksen kuvaus

Sovellus koostuu kolmesta loogisesta kokonaisuudesta: sisäilman monitoroinnin moduulista, hälytysmoduulista ja ilmalämpöpumpun automaattisen ohjauksen moduulista (Kuva 4).



Kuva 4. Sovelluksen moduulit

Sisäilman monitoroinnin moduuli koostuu Dashboard -näkymästä ja `/Targets/Details/{targetId}` -näkymästä. Dashboard -näkymässä käyttäjä pääsee tarkastelemaan yhteenvedon kaikista kohteista ja jokaisen kohteen uusimman lämpötilan ja kosteusasteen mittaustuloksia. Dashboard -näkymä on sovelluksen oletusnäkymä. Dashboard -näkymästä ja sen teknisestä toteutuksesta kerrotaan tarkemmin luvussa ”Sisäilman monitoroinnin moduuli”.

`/Targets/Details/{targetId}` -näkymässä, jossa `{targetId}` on valitun kohteen tunniste, käyttäjä pääsee tarkastelemaan tarkemmin kohteen lämpötila ja kosteuslukemia sekä voi seurata lämpötilan ja kosteusasteen muutostrendiä viivakaavioista. Lämpötila- ja kosteustason viivakaaviot graafisesti esittävät mittaustulosten muutostrendiä. Lämpötila- ja kosteustasokaaviot päätettiin toteuttaa käyttöliittymässä erillisinä kaavioina sen takia, että mitattavilla arvoilla on erilaiset mittausasteikot ja niiden esittäminen samassa

mittakaavassa on vaikeata. Usein kun sisätilojen lämpötilat muuttuvat 18 – 22 ºC:n -välillä, kosteustason vaihtelut saattavat sijoittua esimerkiksi 40 – 55 %:n välille, jolloin muutostrendien havainnollistaminen samassa kaaviossa saattaa olla haastavaa. Yksi asennetuista antureista sijaitsee ulkona, jolloin 20 ºC lämpötilassa saattaa samanaikaisesti olla 85% prosentin kosteus. Toteutuksen aikana kokeilun jälkeen todettiin, että tämäntyyppisten arvojen sijoittaminen samalle kaaviolle ei ole järkevä vaihtoehto. `/Targets/Details/{targetId}` -näkymästä ja sen teknisestä toteutuksesta sekä viivakaavioiden toteutuksesta kerrotaan tarkemmin luvussa 3.5 Sisäilman monitoroinnin moduuli.

Hälytysmoduuli koostuu hälytysten dashboard -päänäkymästä, jossa käyttäjä pystyy luomaan, muokkaamaan ja poistamaan hälytykset, sekä komponentista, joka hoitaa varsinaisen notifikaatioiden lähettämisen ja käyttää siihen ulkoista Pushover -järjestelmää. Hälytysten dashboard -näkymästä, hälytysten hallinnasta (uuden hälytyksen luomisesta, olemassa olevan hälytyksen muokkaamisesta ja poistosta) sekä notifikaatioiden lähettämisen prosessista kerrotaan tarkemmin opinnäytetyön luvussa 3.6 Hälytysmoduuli.

Ilmalämpöpumpun automaattisen ohjauksen moduuli hoitaa huoneiston ilmalämpöpumpun automaattisen käynnistämisen ja pysäyttämisen sisäilman tilasta riippuen Sensibo Sky rajapinnan kautta. Ilmalämpöpumpun automaattisen ohjauksen toteutus ja sovelluksen kommunikointi ilmalämpöpumpun kanssa rajapinnan kautta on kuvattu opinnäytetyön kohdassa 3.7 Ilmalämpöpumpun automaattisen ohjauksen moduuli.

Keskeisimmissä rooleissa sovelluksen toteutuksessa ovat Entity Framework Core ja seuraavat ASP.NET Coren moduulit: MVC Framework, Razor Pages ja Platform-tason toiminnot kuten esimerkiksi: HTTP Server, Middleware, URL Routing, Caching, Model Binding, Razor.

Sovellus käyttää Pushover- ja SensiboSky -ulkoisia järjestelmiä. Ulkoiset järjestelmät ovat integroitu sovellukseen HostedService:ien avulla ja löytyvät projektin HostedServices -hakemistosta. Sovelluksessa käytettyjen HostedService -palveluiden tarkoitus on se, että palvelut käynnistetään sovelluksen käynnistämisen yhteydessä ja palvelut pyörivät taustalla koko ajan, kun sovellus on päällä. HostedService-palvelut implementoivat IHostedService -rajapinnan.

3.2 Tietokannat

Sovellus käyttää kahta tietokantaa ja vastaavasti sovelluksessa on kaksi eri Entity Framework Database Context -luokkaa: `HomeClimateAppDbContext` ja

`HomeClimateAppMariaDb`, jotka molemmat rekisteröidään `AddDbContext` -metodilla sovelluksen Startup -luokassa. Muutokset tietokantojen entiteetteihin tehdään tietokantamigraatioiden kautta.

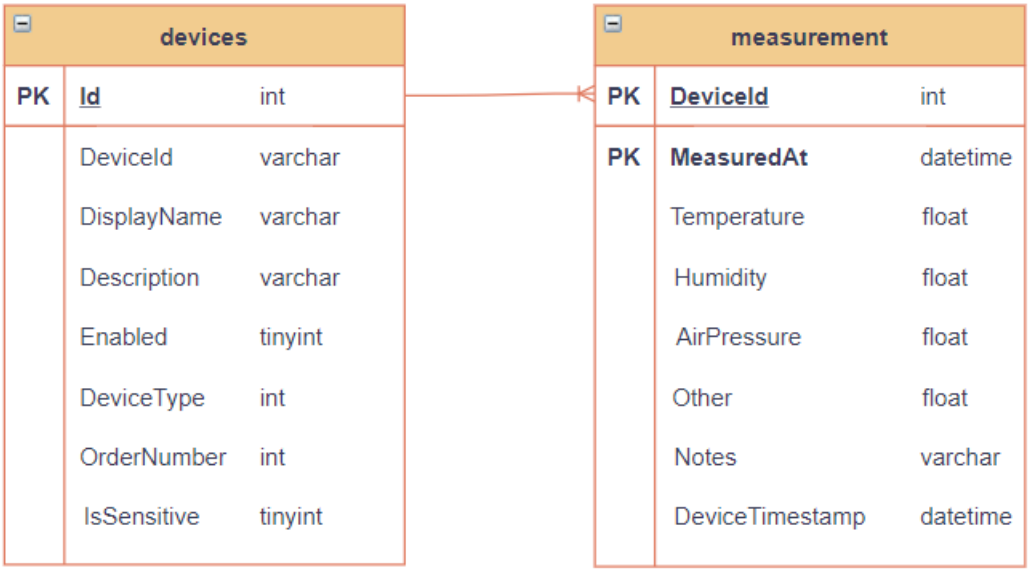
Ohjaimet saavat `DbContext` -luokkien instanssit käyttöönsä `dependency injection` -ominaisuuden kautta.

```
    public class AlarmsController : Controller
    {
        private readonly HomeClimateAppMariaDb _contextMariaDB;
        private readonly HomeClimateAppDbContext _context;

        public AlarmsController(HomeClimateAppDbContext context, HomeClimateAppMari-
aDb contextMariaDB)
        {
            _context = context;
            _contextMariaDB = contextMariaDB;
        }


             ...
```

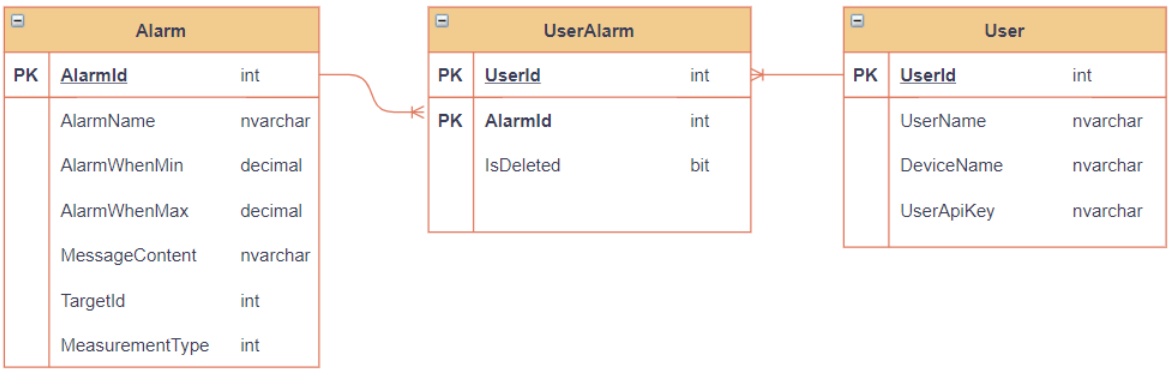
Sovellus hakee kohteiden tiedot ja mittaustulokset MariaDB -tietokannasta, joka sijaitsee Raspberry PI -palvelimella. Sovelluksen sisäilman monitoroinnin moduulin toiminnot perustuvat kahteen MariaDB -tietokannan tauluun (”devices” ja ”measurement”), jotka sisältävät olennaista dataa sovelluksen toimivuuden kannalta. ”devices” -tauluun on tallennettu tiedot kaikista kohteista, joihin on asennettu RuuviTag -anturi. ”measurement” -tauluun minuutin välein tallennetaan RuuviTag -antureiden mittaustulokset kohdekohtaisesti. Sovelluksessa käytetyn MariaDB -tietokannan taulut ja niiden rakenne on esitetty kuvassa Kuva 5.

Jotta Entity Framework Core – olio-relaatio-mallinnustyökalun käyttö Code First-tapaan olisi mahdollista läpi sovelluksen ja myös MariaDB -tietokantakontekstissa, projektiin asennettiin `Pomelo.EntityFrameworkCore.MySql` -kirjasto. Kyseinen kirjasto mahdollistaa MariaDB -tietokannan käyttöä EF Core:n kanssa.

Sisäilman monitoroinnin moduulin lisäksi sovellukseen rakennettiin hälytysmoduuli. Hälytysmoduulia varten suunniteltiin ja toteutettiin oma SQL -tietokanta. Kehityksen aikana tietokantapalvelimena käytettiin SQL Server Express -palvelimen Microsoft SQL Server Express LocalDB -ominaisuutta. Hälytysmoduulin tietokannan rakenne on esitetty kuvassa Kuva 6.



Kuva 5. Sisäilman monitoroinnin moduulin tietokannan rakenne (MariaDB)

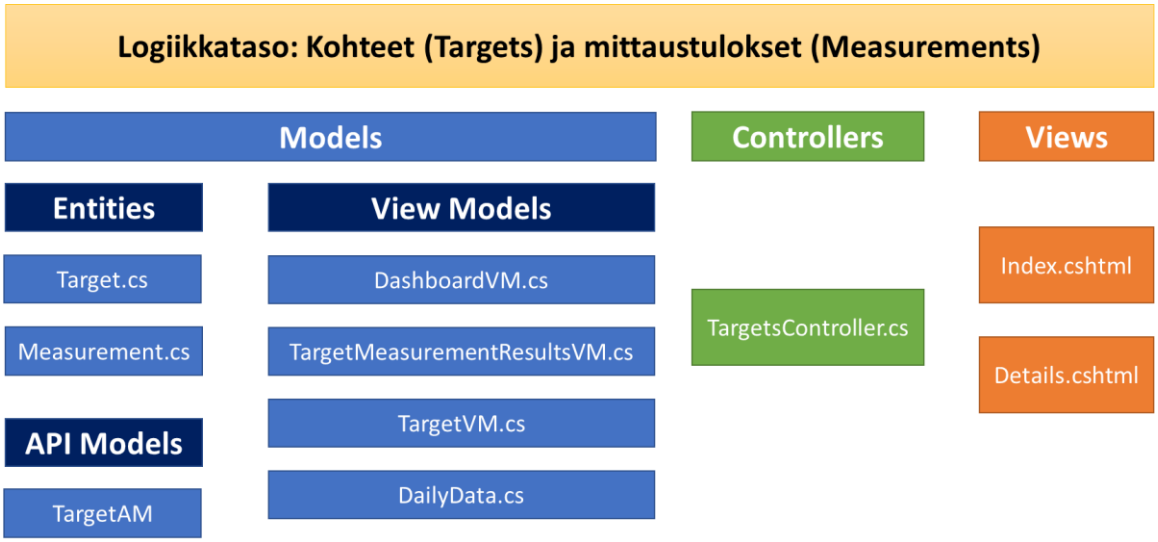


Kuva 6. Hälytysmoduulin tietokannan rakenne.

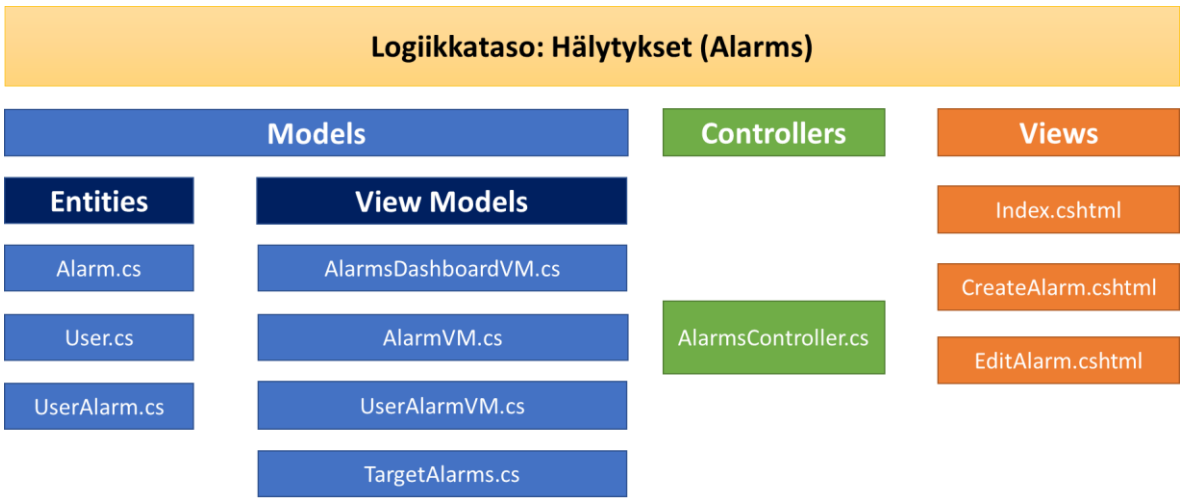
3.3 MVC -ohjelmistoarkkitehtuurin periaatteiden implementointi sovelluksessa

Sovellus on toteutettu MVC -ohjelmistoarkkitehtuurimallia noudattaen. Jokainen looginen kokonaisuus sovelluksen sisällä seuraa MVC -ohjelmistoarkkitehtuurimallia. MVC -ohjelmistoarkkitehtuurimallin käyttöä hälytysmoduulin ja sisäilman monitoroinnin moduulin kokonaisuuksissa havainnollistavat kuvat: Kuva 7 ja Kuva 8.

MVC-arkkitehtuurimallin mukaisesti toteutetuissa moduuleissa, sovelluksen logiikka on jaettu kolmeen komponenttikokonaisuuteen: mallit, ohjain ja näkymät. Hälytysmoduulissa mallit ovat jaettu vielä kahteen loogiseen kokonaisuuteen: entiteettimalleihin ja näkymämalleihin. Sisäilman monitoroinnin moduulissa entiteetti- ja näkymämallien lisäksi käytettiin myös API-kutsun palveluun tarkoitettua API-mallia.



Kuva 7. MVC -ohjelmistoarkkitehtuurin käyttö sisäilman monitoroinnin moduulissa



Kuva 8. MVC -ohjelmistoarkkitehtuurin käyttö hälytysmoduulissa

Kaavioissa nähdään, että sovelluksen mallitasoa laajennettiin lisäämällä näkymämalli -abstraktiotason. Kyseisen abstraktiotason tehtävä sovelluksessa on eriyttää tietokantojen relaatioita peilaavat entiteettimallit (luokat) niistä malleista (luokista), joita käyttävät sovelluksen näkymät. Toisin sanoen näkymämallitaso eriyttää sovelluksen tietokannan ja käyttöliittymän toisistaan.

Ylimääräisen näkymämallin abstraktiotason käyttö sovelluksessa on perusteltu valinta, koska ViewModel -näkymämalliluokat ilmaisevat juuri nämä tiedot ja juuri niissä muodoissa, kuin käyttöliittymä tietoja tarvitsee (tietokanta ei aina toimi näin).

Näkymä -tasolla sovellus käyttää jaettua `_Layout.cshtml` -näkymää renderöimään jokaisella sivulla toistuvat yhteiset komponentit kuten esimerkiksi `<header>` -elementin, `navbar` -pseudoelementin ja `<footer>` -elementin,. `_Layout.cshtml` -näkymä sisältää `<div class="container">` -elementin, johon renderöidään sivun pääsisältö:

```
<div class="container">
    <main role="main" class="pb-3">
        @RenderBody()
    </main>
</div>
```

Käyttämällä ASP.NET Core `RenderSection` -metodia `_Layout.cshtml` -näkymässä suoritetaan sivuspesifiset skriptit:

```
@RenderSection("Scripts", required: false)
```

`Details.cshtml` -näkymässä lämpötilan ja kosteustason kaaviot piirretään suorittamalla script-tagin sisältöä.

3.4 ASP.NET Core -komponenttien ja ASP.NET Core MVC käyttö sovelluksessa

Sovelluksen käyttämiä palveluita (services) ja väliohjelmistokomponentteja (middleware components) alustetaan Startup -luokassa. Sovelluksen `Startup` -luokka sisältää kaksi metodia: `ConfigureServices` ja `Configure`.

```
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddControllersWithViews();
            services.AddDbContext<HomeClimateAppDbContext>(options => op-
tions.UseSqlServer(Configuration.GetConnectionString("HomeClimateAppDbContext")));
            services.AddDbContextPool<HomeClimateAppMariaDb>(options => options.Use-
MySql(Configuration.GetConnectionString("MariaDbAppDbContext"), mySqlOptions =>
mySqlOptions
                    .ServerVersion(new Version(10, 3, 23), ServerType.MariaDb)
            ));
            services.AddHostedService<HostedServices.NotificationSender>();
            _notificationsAppToken = Configuration["Notifications:APP_TOKEN"];
            _notificationsUserKey = Configuration["Notifications:USER_KEY"];

            services.AddHostedService<HostedServices.SensiboConfigurator>();
            _sensiboAPIkey = Configuration["Sensibo:API_KEY"];
        }
```

Sovellus käyttää konventioon perustuvaa reititystä, jossa käyttäjän tekemä HTTP-pyyntö ohjataan valitun ohjaimen valitulle toiminnolle.

Sovelluksessa käytetty reitityskaava on konvention mukainen ja on myös määritelty `Startup.cs` tiedostossa. Määritelty reititysmalli sallii reitissä myös vapaaehtoisen id-attribuutin käyttöä:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Targets}/{action=Index}/{id?}");
});
```

Määritelmän mukaisesti `/Targets/Details/2` -HTTP -pyyntö ohjataan controllerille Targets (tiedoston nimi: `TargetsController.cs`) ja toiminnolle `Details`. `Details` -toiminto on varustettu `[HttpGet]` -attribuutilla, joka määrittää, että kyseiseen endpointiin tulevat kutsut palvellaan **vain** GET -pyyntöinä. `/Targets/Details/2` -HTTP -pyyntö sisältää myös vapaaehtoisen id-attribuutin, jota tässä tapauksessa tarvitaan siihen, että sovellus osaisi palauttaa valitun kohteen tiedot.

```
[HttpGet]
public async Task<IActionResult> Details(int? id)
{

    ...

}
```

Sovellus laajasti käyttää sellaisia ASP.NET Core MVC -komponentin sisäänrakennettuja toiminnallisuuksia kuten mallin sitominen (Model binding) ja mallivalidointi (Model validation).

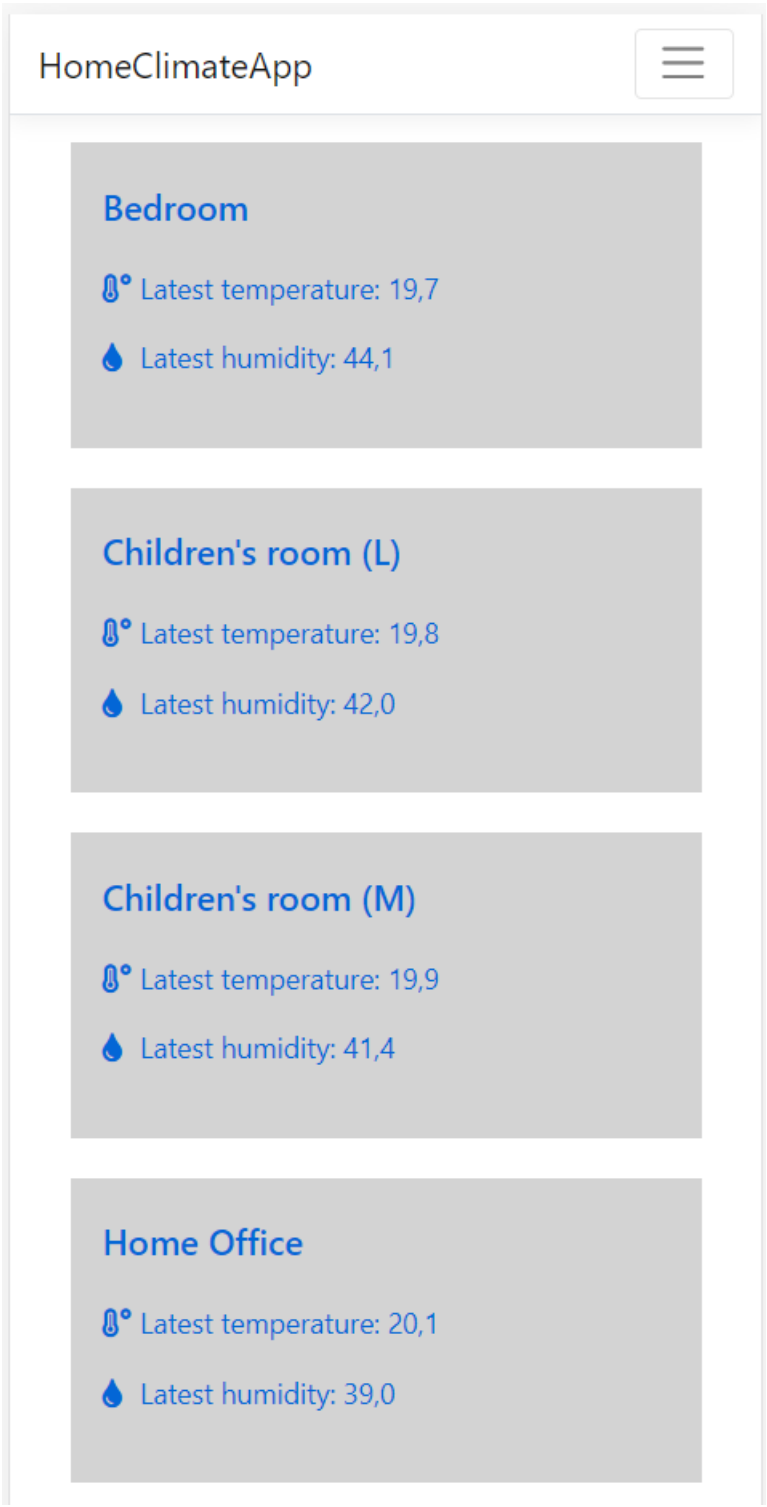
`/Targets/Details/2` -HTTP -pyynnön palauttama vastaus pohjautuu ASP.NET Core MVC mallin sitomiseen. Reittiin sisältyy id-muuttuja, jonka arvo on tässä tapauksessa 2. Kun mallin sitomisjärjestelmä lähtee hakemaan dataa ja huomaa reitissä olevan id-muuttujan, sovellus tietää kutsua Details -toimintometodin, jolle parametrina välitetään arvo 2.

ASP.NET Core MVC:n mallivalidointi on toteutettu sovelluksessa validointiattribuuttien avulla. Uuden hälytyksen luomisessa kaikkien mallin ominaisuuksien kohdalla käytetään `[Required]` -validointiattribuuttia, joka määrittää arvon ominaisuuden pakolliseksi. `[Required]` – validointiattribuutit useamman kentän kohdalla on varustettu `ErrorMessage` -property:lla, joka mahdollistaa kustomoidun virheviestin renderöinnin.

```
public class AlarmVM
```

```
{

    [Required]
    public int AlarmId { get; set; }

    [Required(ErrorMessage = "Fill in the name for the alarm")]
    public string AlarmName { get; set; }

    [Required]
    [Range(1, 100, ErrorMessage = "Value should be greater than 0")]
    public decimal AlarmWhenMin { get; set; }


 ...

}
```

3.5 Sisäilman monitoroinnin moduuli



Kuva 9. Sisäilman monitoroinnin moduulin Dashboard -näkymä

Dashboard -näkymän (Kuva 9) tarkoitus on esittää viimeisimpiä mittaustuloksia sisältävän listan kaikista kohteista, joihin on asennettu RuuviTag -anturi. Dashboard on

oletusnäkymä, johon käyttäjä ohjataan sovelluksen avaamisen yhteydessä. `Startup.cs` -tiedostossa on määritelty, millä mallilla (pattern) reititys sovelluksen sisällä tulee tapahtumaan.

Oletusohjaus Dashboard-näkymälle on myös määritelty Startup.cs -tiedostossa:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Targets}/{action=Index}/{id?}");
});
```

Tällä kommennolla avaamisen yhteydessä sovellusta käsketään siirtymään `Targets`-controllerin `Index`-metodiin. Targets-controllerin Index-metodi on varustettu `[HttpGet]` -attribuutilla, mikä tarkoittaa sitä, että kyseistä metodia voi kutsua vain GET HTTP-pyyntönä.

Index-metodi hakee tietokannasta `DbContext-luokassa` määritellyt entiteetit ja muokkaa haetut tiedot näkymälle sopivaan muotoon. Index-metodi hakee tiedot kahdesta tietokannan taulusta: Target ja Measurement.

Huoneiden tiedot haetaan seuraavalla LINQ kyselyllä:

```
var targets = await (from t in _context.Target
                     where t.DeviceType == 1 || t.DeviceType == 2
                     select new Target {
                         TargetId = t.TargetId,
                         DeviceId = t.DeviceId,
                         DeviceName = t.DeviceName,
                         TargetName = t.TargetName,
                     }).ToArrayAsync();
```

Datan esittämiseen Dashboard-näkymässä käytetään ViewModel-malliluokkia, jolle välitetään kannasta haetut tiedot. Tätä varten Index -metodissa luodaan uusi instanssi DashboardVM -luokasta:

```
DashboardVM dashboardVM = new DashboardVM();
```

Jokaisesta kannasta haetusta huoneesta (Target -entiteetti) luodaan oma TargetVM-ViewModel luokan instanssi.

```
TargetVM targetVM = new TargetVM();
```

DashboardVM -luokan esiintymälle on tarkoitus välittää kaikkien huoneiden tiedot ja jokaisen huoneen viimeisimmät mittaustulokset. Tätä varten luodaan uusi tyhjä lista, joka on tarkoitettu TargetVM instansseille, ja välitetään lista dashboardVM-instanssille.

```
dashboardVM.Targets = new List<TargetVM>();
```

TargetVM -ViewModel instanssi tulee saamaan kaikki tiedot, jotka halutaan näyttää käyttäjälle. Tässä tapauksessa ViewModel-luokan käyttö on perusteltua, koska näkymässä tarvitaan vain rajoitetusti kannassa olevaa dataa, joka on tarkoitus kasata useammasta taulusta. Tällä tavalla ViewModel-luokan avulla saadaan kasattua kaikki näkymään tarvittavat tiedot yhteen instanssiin.

Jokaisesta huoneesta (Target) luodaan erillinen TargetVM instanssi. Nämä instanssit täytetään `foreach` -loopin avulla, missä haetaan viimeisimmät mittaustulokset (Measurement).

Luodulle dashboardVM -instanssille välitetään jokainen täytetty TargetVM -instanssi `foreach` -loopin avulla:

```
dashboardVM.Targets.Add(targetVM);
```

`Index` -metodi palauttaa `Index.cshtml` -näkymän, jolle parametrina välitetään kaikki dashboard-näkymässä tarvittavat tiedot:

```
return View(dashboardVM);
```

`Index.cshtml` on vahvasti tyypitetty näkymä. `@model` -direktiivi määrittää mallin (Model), joka on välitetty näkymälle. Razor tarjoaa Model -nimisen propertyn, jonka kautta päästään käsiksi näkymälle välitettyyn malliin. Sovelluksen `Index.cshtml` -näkymälle on välitetty DashboardVM -mallinäkymä.

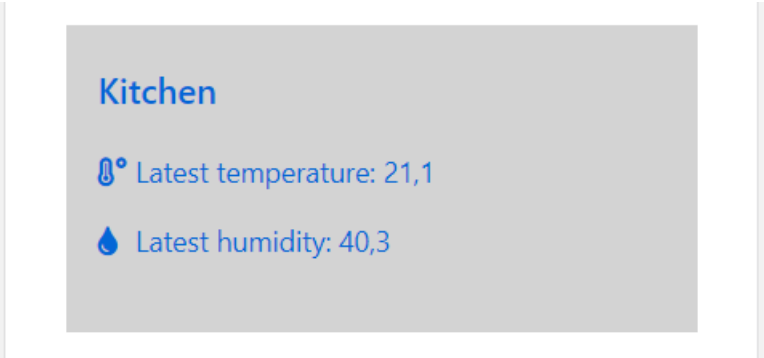
```
@model HomeClimateApp.Models.DashboardVM;
```

Näin ollen Index.cshtml -näkymä seuraa DashboardVM -näkymämallin rakennetta ja sai parametriarvona DashboardVM -luokan dashboardVM -instanssin, joka sisältää mittauspaikkojen tiedot ja jokaisen mittauspaikan mittaustulostiedot. Koska Razor-syntaksin käyttö mahdollistaa C# -kielen käyttöä html:n sisällä, Razor-syntaksin avulla käydään läpi mittauspaikkoja sisältävän listan ja näytetään tiedot käyttäjälle.

Käyttämällä Razorin tarjoamaa Model -nimistä propertya päästään käsiksi DashboardVM -mallinäkymän Targets -listaan ja voidaan näyttää käyttäjälle kaikkien kohteiden tiedot käymällä läpi Targets -listaa foreach-silmukan avulla.

```
@foreach (var target in Model.Targets)
{...}
```

Kuva 10 esittää yhden kohteen näkymän käyttöliittymässä ja seuraavana on osa html -koodia, joka renderöi kyseisen näkymän (yksinkertaisuuden vuoksi koodinäkymästä on poistettu käyttöliittymässä näkyvien ikonien koodit).

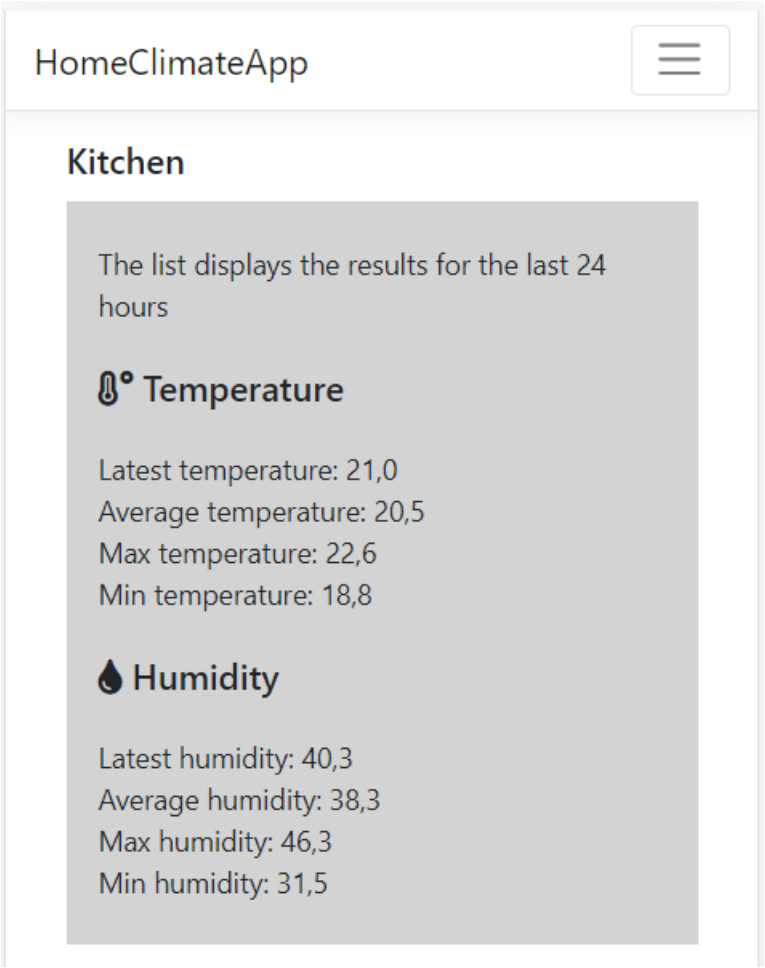


Kuva 10. Yhden kohteen näkymä Dashboardilla käyttöliittymässä

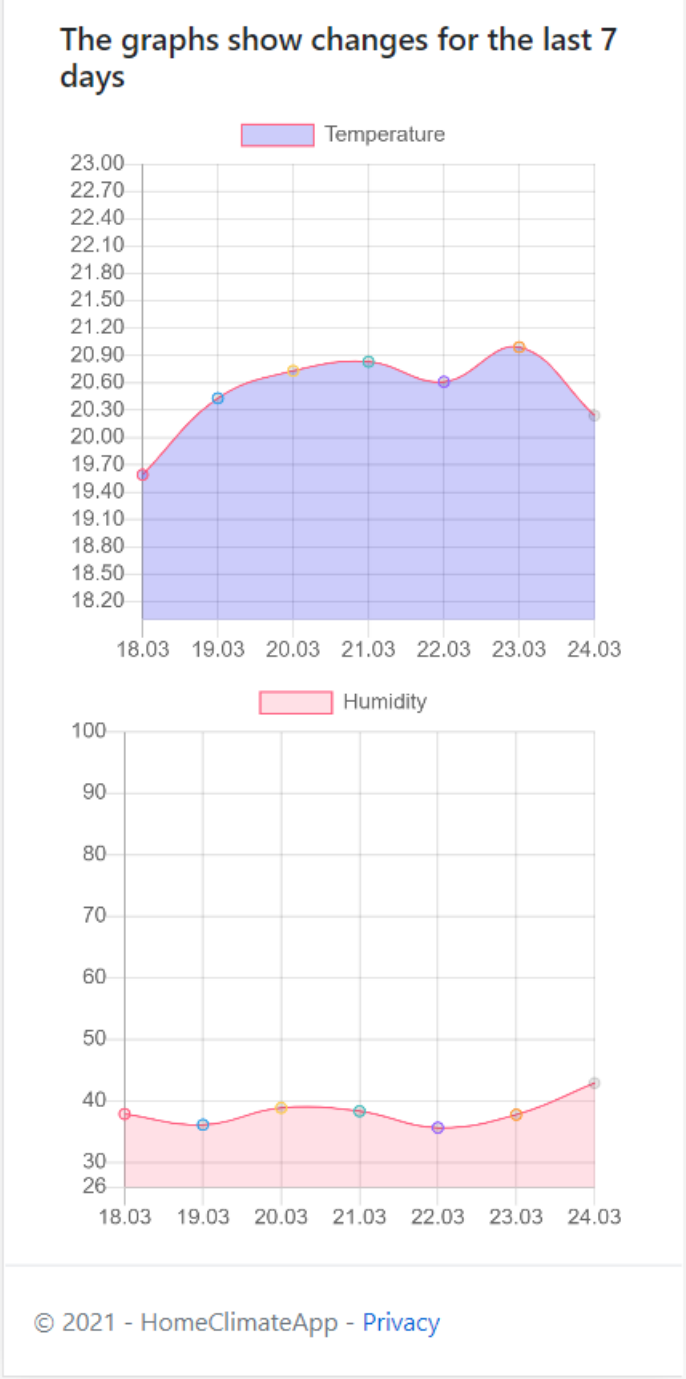
```
@foreach (var target in Model.Targets)
{
  <a asp-controller="Targets" asp-action="Details" asp-route-id="@target.TargetId">
    <div class="col-sm-12 col-md-12 col-lg-6 target-container">
       <h5 class="dashboard-target-container-heading"> @target.TargetName</h5>
       <p>Latest temperature: @target.LatestTemperature?.ToString("f1")</p>
       <p>Latest humidity: @target.LatestHumidity?.ToString("f1")</p>
    </div>
  </a>
}
```

Yllä olevassa koodipätkässä `<div>` -elementti, joka sisältää tiedot kohteesta ja mittaustuloksista, on kääritty `<a>` -tagiin. Klikkaamalla kohdetta käyttäjä pääsee tarkastelemaan valitun kohteen tiedot. `<a>` -tagin käyttö tässä tapauksessa on käytännön esimerkki ASP.NET Core:n ”Anchor Tag Helper” -ominaisuudesta. `<a>` -tagi sisältää `asp-controller`, `asp-action` ja `asp-route-id` -attribuutit, joiden arvoista muodostuu `<a></a>` -elementin `href` -arvo. Kun käyttäjä klikkaa kuva 2:n kohdetta käyttöliittymässä, käyttäjä ohjataan osoitteeseen: `... /Targets/Details/2`, jossa 2 on valitun kohteen id.

Klikkaamalla kohdetta Dashboard-näkymässä käyttäjä pääsee osoitteeseen `/Targets/Details/{id}` tarkastelemaan valitun kohteen lämpötilan ja kosteuden mittaustuloksia tarkemmin. Details -näkymä on kohteen mittaustulosten yhteenvetonäkymä. Näkymä koostuu kahdesta osuudesta: kohteen viimeisen vuorokauden mittaustulosten yhteenvetonäkymästä (Kuva 11) ja kaavionäkymästä, joka sisältää erillisen viivakaavion viimeisen seitsemän vuorokauden lämpötila- ja kosteustason keskiarvoille (Kuva 12).



Kuva 11. Kohteen mittaustulosten yhteenvetonäkymä (viimeinen vuorokausi)



Kuva 12. Seitsemän vuorokauden muutokset viivakaavioina

Valitun kohteen mittaustulosten yhteenveto -näkymän endpoint on `Targets/Details/2`, jossa 2 on tietokannasta tuleva kohteen yksilöivä tunniste (id). Edellisessä luvussa käsiteltiin, miten ASP.NET Core:n ”Anchor Tag Helper” -ominaisuuden avulla muodostetaan kyseinen polku. Endpointin muotoilu noudattaa `Startup.cs` -tiedostossa määriteltyä mallia, jossa polku muodostuu controllerin nimestä (`Targets`), actionin nimestä

(`Details`) ja `id`:stä, joka ei ole pakollinen. Kyseinen näkymä tarvitsee kohteen id:tä, sillä id:n perusteella haetaan valitun kohteen tiedot.

`Details` -action on tyypiltään `GET` -metodi ja se on varustettu `[HttpGet]` -attribuutilla. Metodin suorituksen alussa tarkistetaan, että polkuun on syötetty id-arvo ja että kyseinen kohde löytyy tietokannasta. Mikäli polusta puuttuu id tai syötetty id ei löydy tietokannasta, palautetaan `NotFoundResult` -vastauksen. `NotFoundResult` -vastauksen palauttaminen metodista palauttaa selaimelle 404 HTTP statuksen.

Measurement -taulusta haetaan kaikki valitun kohteen mittaustulokset erikseen lämpötila- ja kosteusmittauksista erikseen määritellylle ajalle. Tähän tarkoitukseen toteutettiin omat privaatit apumetodit, jotka palauttavat `Measurement` -luokan instansseja sisältävän taulukon kohteen lämpötilan ja kosteustason mittaustuloksista. Kohteen id ja haluttu aika (tunneissa) annetaan metodeille parametreina.

```
private Measurement[] GetTemperatureResultsForSelectedPeriod(int numberOfHours, int?
targetId)
  {
    var datetimeLimit = DateTime.UtcNow.AddHours(-numberOfHours);
    var temperatureResults = (from m in _context.Measurement
                          join t in _context.Target on m.TargetId equals t.TargetId
                          where m.TargetId == targetId
                                      && m.MeasurementTime > datetimeLimit
                          orderby m.MeasurementTime descending
                          select new Measurement
                          {
                            MeasurementTime = m.MeasurementTime,
                            TemperatureResult = m.TemperatureResult,
                            Target = new Target
                            {
                              TargetId = t.TargetId,
                              TargetName = t.TargetName
                            }
                           }).ToArray();
    return temperatureResults;
  }
```

`GetTemperatureResultsForSelectedPeriod` -metodissa `temperatureResults` -taulukon datan hakeminen ja varsinainen taulukon muodostaminen on toteutettu LINQ-kyselylausekkeen muodossa.

Vastaavanlainen metodi on toteutettu kosteudenmittaustulosten hakua varten. Metodit kutsutaan Details -actionin sisällä:

```
var temperatureResults = GetTemperatureResultsForSelectedPeriod(168, id);
```

Paluuarvoina saaduista taulukoista haetaan mittaustulosten keskiarvon, maksimiarvon ja minimiarvon erikseen lämpötilalle ja kosteudelle. Keskiarvon, maksimiarvon ja minimiarvon laskentaan käytetään C#-kielen metodeita: `Average(), Max(), Min()`. Nämä arvot näytetään käyttäjälle käyttöliittymässä mittaustulosten yhteenvetonäkymässä.

Kohteen mittaustulostenyhteenvedon kaavionäkymä (Kuva 12) sisältää kaksi erillistä viivakaaviota: toinen viivakaavio näyttää lämpötilamuutosten trendiä ja toinen – kosteustason muutosten trendiä viimeisen seitsemän vuorokauden aikana. Kaavionäkymät on toteutettu ulkoisen `Chart.js` -komponentin avulla. `Details.cshtml` -näkymä renderöi kaavionäkymät sivulle `<script></script>` -tagin kautta.

Valitun kohteen tunniste (id) haetaan näkymälle välitetystä mallista:

```
const targetId = @Model.Target.TargetId;
```

Käyttämällä valitun kohteen tunnistetta haetaan kaavioissa näytettävät tiedot valitulle kohteelle. Datan hakuun käytetään Javascriptin `fetch()` -metodia. Pakollisena parametrina `fetch()` -metodille syötetään haettavan resurssin polku:

```
fetch('/Targets/TargetData/' + targetId)
```

`TargetsController.cs` -ohjaimessa on määritelty, mitä data palautetaan, kun kyseiseen osoitteeseen tulee pyyntö. `'/Targets/TargetData/{targetId}'` -endpoint on määritelty olevan `HttpGet` -metodi. `Ok` -metodin paluuarvona endpoint palauttaa `OkObjectResult` -luokan instanssin, joka on tässä tapauksessa `TargetAM` -malli

```
var model = new TargetAM()
{
    Last7Days = last7Days,
    DailyAveragesFor7Days = dailyAveragesFor7Days
};

return Ok(model);
```

REST API -kutsuja varten on luotu oma `TargetAM` -niminen malli, jossa AM on lyhenne ”API Model” -sanoista. `TargetAM` -malli sisältää vain API -kutsuissa tarpeellisia tietoja siinä muodossa, jossa niitä on kätevä käyttää pyyntöä lähettävässä komponentissa.

`TargetAM` -instanssin sisältönä palautetaan `last7Days` -taulukon (array), joka sisältää `DateTime` -tyyppisiä viimeisen seitsemän vuorokauden instansseja. Kaavionäkymässä tarvitaan myös lämpötilan ja kosteustason mittaustuloksia viimeisten seitsemän vuorokauden ajalta. Tätä varten tietokannasta haetaan kaikki viimeisen seitsemän

vuorokauden (168 tunnin) mittaustulokset kutsumalla metodin, joka palauttaa `Measurement` -tyyppisiä instansseja sisältävän taulukon:

```
var measurementResults = GetMeasurementResultsForSelectedPeriod(168, id);
```

Jokaiselle vuorokaudelle lasketaan lämpötilamittaustulosten ja kosteustason mittaustulosten keskiarvon ja laitetaan keskiarvot `dailyAveragesFor7Days` -taulukkoon, joka lähtee palvelimen vastauksessa `fetch('/Targets/TargetData/' + targetId)` -kutsuun.

Sen jälkeen, kun vastaus palvelimelta on saatu, json() -metodi lukee JSON notaation mukaisen merkkijonon ja palauttaa siitä tehdyt oliot:

```
    fetch('/Targets/TargetData/' + targetId)
        .then((resp) => resp.json())
        .then(function (data) {
            let measurementResults = data.dailyAveragesFor7Days;

     ... }
```

`Details.cshtml` -näkymä sisältää kaksi `<canvas>` HTML elementtiä, joista toinen on tarkoitettu containeriksi lämpötilakaaviolle ja toinen – kosteustason kaaviolle (esimerkkinä käsitellään lämpötilakaaviota):

```
        <canvas id="temperatureChart" width="400" height="400"></canvas>
```

Palvelimelta saatua `json` -dataa muunnetaan kaavioissa esitettävään muotoon.

```
var dates = [];
var temperatureResults = [];

data.dailyAveragesFor7Days.map(function (dailyResult) {
    dates.push(dailyResult.date);
    temperatureResults.push(parseFloat(dailyResult.temperatureDailyAver-
age.toFixed(2)));
    });
```

Kaavioiden piirtäminen containereihin tapahtuu niin, että `<canvas>` -elementeille luodaan `Chart` -tyyppiset oliot käyttäen palvelimelta saatua ja käsiteltyä dataa.

```
var ctxTemperature = document.getElementById('temperatureChart');
var temperatureChart = new Chart(ctxTemperature, {
                type: 'line',
                data: {
                    labels: dates,
                    datasets: [{
```

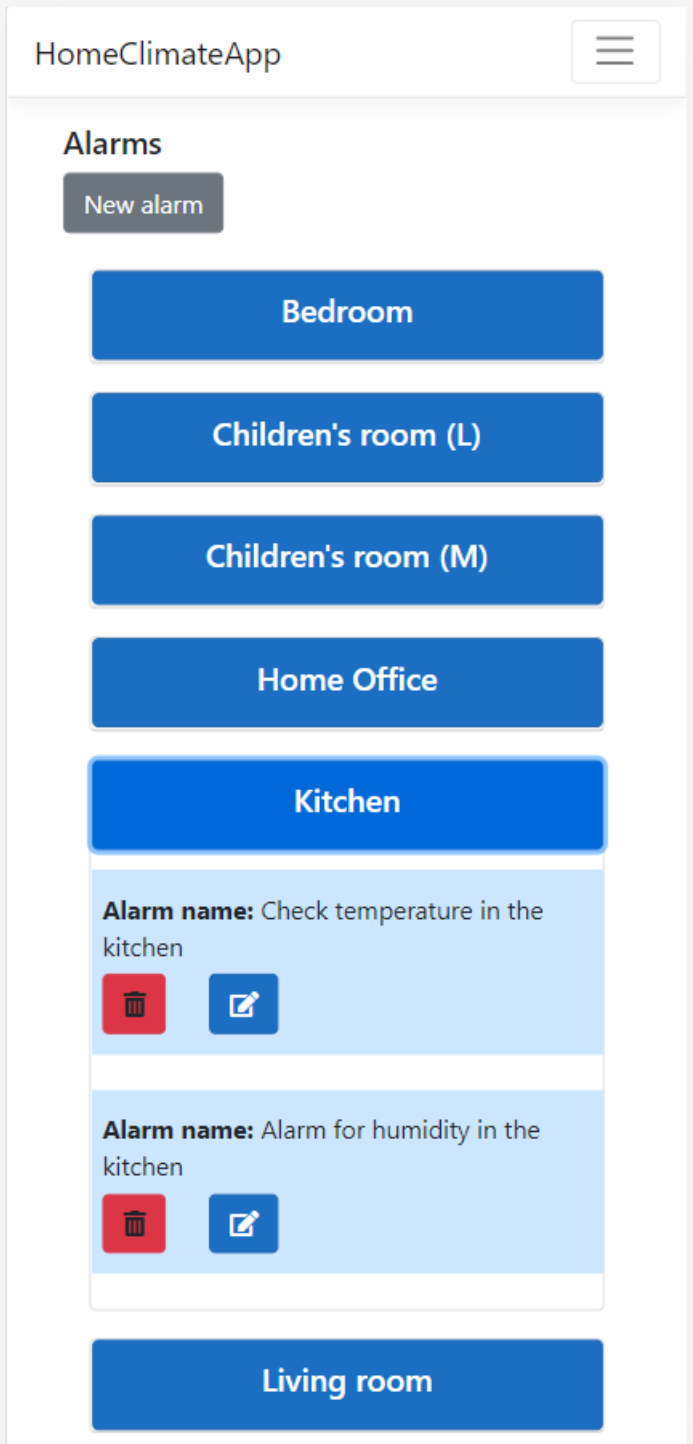
```
                label: 'Temperature',
                data: temperatureResults,
                backgroundColor: [
                    'rgba(0, 0, 230, 0.2)'
                ],
                borderColor: [
                    ...
                ],
                borderWidth: 1
            }
            ]
        },
        options: {
            scales: {
                yAxes: [{
                    ticks: {
                        beginAtZero: false,
                        max: maxTempValueForGraph,
                        min: minTempValueForGraph,
                        stepSize: 0.1
                    }
                }]
            }
        }
    });
```

Kummankin kaavion vaakatasoinen akseli (x -akseli) näyttää viimeisen seitsemän vuorokauden päivämäärät aloittaen aikaisimmasta päivämäärästä ja päättyen nykypäivään (esimerkkikoodissa päivämäärien arvot on annettu arvoiksi `labels` -propertylle/avaimelle). Pystysuuntainen akseli (y -akseli) näyttää mitta-asteikkoa – lämpötilakaavioissa mitta-asteikkona on ºC ja kosteustasokaaviossa mitta-asteikkona on %.

Raja-arvot mitta-asteikoille lasketaan dynaamisesti perustuen seitsemän vuorokauden matalimpaan ja korkeimpaan lukemaan. Kaavioissa käytetyt raja-arvot lämpötilan mitta-asteikoille (esimerkkikoodissa `maxTempValueForGraph` ja `minTempValueForGraph`), on laskettu vähentämällä 2 ºC ja lisäämällä 2 ºC vastaavasti minimi- ja maksimilukemaan. Esimerkkikoodissa y-akselin asteikon raja-arvot on määritelty `ticks.min` ja `ticks.max` -propertyjen kautta. Raja-arvot kosteustason mitta-asteikolle on laskettu vähentämällä 10% ja lisäämällä 10% vastaavasti minimi- ja maksimilukemaan.

3.6 Hälytysmoduuli

Hälytysmoduulin päänäkymä on Alarms -päänäkymä, joka esittää yhteenvedon kaikista olemassa olevista kohteista ja kohteille luoduista hälytyksistä (Kuva 13).



Kuva 13. Hälytysmoduulin Alarms -päänäkymä

Alarms -päänäkymän päätepiste on `/Alarms` ja se on määritelty `AlarmsController` -ohjaimen `Index` -nimiseksi oletusmetodiksi. Metodi on varustettu `[HttpGet]` -attribuutilla ja se palauttaa näkymälle välitettävän mallin. Malli täytetään datalla, jota näkymä tulee käyttämään. Metodin alussa LINQ -kyselylausekkeella haetaan tiedot olemassa olevista kohteista ja kohteille luoduista hälytyksistä. Kohteille luoduista hälytyksistä muodostetaan lista, joka välitetään `AlarmsDashboardVM` -malliluokan instanssille.

```
var alarmsDashboard = new AlarmsDashboardVM
{
    Alarms = allAlarmsForAllTargets
};
return View(alarmsDashboard);
```

`Index.cshtml` -näkymä on vahvasti tyypitetty näkymä, se on varustettu `@model` -direktiivillä ja käyttää `AlarmsDashboardVM` -näkymän mallia. Kyseinen näkymän malli kuvaa tiedot, jotka tulee käyttää `Index.cshtml` -näkymässä. `AlarmsDashboardVM` -näkymän mallin perusteella ohjain luo ja täyttää näkymän instanssit.

```
@model HomeClimateApp.Models.DashboardVM;
```

AlarmsDashboard -näkymä renderöidään Razor -syntaksia käyttäen ja sivun ulkoasu on toteutettu käyttäen Bootstrap -kirjastoa. Sivun ulkoasun toteutuksessa käytetään Bootstrap -kirjaston accordion -luokkaa. Jokaisen kohteen tiedot muun muassa kohteen nimi ja kohteen hälytysten tiedot esitetään Bootstrap -kirjaston card -luokan elementtien avulla.

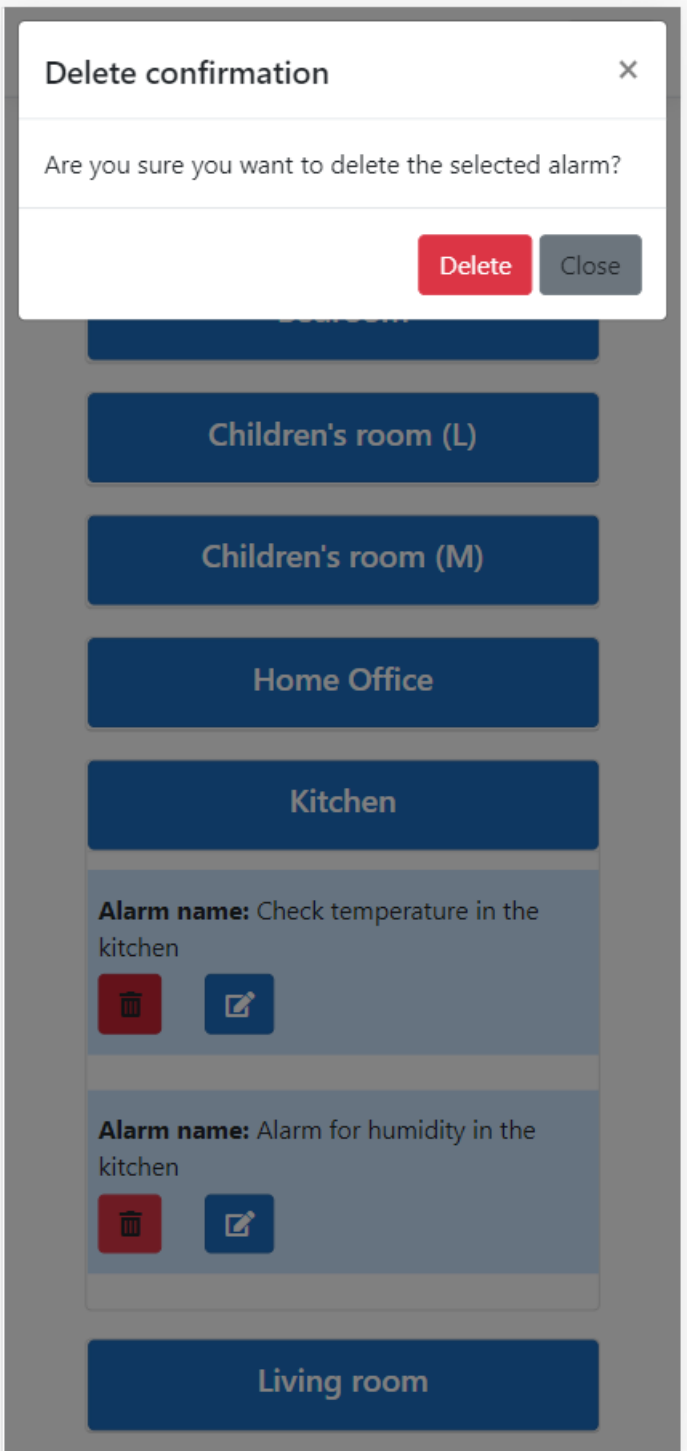
Jokaisen hälytyksen kohdalla on renderöity ”Muokkaa” ja ”Poista” -ikonit (Kuva 13). Elementtien renderöinnissä käytetään ASP.NET Core Tag Helpers -ominaisuutta. `asp-controller` ja `asp-action` -attribuuttien avulla ”Muokkaa” -ikonina kohdalla välitetään tiedot siitä, mille ohjaimelle ja mille ohjaimen metodille kutsu välitetään, kun käyttäjä klikkaa "Muokkaa" -ikonia. `asp-route-id` -attribuutin avulla välitetään tiedot siitä, mikä hälytys tulee muokata eli mihin instanssiin muokkauksen toimenpide kohdistuu.

```
<a class="btn btn-primary" role="button" asp-area="" asp-controller="Alarms" asp-
action="Edit" asp-route-id="@alarmsForTarget.AlarmId"><i class="fas fa-
edit"></i></a>
```

Hälytyksen poisto ei tapahdu suoraan klikkaamalla ”Poista” -ikonia. Ennen varsinaista poistoa käyttäjältä pyydetään poiston varmistus. Poiston varmistusdialogina sivun ulkoasussa on käytetty Bootstrap -kirjaston `modal` -niminen luokka (Kuva 14). `Modal` -luokan `<div>` -elementin sisällä `<form>` -elementtinä renderöidään lomake. Lomakkeen

metodi on ”post” ja lomake on varustettu `asp-controller, asp-action` ja `asp-route-id` -attribuuteilla.

```
<form method="post" asp-controller="Alarms" asp-action="Delete" asp-route-
id="@alarmsForTarget.AlarmId">
   <button type="submit" class="btn btn-danger">Delete</button>
   <a class="btn btn-secondary" role="button" data-dismiss="modal">Close</a>
</form>
```



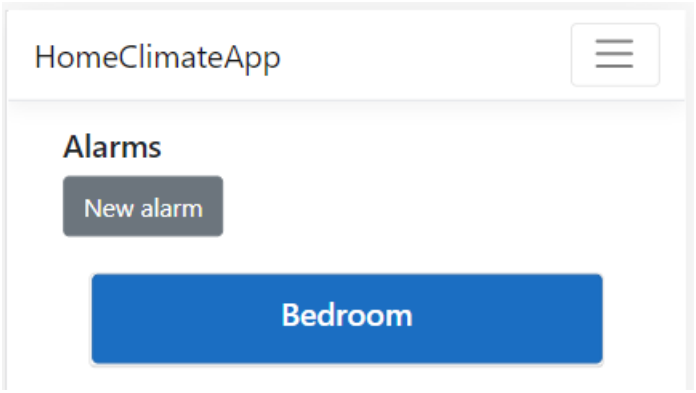
Kuva 14. Hälytyksen poiston varmistus

Kun käyttäjä varmistaa poiston klikkaamalla ”Delete” -painiketta `modal` -elementin sisällä, sovelluksen sisällä tapahtuu ohjaus `Alarms` -ohjaimen (`AlarmsController.cs`) `Delete` -metodille, joka on varustettu `[HttpPost]` -attribuutilla ja hoitaa valitun instanssin varsinaisen poiston tietokannasta. ASP.NET Core Tag Helpers `asp-route-id` -attribuutin avulla näkymä välittää `Delete` -metodille poistettavan halytyksen id:n. Mikäli tietokannasta

löytyy parametrina välitetty hälytyksen id, suoritetaan tietokantatransaktio, jolla tietokannasta poistetaan sekä varsinainen hälytys, että kaikki rivit, jotka linkittävät kyseisen hälytyksen käyttäjille. Jos yhdenkin rivin poisto tietokannasta ei onnistu, kaikki muutkin transaktiossa tehdyt muutokset peruuntuvat. Käyttämällä transaktiota tässä tapauksessa estetään rikkinäisten tietojen jäämistä tietokantaan ja tietokanta pysyy eheänä.

```
            using (var transaction = _context.Database.BeginTransaction())
            {
                try
                {
                    _context.Alarm.Remove(alarm);
                    _context.RemoveRange(alarm.UserAlarms);

                    await _context.SaveChangesAsync();
                    transaction.Commit();
                    return RedirectToAction(nameof(Index), new { deleted = true });
                }
                catch (DbUpdateException)
                {
                    transaction.Rollback();
                    return RedirectToAction(nameof(Index), new { id = id, saveChang-
esError = true });
                }
            }
```

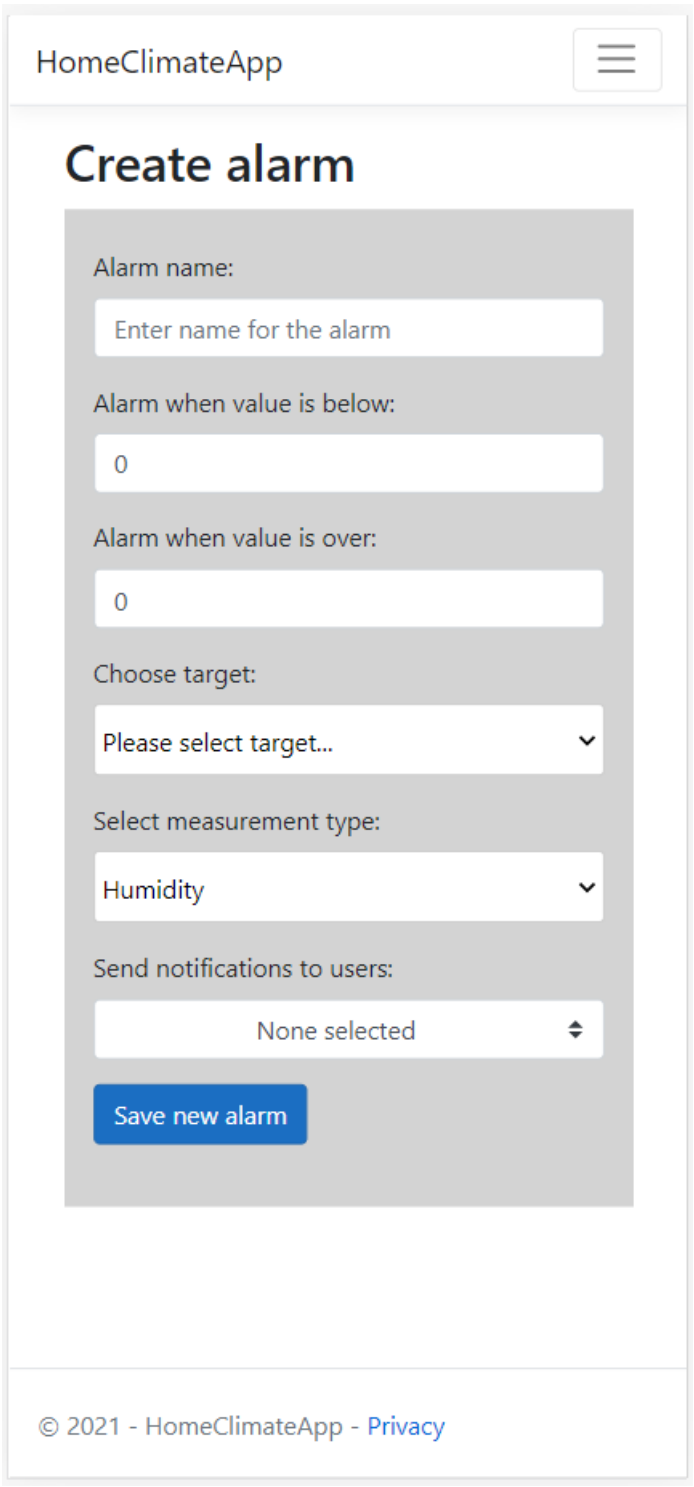
Klikkaamalla ”New alarm” -paniketta (Kuva 15) Alarms -päänäkymässä käyttäjä pääsee uuden hälytyksen luomisnäkymään (Kuva 16).



Kuva 15. "New alarm" -painike hälytysten yhteenvetonäkymässä

```
<a asp-action="CreateAlarm"><button class="btn btn-secondary">New alarm</button></a>
```

Painikkeen attribuuteista puuttuu `asp-controller` -attribuutti, mikä tarkoittaa, että painikkeen klikkaamisella käyttäjä ohjataan oletuksena samalle ohjaimelle, joka renderöi `Index` -päänäkymän.



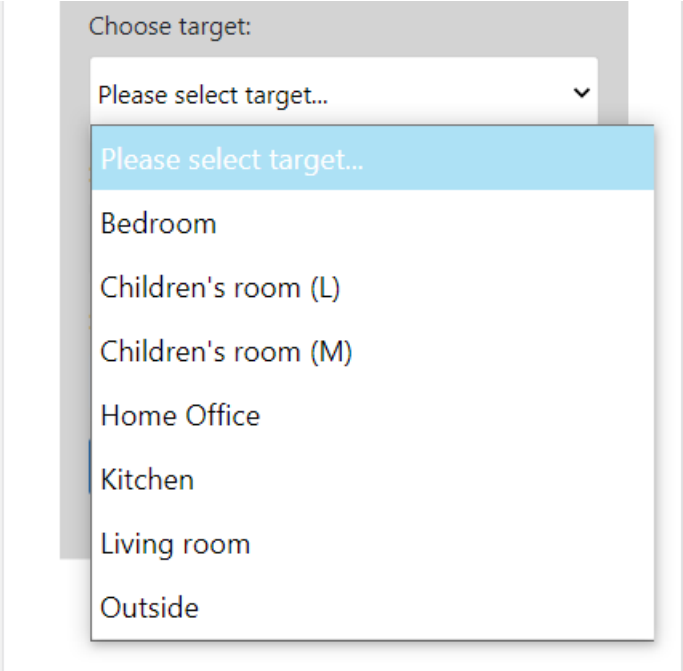
Kuva 16. Create Alarm -hälytyksen luomisnäkymä

`CreateAlarm.cshtml` -näkymän päätepiste on `/Alarms/CreateAlarm`. `CreateAlarm` -metodi on ylikuormitettu (method overloading). Ylikuormitetuista metodeista valitaan jokaisen HTTP -pyynnön yhteydessä se, minkä attribuutti täsmää HTTP metodin kanssa.

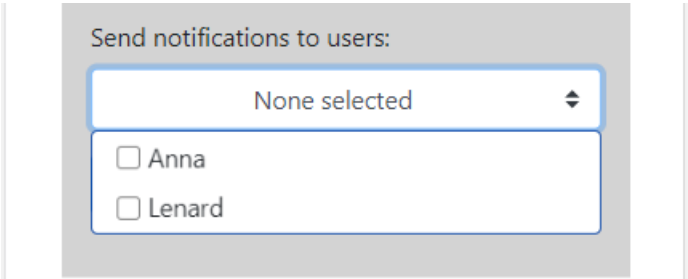
`[HttpGet]` -attribuutilla varustettu `CreateAlarm()` -metodi palauttaa `AlarmVM` -näkymämallin instanssin tiedoilla varustetun `CreateAlarm.cshtml` -näkymän, johon käyttäjä pääsee syöttämään uuden hälytyksen tiedot.

```
[HttpGet]
 public async Task<IActionResult> CreateAlarm()
 {
     var model = new AlarmVM();
     var availableTargets = await GetAllTargets();
     var allUsers = await GetAllUsers();
     ViewData["AvailableTargets"] = availableTargets;
     ViewData["Users"] = allUsers;

     return View("CreateAlarm", model);
 }
```

ViewData -Dictionary:n kautta `CreateAlarm` -näkymälle välitetään data, jota käytetään alasvetovalikoiden renderöinnissä. Tyhjälle näkymälle välitetään valmiiksi tiedot kantaan tallennetuista kohteista (Kuva 17) ja sovelluksen käyttäjistä (Kuva 18).



Kuva 17. Alasvetovalikko, jossa näkyy tietokantaan tallennetut kohteet



Kuva 18. Tietokantaan tallennetut käyttäjät

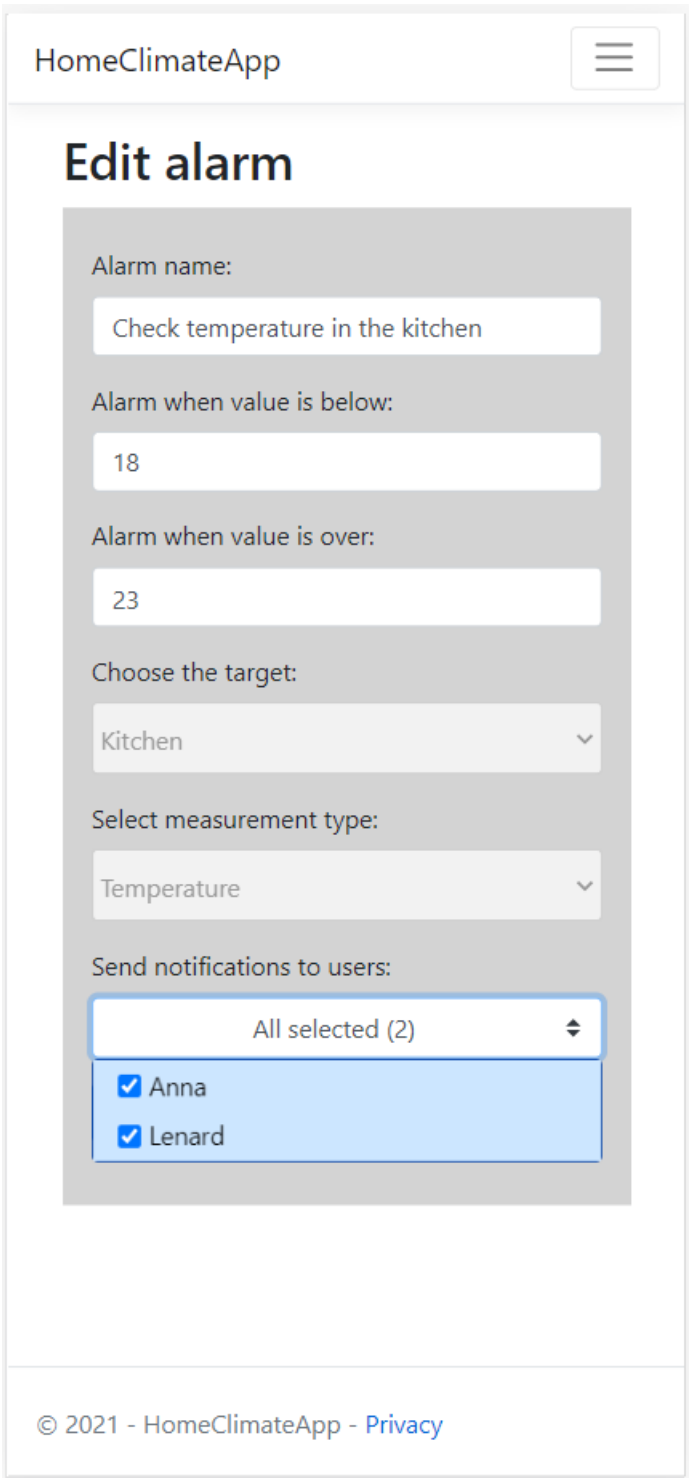
Alasvetovalikoiden sisältö CreateAlarm.cshtml -näkymässä renderöidään ASP.NET Core Tag Helpers -ominaisuuden avulla käyttämällä `asp-items` -attribuuttia.

```
            <div class="form-group">
                <label>Choose target:</label>
                <select asp-for="TargetId"
                        asp-items="@(new SelectList(availableTargets, "TargetId",
"TargetName"))">
                    <option>Please select target...</option>
                </select>
            </div>
```

`[HttpPost]` -attribuutilla varustettu `CreateAlarm` -metodi hoitaa hälytyksen varsinaisen käsittelyn ja tallennuksen kantaan. Metodi palauttaa uudelleenohjauksen tuloksen eli tässä tapauksessa onnistuneesti luodun hälytyksen jälkeen käyttäjä uudelleenohjataan `Index.cshtml` -päänäkymälle.

```
return RedirectToAction(nameof(Index));
```

`EditAlarm` -hälytyksen muokkausnäkymä (Kuva 19) renderöidään samalla periaatteella kuin `CreateAlarm` -näkymä, mutta muokkausnäkymässä hälytyksen tiedot ovat lomakkeessa valmiiksi esitäytetty valitun hälytyksen tiedoilla.



Kuva 19. Edit Alarm -hälytyksen muokkausnäkymä

Notifikaatioiden lähettäminen hälytysmoduulissa on toteutettu Pushover -ulkoisen järjestelmän avulla. Pushover -palvelu tarjoaa REST API -rajapinnan notifikaatioiden lähettämiselle mobiililaitteisiin.

Pushover-notifikaatioiden lähettäminen on toteutettu `NotificationSender.cs` -luokan implementointina. NotificationSender-palvelu käynnistetään sovelluksen käynnistämisen yhteydessä ajamalla `StartAsync` -metodia.

`StartAsync` -metodissa luodaan uusi instanssi `System.Threading.Timer` -oliosta, joka ajaa asynkronisen SendNotification -metodin 15 minuutin välein.

```
_timer = new Timer(SendNotification, null, TimeSpan.Zero, TimeSpan.FromMinutes(15));
```

Muokkaamalla FromMinutes -metodin parametrina annettua arvoa, pääsee säätämään notifikaatioiden lähetysintervaalia.

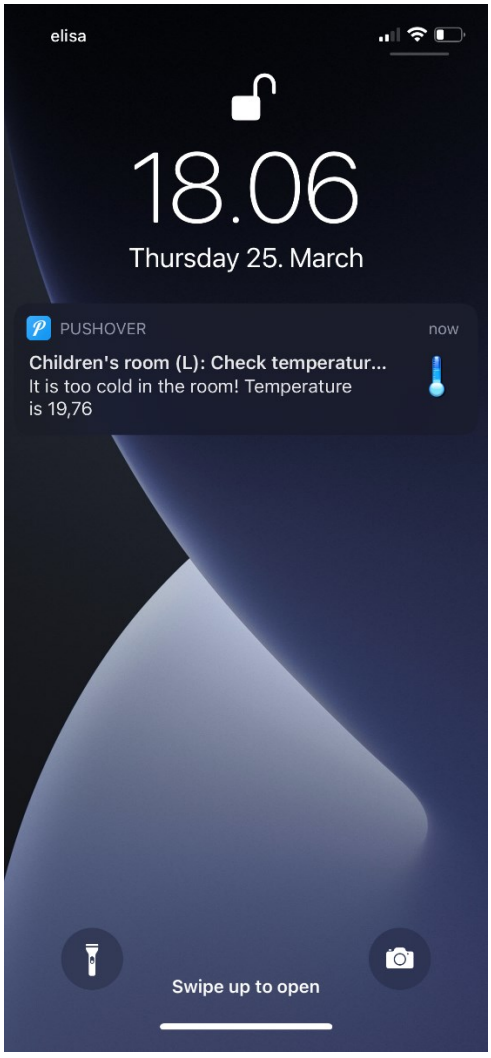
`SendNotification` -metodi, joka hoitaa varsinaiset notifikaatioiden lähetykset, joutuu ottamaan yhteyttä tietokantoihin sitä varten, että saisi tiedot siitä, mille käyttäjälle ja mitkä notifikaatiot tulee lähettää. Tietokantayhteyden avaaminen ja datan hakeminen tietokannasta oletuksena toteutetaan scoped service:na, kun taas HostedService -luokka on oletuksena singleton. Sitä varten, että HostedService -singleton palvelun sisällä olisi mahdollista hakea dataa tietokannasta, `NotificationSender` -luokalle annetaan konstruktorissa `IServiceProvider` -tyyppinen parametri, joka mahdollistaa scope:n luomisen singleton:n sisällä. `IServiceProvider`:n kautta avataan tietokantayhteydet tietojen hakua varten

```
        private async void SendNotification(object state)
        {

            using (var scope = provider.CreateScope())
            {


                var _context = scope.ServiceProvider.GetRequiredService<HomeClimate-
AppDbContext>();
                var _contextMariaDB = scope.ServiceProvider.GetRequiredService<Home-
ClimateAppMariaDb>();

...
```

Notifikaatioiden lähettäminen on toteutettu sovelluksessa niin, että ensin haetaan tietokannasta kaikki käyttäjät ja käyttäjien hälytykset. Sitten jokaiselle aktiiviselle hälytykselle haetaan lisätietoa: hälytyskohteen nimi, mitä mittaustyyppiä hälytys koskee ja

mitkä ovat sallitut arvot valitulle mittaustyypille. Saaduista tiedoista kasataan käyttäjien hälytysten lista eli tässä vaiheessa tapahtuu tietorakenteen muunnos – sovellukselle lähetetään useammasta tietokannan taulusta kerättyä oleellista tietoa sopivassa muodossa. Yhteneväisyyden takia käytetään ViewModel-tyyppistä luokkaa, vaikka tietoja ei varsinaisesti näytetä käyttäjälle.

Sitten kun on kasattu lista käyttäjistä ja käyttäjien asettamista hälytyksistä, rakennetaan varsinainen notifikaatiolista lähetettäväksi. Sitä varten tarkistetaan, mille huoneille on asetettu hälytykset ja haetaan tietokannasta viimeisimmät mittaustulostiedot valitulle huoneelle valitun mittaustyypin mukaisesti. Jos mittaustulos osuu ”hälytysalueelle”, mittaustuloksesta ja hälytyksestä muodostetaan notifikaatio, joka sitten lähetetään käyttäjälle (Kuva 20).



Kuva 20. Pushover notifikaatio

3.7 Ilmalämpöpumpun automaattisen ohjauksen moduuli

Vanhaan ilmalämpöpumppuun on hankittu Sensibo Sky -etäohjausjärjestelmä. Sensibo tarjoaa REST API rajapinnan, jonka kautta ohjelmallisesti voi ohjata ilmalämpöpumppua. Ilmalämpöpumpun automaattisen ohjauksen moduuli hyödyntää Sensibon tarjoamaa API:a ilmalämpöpumpun ohjaukseen ja käyttää siihen Flurl Fluent HTTP -kirjastoa.

Ilmalämpöpumpun automaattisen ohjauksen moduuli on toteutettu taustapalveluna (IHostedService) ja se käynnistetään Startup.cs -tiedostossa.

```
services.AddHostedService<HostedServices.SensiboConfigurator>();
```

Taustapalvelun käynnistys seuraa samoja periaatteita, kuin hälytysten taustapalvelu. Lisätietoja löytyy kohdasta 3.6 Hälytysmoduuli.

Ilmalämpöpumpun konfiguraatiotietojen automaattinen lähettäminen on toteutettu `SensiboConfigurator.cs` -luokan implementointina. Ensin Flurl Fluent HTTP -kirjastoa käyttäen lähetetään GET-pyyntö Sensibo API:lle, joka hakee ilmalämpöpumpun nykyiset asetukset ja palauttaa JSON-objektin.

```
var sensiboCurrentACState = await "https://home.sensibo.com/api/v2/pods/"
      .AppendPathSegment(deviceId)
      .AppendPathSegment("acStates")
      .SetQueryParams(new { fields = "*", apiKey = sensiboAPIkey })
      .GetJsonAsync();

bool currentOnOffSetting = sensiboCurrentACState.result[0].acState.on;
```

15 minuutin välein MariaDB -tietokannasta haetaan viimeisimmät mittaustulokset. Ylä- ja alakerran lämpötiloista riippuen ilmalämpöpumppu joko sammutetaan tai säädetään ilmalämpöpumpun asetuksia. Jos ilmalämpöpumppu on päällä, se sammutetaan lähettämällä PATCH-pyyntö Sensibo API:lle:

```
        private async Task SwitchOffAC(dynamic currentState, string deviceId,
string sensiboAPIkey)
       {

           if (currentState.result[0].acState.on == true)
           {
               string switchOffSucceeded = await "https://home.sen-
sibo.com/api/v2/pods/"
                   .AppendPathSegment(deviceId)
                   .AppendPathSegment("acStates")
                   .AppendPathSegment("on")
                   .SetQueryParams(new { fields = "*", apiKey = sensiboAPIkey })
                   .PatchJsonAsync(new {newValue = false })
                   .ReceiveString();
```

```
            _logger.LogInformation("AC switched off " + switchOffSucceeded);
        }
    }
```

Ilmalämpöpumpun asetusten säätämistä varten on toteutettu `ApplySettings` -metodi, joka ottaa parametreina tulevat asetukset, ilmalämpöpumpun tunnisteen ja API-avaimen. Uudet asetukset lähetetään Sensibo API:lle POST-pyynnön body:ssa.

```
private async Task ApplySettings(bool on_off,
                                 string mode,
                                 string fanSpeed,
                                 int temperature,
                                 string deviceId,
                                 string sensiboAPIkey)
    {
        string settingsApplied = await "https://home.sensibo.com/api/v2/pods/"
            .AppendPathSegment(deviceId)
            .AppendPathSegment("acStates")
            .SetQueryParams(new { fields = "*", apiKey = sensiboAPIkey })
            .PostJsonAsync(new
            {
                acState = new
                {
                    on = on_off,
                    mode = mode,
                    fanLevel = fanSpeed,
                    targetTemperature = temperature,
                    temperatureUnit = "C"
                }
            }).ReceiveString();

        _logger.LogInformation("New settings applied" + settingsApplied);
    }
```

Esimerkki ApplySettings -metodin kutsusta:

```
if (kitchenTemperature < 21 || livingRoomTemperature < 21)
     {
        await ApplySettings(true, "heat", "high", 24, deviceId, sensiboAPIkey);
     }
```

4 Johtopäätökset ja kehittämisehdotukset

Tässä luvussa käsitellään opinnäytetyöprojektin aikana saavutettuja tuloksia ja esitetään kehittämisehdotukset toteutetulle sovellukselle.

4.1 Johtopäätökset

Opinnäytetyöprojektin alussa asetetut tavoitteet on saavutettu. Opinnäytetyöprojektin päätavoitteeksi asetettu tavoite eli varsinainen sisäilmaa monitoroiva sovellus on toteutettu. Tämän lisäksi opinnäytetyöprosessin aikana on harjoiteltu käyttämään valittuja Microsoftin teknologioita, harjoiteltu semmoisen sovelluksen rakentamista, jossa liiketoimintalogiikkakerros on eroteltu tietokanta- ja näkymäkerroksista. Kokonaisuuden ymmärrystä ja osaamista on huomattavasti kasvatettu opinnäytetyöprosessin aikana.

Opinnäytetyöprojektia aloitettiin varsinaisen sovelluksen toteutuksella. Sen jälkeen, kun sovelluksen sisäilman monitoroinnin moduuli ja hälytysmoduuli olivat jo pitkällä toteutusvaiheessa, aloitettiin kirjoittamaan opinnäytetyön teoriaosuutta. Mielestäni tämä lähestymistapa oli hyvä kokonaisuuden ymmärtämisen näkökulmasta. ”Toteutus ensin” periaatteella tässä tapauksessa saavutettiin syvempää ymmärrystä sovelluksen rakentamisen periaatteista ja sovelluksen eri kerrosten yhteistoimivuudesta.

Toteutettu sovellus on räätälöity omiin tarpeisiin ja koostuu juuri niistä elementeistä, jotka taloudessamme tarvitaan tällä hetkellä. Opinnäytetyöprojektin aloittamisen hetkellä markkinoilla oli jo olemassa etäohjattavat ilmalämpöpumput, mutta toteutetun sovelluksen myötä uutta ilmalämpöpumppua ei tarvinnut hankkia. Sovelluksen ilmalämpöpumpun automaattisen ohjauksen moduulin kautta taloutemme vanhaa ilmalämpöpumppua voi ohjata automaattisesti perustuen eri huoneissa mitattuihin lukemiin. Toteutettu sovellus ratkaisee perheemme ongelmia talviaikaan. Sovelluksen kautta sisäilman monitorointi on huomattavasti helpompaa ja antaa mahdollisuudet nopeaan reagointiin mikäli sisäilman optimaaliset olosuhteet huoneistossa eivät toteudu.

Sovellusta voisi hyödyntää muissakin talouksissa kuten esimerkiksi ikääntyvien ihmisten talouksissa. Tutkimusten mukaan kesäiset korkeat lämpötilat ovat vaarallisia ikääntyville ja pitkäaikaissairaille ihmisille. Maaliskuussa 2019 Suomen Terveyden ja hyvinvoinnin laitos julkaisi tiedotteen, jossa kehottaa varautumaan helteisiin jo ennen niiden alkua. Terveyden ja hyvinvoinnin laitos suosittelee viilennysjärjestelmien käyttöä helteiden aikana ikääntyneiden ja pitkäaikaissairaiden ihmisten oleskelutiloissa kuten kodeissa ja hoitolaitoksissa. (Kollanus 2019.) Tämän takia ikääntyneillä ja pitkäaikaissairailla ihmisillä

olisi hyvä olla helppo keino seurata kodin olosuhteita. Opinnäytetyöprojektin aikana toteutettu sovellus mahdollistaa helppoa kodin olosuhteiden seurantaa.

Yksi opinnäytetyöprojektin tavoitteista oli kasvattaa valittujen Microsoftin teknologioiden osaamista. Kyseinen tavoite on myös saavutettu, osaamisen taso on huomattavasti kasvanut sovelluksen toteutuksen myötä. Opinnäytetyöprojektin aikana hankittua osaamista voi hyödyntää tulevaisuudessa ohjelmistokehittäjän tehtävissä. Opinnäytetyön kirjoittamisen hetkellä Micrisoftin teknologioiden kuten esimerkiksi .NET Core osaajista on pula.

4.2 Kehittämisehdotukset

Yhtenä sovelluksen jatkokehityksen osa-alueena voi olla käyttöliittymätuen toteutus kaikille komponenteille, joita sovelluksessa esitetään. Tällä hetkellä kohteiden lisääminen tietokantaan ei ole tuettu käyttöliittymässä. Kohteiden tietojen muokkausta (esimerkiksi kohteen nimen muokkausta) ei ole mahdollista tehdä käyttöliittymän kautta. Käyttäjätietojen lisääminen tietokantaan käyttöliittymän kautta ei ole myöskään tuettu sovelluksen ensimmäisessä versiossa. Jotta sovellus pystyisi lähettämään notifikaatiot käyttäjälle, tällä hetkellä käyttäjätiedot joudutaan syöttämään suoraan tietokantaan. Käyttäjätietojen syöttäminen tietokantaan käyttöliittymän kautta olisi myös hyvä mahdollistaa seuraavassa sovelluksen versiossa.

Tärkeänä jatkokehityksen voisi olla sovelluksen käyttöliittymän ulkoasun parantaminen. Opinnäytetyöprojektin puitteissa toteutettu sovelluksen ensimmäinen versio käyttää yksinkertaista ulkoasua, jonka taustana on Bootstrap -kirjasto. Seuraavassa versiossa olisi hyvä toteuttaa sovellukselle modernimpi ja käyttäjäkokemukseltaan kehittyneempi ulkoasu.

Sovellusta voi laajentaa lisäämällä mahdollisuuden seurata lämpötilan ja kosteusasteen muutoksia pidemmältä ajalta, esimerkiksi kuukausitasolla. Sovellukseen voisi myös lisätä tuki muille etäohjattaville komponenteille, kuten esimerkiksi etäohjattaville pistorasioille. Silloin ilmankostuttimet voisi kytkeä etäohjattaviin pistorasioihin ja sovellus voisi automaattisesti laittaa ilmankostuttimet päälle tai kytkeä ne pois päältä, kun kosteustaso kohteessa alkaa olla kriittisessä pisteessä.

Talouden energiansäästötoimenpiteiden kannalta olisi hyvä saada ilmalämpöpumpun automaattisen ohjauksen moduuliin toiminnallisuus, joka käyttäjien Bluetooth:ien avulla osaa detektoida, onko käyttäjiä kotona vai ei ja sen mukaisesti säätää asunnon lämpötilaa matalammalle tai korkeammalle.

Kaikkien yllämainittujen laajennusten ja täydennysten lisäksi, sovellukseen olisi hyvä kehittää ja ottaa käyttöön virhekäsittelyprosessi ja käyttäjien autentikaatio. Jos sovelluksen haluttaisiin tuotteistaa, olisi hyvä miettiä läpi ja kehittää sovelluksen käyttöönottoprosessi ja tuki useammalle taloudelle mahdollisesti ”software as a service” palveluna.

Lähteet

Addie, S., Kellner, P., 2019. Anchor Tag Helper in ASP.NET Core. Luettavissa: https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/built-in/anchor-tag-helper?view=aspnetcore-5.0. Luettu: 7.6.2020

Anderson, R., 2019. Tag Helpers in ASP.NET Core. Luettavissa: https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/intro?view=aspnetcore-5.0. Luettu: 11.12.2020

Anderson, R., Dykstra, T. Part 4, add a model to an ASP.NET Core MVC app. Luettavissa: https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-model?view=aspnetcore-5.0&tabs=visual-studio#strongly-typed-models-and-the--keyword. Luettu: 11.12.2020.

Anderson, R., Luttin, S., Roth, D., 2020. Introduction to ASP.NET Core. Luettavissa: https://docs.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-5.0. Luettu: 12.7.2020.

Anderson, R. & Smith, S. 2020. ASP.NET Core Middleware. Luettavissa: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-5.0. Luettu: 23.1.2021.

Freeman, A. 2020. Pro ASP.NET Core 3. Develop Cloud-Ready Web Applications Using MVC, Blazor, and Razor Pages. Apress. London UK.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 2005. Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley. USA.

Heikniemi, R. 9.4.2021. Senior Consultant. Devisioona Oy. Keskustelu. Helsinki.

Hengitysliitto. Sisäilman kosteus ja lämpötila. Luettavissa: https://www.hengitysliitto.fi/fi/sisailma/sisailma-asiat-sisailmaongelmat/sisailman-kosteus-ja-lampotila. Luettu 06.09.2019

Kollanus, V., 2019. Viime kesän helleaalto lisäsi ikääntyneiden kuolleisuutta – helteisiin on hyvä varautua ajoissa. Terveyden ja hyvinvoinnin laitos. Luettavissa: https://thl.fi/fi/-/viime-kesan-helleaalto-lisasi-ikaantyneiden-kuolleisuutta-helteisiin-on-hyva-varautua-ajoissa. Luettu: 9.9.2019

Lampi, J. & Pekkanen, 2008. Terve ihminen terveissä tiloissa. Kansallinen sisäilma ja terveys - ohjelma 2018–2028. Terveyden ja hyvinvoinnin laitos, 8, s. 23. Luettavissa: http://www.julkari.fi/bitstream/handle/10024/137064/THL_RAP2018_8_sis%c3%a4ilma%20ja%20terveys_WEB_250319pdf.pdf?sequence=1&isAllowed=y. Luettu: 7.9.2019.

Larkin, K., 2019. Model validation in ASP.NET Core MVC and Razor Pages. Luettavissa: https://docs.microsoft.com/en-us/aspnet/core/mvc/models/validation?view=aspnetcore-5.0. Luettu: 13.1.2020.

Li Huan, J. 2020. Background tasks with hosted services in ASP.NET Core. Luettavissa: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/host/hosted-services?view=aspnetcore-2.2&tabs=visual-studio#consuming-a-scoped-service-in-a-background-task. Luettu: 20.12.2020.

Michaelis, M., 2018. Essential C# 7.0. Addison-Wesley. USA.

Microsoft 2016. Language Integrated Query (LINQ). Luettavissa: https://docs.microsoft.com/en-us/dotnet/csharp/linq/. Luettu 13.12.2020.

Microsoft 2020. Razor syntax reference for ASP.NET Core. Luettavissa: https://docs.microsoft.com/en-us/aspnet/core/mvc/views/razor?view=aspnetcore-5.0. Luettu: 12.02.2021

Microsoft 2020. .NET Core and .NET 5. Luettavissa: https://docs.microsoft.com/en-us/dotnet/core/introduction#net-core-and-net-5. Luettu 5.11.2020

Smith, S. 2020. Overview of ASP.NET Core MVC. Luettavissa: https://docs.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-5.0#features. Luettu: 15.11.2020.

Smith, S., Brock, D., 2019. Layout in ASP.NET Core. Luettavissa: https://docs.microsoft.com/en-us/aspnet/core/mvc/views/layout?view=aspnetcore-5.0. Luettu: 23.12.2020.

Toub, S. 2020. Performance Improvements in .NET 5 13.07.2020. Luettavissa: https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-5/. Luettu: 19.2.2021.

Valvira 2016. Asumisterveysasetuksen soveltamisohje. Osa I. Asumisterveysasetus § 1-10, s. 12. Luettavissa:

https://www.valvira.fi/documents/14444/261239/Asumisterveysasetuksen+soveltamisohje/ac8d5e16-97be-456c-9c9c-ce8560f2092e. Luettu: 6.9.2019.

Wikipedia. .NET Core. Luettavissa: https://en.wikipedia.org/wiki/.NET_Core. Luettu: 3.12.2020.

Wikipedia. Representational state transfer. Luettavissa: https://en.wikipedia.org/wiki/Representational_state_transfer. Luettu: 5.3.2021.