Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Ville Kerminen

SaaS-palveluiden valvontajärjestelmä

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikka

Insinöörityö

20.5.2020

Tekijä Otsikko Sivumäärä Aika	Ville Kerminen SaaS-palveluiden valvontajärjestelmä 45 sivua 20.5.2020
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tietotekniikka
Ammatillinen pääaine	Ohjelmistotekniikka
Ohjaajat	Lehtori Simo Silander Tuoteomistaja Sami Niemisalo

Insinöörityön tavoitteena oli tuottaa Mepco-tuoteperheelle räätälöity valvonta- ja ylläpitojärjestelmä. Järjestelmän tarkoituksena oli parantaa sovellusten toimintavarmuutta ja tietoturvaa varmistamalla, että palvelimien ja sovellusten konfiguraatiot ovat kunnossa ja määriteltyjen suositusten mukaisia.

Aluksi työssä kuvataan valvontajärjestelmän tarve, esitellään sidosryhmät sekä määritellään asiat, joihin sen tulisi tarjota ratkaisuja. Lisäksi käydään läpi muutamia käyttötapauksia. Lopuksi kuvataan järjestelmän rakenne sekä käydään läpi sen toteutukseen käytettyjä tekniikoita ja itse toteutusta.

Valvontaportaalin ja tilasivujen selainkäyttöliittymä ohjelmoitiin TypeScript-ohjelmointikielellä hyödyntäen Vue.js-kirjastoa. Valvontaportaalin palvelintoteutukset ohjelmoitiin C#-kielellä hyödyntäen ASP.NET Core -ohjelmistokehystä ja Entity Framework Coren tarjoamaa tietokantarajapintaa.

Lopputuloksena syntyi järjestelmä, joka koostuu keskitetystä valvontaportaalista, johon tila- ja virhetiedot kerätään ja josta niitä voi selata selaimella sekä sovelluskirjastosta, jonka avulla sovellusten ja palvelimien konfiguraatio validoidaan ja raportoidaan tilatietoina valvontaportaalille. Ohjelmakirjasto voidaan liittää osaksi eri Mepco-tuotteita ja sitä käytetään myös palvelimen tilatietojen raportoinnissa.

Avainsanat	SaaS, sovellusten valvonta, Windows Server -kovennus, ASP.NET Core, Entity Framework Core, Vue.js

Author Title Number of Pages Date	Ville Kerminen Monitoring System for SaaS Services 45 pages 20 May 2020
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technology
Professional Major	Software Engineering
Instructors	Simo Silander, Senior Lecturer Sami Niemisalo, Product Owner

The purpose of this thesis was to create a tailored monitoring and maintenance system for the Mepco product family. The purpose of the system was to improve the reliability and security of the applications by ensuring that the configurations of the servers and applications are in order and in accordance with the defined recommendations.

At first, the thesis describes the need for the monitoring system, introduces the user groups and identifies issues for which it should provide solutions. In addition, a few use cases are reviewed. Finally, the structure of the system is described, as well as the technologies used for its implementation and the implementation itself.

The browser interface for the monitoring portal and status pages was programmed in the TypeScript programming language utilizing the Vue.js framework. In the server implementations of the monitoring portal were programmed in C# using the ASP.NET Core software framework and the database interface provided by Entity Framework Core.

The result was a system consisting of a monitoring portal where status and error information is collected and where they can be browsed with a browser, and an application library that validates and reports the configuration of applications and servers as status information to the monitoring portal. The program library can be integrated into various Mepco products and it is also used to report server status information.

Keywords	SaaS, software monitoring, Windows Server hardening, ASP.NET Core, Entity Framework Core, Vue.js

Sisällys

Lyhenteet ja käsitteet

AD	Active Directory. Microsoft Windowsin toimialueen käyttäjätietokanta ja hakemistopalvelu.
ADFS	Active Directory Federation Services. Active Directory -liittoutumispalvelu, joka helpottaa kertakirjautumisen toteuttamista Windows-käyttäjähakemiston päälle.
API	Application Programming Interface. Ohjelmointirajapinta.
C#	C Sharp. Microsoftin .NET-alustoille kehittämä ohjelmointikieli.
CIS	Center for Internet Security. Voittoa tavoittelematon organisaatio, joka pyrkii edistämään sovellusten turvallisuutta.
CSP	Content Security Policy. HTTP-otsake, jolla voidaan rajata lähteitä, joista selain saa ladata sivulle sisältöä.
CSRF	Cross-Site Request Forgery. Hyökkäystyyppi, jossa pyritään saamaan järjestelmään kirjautunut käyttäjä tekemään ei-toivottuja kutsuja kohdejärjestelmään lähettämällä kutsuja toisen verkkosivun koodista.
CSS	Cascading Style Sheets. Web-dokumenteissa käytetty tyyliohjeiden määrittelykieli.
EF	Entity Framework. Olio-relaatiomuunnin, jonka avulla .NET-sovellus voi käsitellä tietokantaa ohjelmakoodista käsin.
HSTS	HTTP Strict Transport Security. HTTP-otsake, joka varmistaa, että selain kutsuu sivua aina salatulla yhteydellä HTTPS:n yli.
HTML	Hypertext Markup Language. Hypertekstin standardisoitu merkintäkieli, jota käytetään web-dokumenttien muodostamiseen.

HTTP	Hypertext Transfer Protocol. Protokolla, jota käytetään selaimen ja WWW-palvelimen väliseen tiedonsiirtoon.
HTTPS	Hypertext Transfer Protocol Secure. HTTP-protokollan versio, jossa tiedonsiirto on salattu.
IIS	Internet Information Services. Windowsin sisäänrakennettu web-palvelinohjelmisto.
JSON	JavaScript Object Notation. Yksinkertainen ja avoin tiedostomuoto tiedonvälitykseen.
JWE	JSON Web Encryption. JSON-rakenteiden salaamiseen käytetty tekniikka.
OAuth	Open Authorization. Avoin auktorisointiprotokolla käyttäjän valtuuttamiseen.
OIDC	OpenID Connect. OAuth2:n päälle rakennettu autentikointitaso.
ORM	Object-relational mapping / olio-relaatio-mallinnus. Tapa yhteensovittaa oliomalli relaatiomallin kanssa.
OWASP	Open Web Application Security Project. Voittoa tavoittelematon organisaatio, joka pyrkii edistämään erityisesti verkkosovellusten turvallisuutta.
REST	Representational State Transfer. Ohjelmointirajapintojen (API) toteuttamiseen käytetty arkkitehtuurimalli.
SaaS	Software as a Service. Palveluna hankittu sovellus.
SAML	Security Assertion Markup Language. Standardi käyttäjän tunnistautumisen ja valtuutuksen välittämiseen tietojärjestelmien välillä.
SLA	Service Level Agreement. Palvelutasosopimus, jossa määritellään palvelun vaatimustasot.

SQL	Structured Query Language. Relaatiotietokantojen kyselykieli.
SPA	Single-page Application. Web-sovelluskehitystapa, jossa sivulle ladataan käyttäjän toimien perusteella tarvittavat näkymät ilman koko sivun päivitystä.
TFS	Team Foundation Server. Microsoftin tuote, joka tarjoaa versiohallinnan, tiketöinnin ja muita sovelluskehitystä helpottavia toimintoja.
TLS	Transport Layer Security. Salausprotokolla, jolla voidaan suojata tiedonsiirtoa.

1 Johdanto

Opinnäytetyön aiheena oli toteuttaa Accountor HR Solutions Oy:n Mepco-tuotteille räätälöity keskitetty valvontajärjestelmä.

Accountor HR Solutions Oy (AHR) on Suomessa toimiva yritys, joka kehittää Mepco-tuoteperheen palkka-, HR-, tuntikirjaus- ja rekrytointijärjestelmiä. Yritys työllistää noin 130 työntekijää Espoossa, Tampereella, Turussa ja Jyväskylässä. AHR kuuluu Accountor-konserniin, joka on erikoistunut talous- ja henkilöstöhallinnon sekä ICT-ratkaisuihin ja ulkoistuspalveluihin. Accountor toimii 7 maassa ja työllistää noin 2500 työntekijää.

Toteutettavan keskitetyn valvontajärjestelmän oli tarkoitus helpottaa sovellusasennusten valvontaa, parantaa reagointinopeutta ongelmatilanteisiin. Räätälöidylle järjestelmälle on tarve, koska Mepco-tuotteiden asennuksia on lukuisia, ja sitä on asennettu niin asiakkaiden omille palvelimille kuin myös AHR:n ja emoyhtiö Accountorin ohjelmistopalvelun palvelimille, eli SaaS-palvelimille (Software as a Service).

Valvontajärjestelmän ensimmäisiä vaiheita aloitettiin 2017 vuoden alussa, jolloin aloin hahmottelemaan sovelluksille niin kutsuttua tilasivua, jonka tietoihin valvontajärjestelmäkin osin tukeutuu.

Tarkoituksena oli toteuttaa selaimella käytettävä keskitetty järjestelmä, jonka kautta pystyisi näkemään eri palvelimille asennettujen sovellusten tilat ja mahdolliset ongelmat. Järjestelmän oli tarkoitus lisätä asennusten tietojen läpinäkyvyyttä, ajantasaisuutta ja siirtää valvonta passiivisesta valvonnasta proaktiiviseen, eli ennakoivaan, valvontaan sekä varmistaa asennusten tietoturvallisuus.

Järjestelmä toteutettiin käyttäen ASP.NET Core -ohjelmistokehystä ja C#-ohjelmointikieltä. Tietokantapuolella käytössä on Microsoft SQL Server, jota käytetään Entity Framework Core -rajapintakerroksen avulla. Web-käyttöliittymässä käytetään pääkirjastona Vue.js-kirjastoa, jonka koodi kirjoitettiin TypeScriptillä.

2 Valvontajärjestelmän tarve - lähtökohdat

Tarve räätälöidyn valvontajärjestelmän tekemiselle syntyi, kun yrityksellä ei ollut käytössä kunnon valvontajärjestelmää, jolla sovellustason monitorointia tai vianselvitystä olisi voinut tehdä riittävän hyvin. Käytössä oli vain erillisiä järjestelmiä, jotka monitoroivat lähinnä palvelinten tilaa mm. tietoturvapäivitysten ja muiden korkean tason tunnuslukujen osalta. Tarve on myös pystyä reagoimaan ja ratkaisemaan nopeammin ongelmatilanteita, jotta voidaan täyttää paremmin SLA-sopimusten (Service Level Agreement) ehdot ja tarjota parempi asiakaskokemus.

Vuonna 2017 alettiin kehittää alustavaa ratkaisua ongelmaan. Sovelluksiin alettiin kehittää selaimella käytettävää julkista tilasivua, josta sovelluksen ja sen käyttämien riippuvuuksien tilan voi tarkistaa esimerkiksi asennuksen, päivityksen tai ongelmatilanteen aikana. Koska näitä tiloja ei ole monitoroitu automaattisesti, niin niistä on yleensä hyötyä vain päivityksissä tai vasta, kun joku on jo raportoinut ongelmasta. Valvontajärjestelmän tulisi tarjota ratkaisu tähän ongelmaan ja valvoa näitä tilatietoja keskitetysti. Valvontajärjestelmän myötä valvonta muuttuisi passiivisesta proaktiiviseksi, eli ennakoivaksi, valvonnaksi. Jatkuva valvonta ja hälytykset ongelmatilanteista nopeuttaisivat ongelmien havaitsemista sekä nopeuttaisi niihin reagointia ja siten helpottaisi SLA-sopimusten ehtojen täyttämistä.

Valvontajärjestelmän ja ennakoinnin tarvetta lisää se, että sovelluksia on asennettuna lukuisilla eri palvelimilla ja niiden päivitykset ja konfiguroinnit hoidetaan nykyisellään pääasiassa käsin asentajan toimesta. Tämä aiheuttaa riskin inhimilliselle virheelle siinä, että kaikkia suositeltuja konfiguraatioita ei huomata ottaa käyttöön, kun ympäristöjä päivitetään tai muunnetaan esim. testi- tai esituotantoympäristöstä varsinaiseksi tuotantoympäristöksi. Lisäksi tietoturvan kannalta tärkeitä uusia asetuksia ja käytäntöjä tulee jatkuvasti lisää, joten näiden käyttöönoton valvontaan on syytä olla jokin automaattinen tapa. Väärä konfiguraatio onkin yksi OWASP:n (Open Web Application Security Project) TOP 10 2017 -listan kymmenestä kohdasta (1 s. A6:2017-Security Misconfiguration). OWASP TOP 10 -lista koostuu kymmenestä kriittisimmäksi luokitellusta web-ohjelmistokehityksen haavoittuvuustyypistä ja tarjoaa suosituksia niiden välttämisestä.

Sovelluspäivityksiä on tarkoitus alkaa myöhemmin automatisoimaan, jolloin valvontajärjestelmä voi auttaa automaatiota varmistamaan, että sovelluksen päivitys on onnistunut.

Lisäksi valvontajärjestelmä osaa hälyttää, mikäli automaatioasennuksen tekemät päivitykset eivät ole onnistuneet, jolloin ongelmaan voidaan reagoida nopeasti.

3 Mepco-tuotteet

Mepco on Suomen suosituin HR- ja palkanlaskentaohjelmisto. Mepco-tuotteet on kehitetty helpottamaan työsuhteiden elinkaaren hallintaa. Tuotteilla voidaan hallita työsuhteen elinkaaren vaihteita, kuten rekrytointi, palkanmaksu, henkilöstöhallinto ja tuntikirjaus. (2.)

Tämä luku sisältää lyhyet esittelyt Mepco-tuotteista ja niiden sovelluskomponenteista.

3.1.1 Mepco HRM & Mepco PRO

Mepco HRM on yksityisen sektorin palkanlaskennan ja henkilöstöhallinnon järjestelmä. Mepco PRO taas on julkisen sektorin tarpeisiin räätälöity versio, joka on kehitetty Mepco HRM:n pohjalta.

Mepco HRM:n sovelluskomponentit:

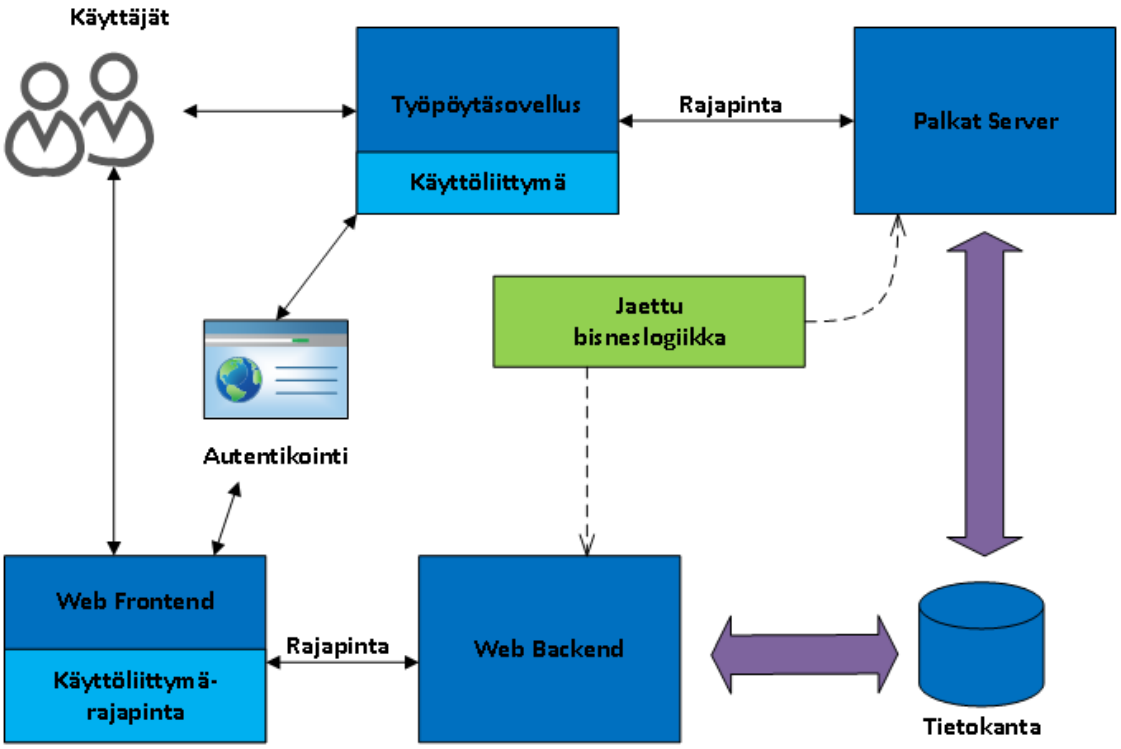
- Windows-työpöytäsovellus – Smart Client
- Palkat Server
- PRO Web Backend
- PRO Web Frontend.

Mepco PRO:n sovelluskomponentit:

- Windows-työpöytäsovellus – PRO Client
- PRO Server
- PRO Web Backend
- PRO Web Frontend.

Palkat Server ja PRO Server ovat Windows-työpöytäsovelluksen taustapalveluita, jotka tarjoavat tietokantayhteydet ja bisneslogiikan. Vastaavasti Web Backend -sovellukset

tarjoavat rajapinnan kautta pääsyn bisneslogiikkaan ja tietokantaan. Web Frontend -sovellukset tarjoavat käyttöliittymän tarvitseman rajapinnan. Käyttöliittymäsovellukset huolehtivat autentikoinnista konfiguroitua kirjautumistapaa käyttäen. Näiden sovelluskomponenttien välisiä riippuvuuksia havainnollistetaan kuvassa 1.



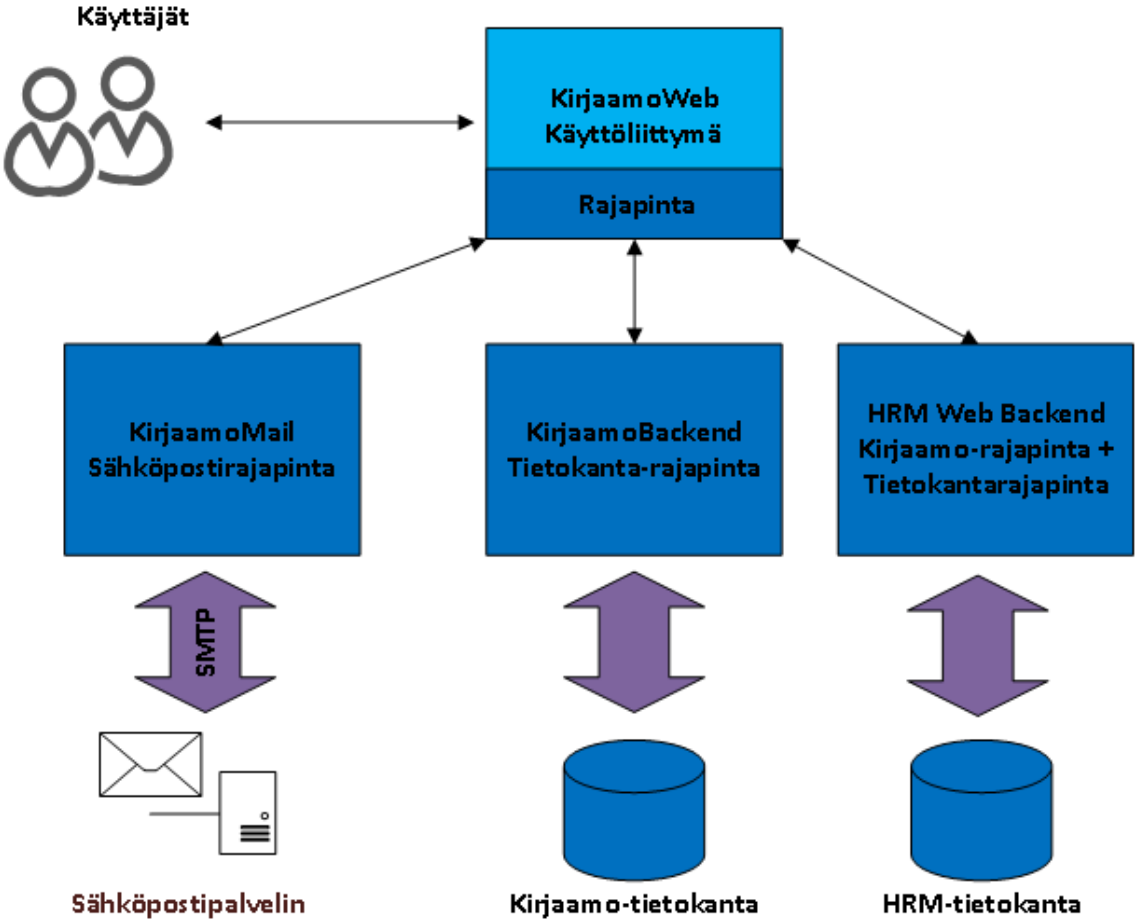
Kuva 1. Mepco-sovellusten sovellusrakenne yksinkertaistettuna. Molempien sovellusten taustapalvelinsovellus (Palkat Server ja Web Backend) käyttävät osin yhteistä koodia jaetun bisneslogiikkakirjaston kautta. Molemmat käyttöliittymäsovellukset huolehtivat autentikoinnista ja sen välittämisestä taustapalvelinsovellukselle.

3.1.2 Kirjaamo

Kirjaamo on Mepco HRM- ja Mepco PRO -järjestelmiin rajapintojen avulla integroituva tunti- ja tapahtumakirjausjärjestelmä. Hyväksytyt kirjaukset siirtyvät HRM-järjestelmiin palkka- ja tilastotapahtumiksi.

Kirjaamon sovelluskomponentit (kuva 2):

- KirjaamoWeb (Frontend) – käyttöliittymän julkinen rajapinta
- KirjaamoBackend – tietokantayhteydet tarjoava rajapinta
- KirjaamoMail – sähköpostien lähetyksen mahdollistava rajapinta
- HRM Web Backend / PRO Web Backend:n tarjoama Kirjaamo-rajapinta.



Kuva 2. Kirjaamon sovellusrakennetta havainnollistava kuva.

3.1.3 Autentikointi

Mepcon autentikointisovellus toimii yhteensovittavana välikomponenttina erilaisten ulkoisten kirjautumistapojen ja Mepco-sovelluksien välissä, ja välittää kirjautumistiedot OIDC (OpenId Connect) -tekniikalla eri Mepco-sovelluksille.

Autentikointisovellus tukee eri kirjautumistekniikoita:

- Windows-kirjautuminen
- SAML-versiot (Security Assertion Markup Language) 1.1 ja 2.0
- OIDC.

SAML on avoin käyttäjätunnistusprotokolla, jossa käyttäjätiedot kuljetetaan XML-muodossa. OIDC puolestaan on uudempi Oauth2-pohjainen käyttäjätunnistusprotokolla, jossa tiedot kuljetetaan web-kehityksen kannalta modernimmin JSON-muodossa.

Näiden tekniikoiden avulla voidaan toteuttaa kirjautuminen eri kirjautumistavoin asiakastarpeen mukaan muun muassa:

- AD (Active Directory), joka tarkoittaa Windows-toimialuekirjautumista käyttäen käyttäjän tunnuksia, joilla käyttäjä on kirjautunut Windows-tietokoneelle.
- ADFS (Active Directory Federation Services), eli Active Directory -liittoutumispalvelu, joka laajentaa Windows-toimialueen kertakirjautumisominaisuuksia.
- Azure AD, joka on Microsoftin kehittämä pilvipohjainen Active Directory.
- Haka, joka on monien Suomen korkeakoulujen ja tutkimuslaitosten käyttämä käyttäjätunnistusjärjestelmä.
- OIDC-pohjaiset kirjautumistavat, kuten Google ja yritysten omat toteutukset.

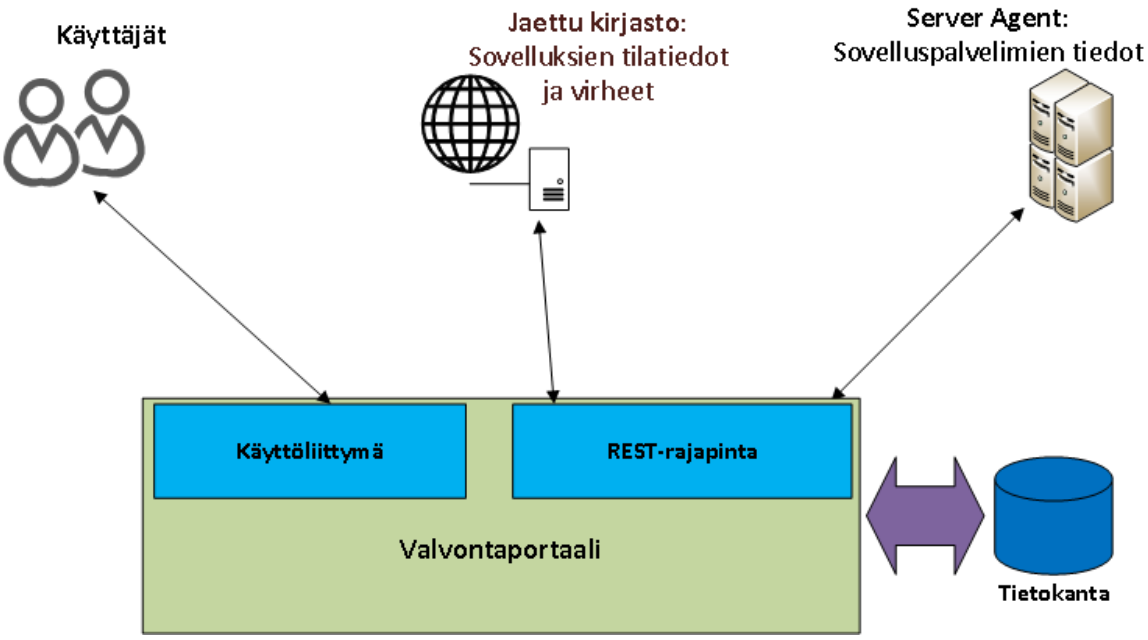
4 Määrittely

4.1 Tarveanalyysi

Yrityksen kanssa kartoitettiin valvonnan kannalta tärkeimmät ominaisuudet, jotka toteutettaisiin ensimmäisessä vaiheessa. Lisäksi päätettiin rajata sovellustason valvonta ensimmäisessä vaiheessa vain web-tuotteisiin, eli yksityisen ja julkisen puolen Mepco-HRM-tuotteiden web-sovelluksiin, Kirjaamoon sekä autentikointiin.

Ominaisuuksia kartoittaessa ilmeni, että tarvitaan kolme eri komponenttia (kuva 3):

- Valvontaportaali – Sovellus, johon tilat raportoidaan rajapintojen kautta ja josta niitä voi selata selaimella.
- Server Agent – Sovelluspalvelimille asennettava palvelinsovellus, joka raportoi palvelimen tilaa ja asetuksia valvontaportaalille.
- Jaettu kirjasto – Mepco-sovelluksiin lisättävä jaettu kirjasto, joka osaa raportoida sovelluksen tilan määrätyin väliajoin ja ilmoittaa mahdollisista virhetilanteista valvontaportaalille.



Kuva 3. Valvontaportaalin rakennetta havainnollistava kuva. Käyttäjät käyttävät sovellusta selaimen avulla, jonka käyttöliittymä kutsuu valvontaportaalin rajapintoja. Lisäksi sovellus- ja palvelinkohtaiset tiedot raportoidaan valvontaportaalin tarjoamaan rajapintaan.

Lisäksi mietittiin sovelluksen ei-toiminnallisia ja laadullisia vaatimuksia (non-functional requirements), jotka pitää huomioida sovelluksen toteutuksessa. Valvontasovelluksen pitää olla tietoturvallinen, tavoitettavissa, integroitavissa ja suorituskykyinen. Integroitavuudella tarkoitetaan tässä tapauksessa muun muassa sitä, että valvontaportaali tarjoaa rajapinnan, jonka avulla esimerkiksi asennusautomaatio voi kysyä sovelluksen tilaa.

4.2 Sidosryhmien roolit

Valvontajärjestelmää käyttävät eri sidosryhmät:

- käyttötuki
- asentajat
- HOT-koodarit
- konsultit
- asennusautomaatio.

Käyttötuen vastuulla on ratkoa ja vastata asiakkaiden palvelupyyntöihin sekä selvittää tarvittavia lisätietoja raportoidusta ongelmasta HOT-koodaria varten.

Asentajat vastaavat uusien ympäristöjen asennuksista ja olemassa olevien ympäristöjen päivityksistä sekä sovelluksesta riippumattomien ongelmien selvittämisestä, esimerkiksi verkko- ja palvelinongelmat.

HOT-koodarit toimivat käyttötuen ja asentajien apuna ja selvittävät mahdollisista ohjelma- tai määrittelyvirheistä johtuvia palvelupyyntöjä, joiden ratkaisemiseksi tarvitsee tutkia ohjelmakoodia.

Konsultit ja konsulttien ratkaisutukitiimi auttavat asiakkaita Mepco-tuotteiden määrityksissä ja ratkaisevat mahdollisia ongelmia, joita asiakkailla ilmenee.

DevOps-asennusautomaatiolle valvontaportaali tarjoaa keskitetyn tavan varmistaa, että sovellukset vastaavat yhä kutsuihin ja niiden tila on kunnossa.

Valvontajärjestelmän tulee tarjota kaikille sidosryhmille ajantasaista tietoa asiakkaalle asennetuista ympäristöistä ja niiden versioista sekä yleisestä tilasta. Lisäksi tilahistoria helpottaa vanhojen ongelmien vianselvitystä, kun järjestelmästä näkee kukin hetken versiohistorian ja tietoa mahdollisista raportoiduista ongelmista ja hälytyksistä.

4.3 Toiminnalliset ja tekniset määritykset

Työn aluksi tutkittiin markkinoilla olevia valmiita monitorointiratkaisuja, mutta sopivaa ja riittävän monipuolisesti räätälöitävää vaihtoehtoa ei löydetty. Päätettiin luoda räätälöity valvontajärjestelmä, jossa voimme huomioida paremmin sovelluksien sisäiset toiminallisuudet ja rakenteet.

Tässä luvussa käyn läpi ominaisuuksia, jotka portaaliin toteutetaan.

4.3.1 Konfiguraatioiden validointi

Mepco-sovelluksia pyöritetään nykyisellään pääasiassa Windows-palvelimilla käyttäen HTTP-palvelimena Windowsiin sisäänrakennettua Internet Information Services (IIS) -web-palvelinohjelmistoa.

Valvontasovelluksen Server Agent -sovellus osaa käydä palvelinten ja sovelluksien eri asetukset läpi ja varmistaa niiden oikeellisuuden ja tietoturvallisuuden. Sovellus käy palvelimen ja sovelluksien asetukset läpi ja varmistaa, että tarvittavat kovennukset on suoritettu ja listaa mahdolliset puutteet asetuksissa.

Koventamisella tarkoitetaan käyttöjärjestelmän ja sovelluksien asetuksien läpikäyntiä, jossa turhat toiminnallisuudet poistetaan käytöstä ja mahdolliset turvattomat oletusarvot muutetaan turvallisiksi. Esimerkiksi käyttöjärjestelmän ja IIS:n osalta suositeltava kovennustapa on CIS-kovennusohjeiden (Center for Internet Security) noudattaminen soveltuvin osin (3). Myös OWASP-yhteisöllä on olemassa IIS:lle kovennusohje (4), josta on CIS-ohjeistuksen lisäksi otettu vaikutteita kovennettaviin asetuksiin.

Palvelimen ja IIS:n konfiguraatiosta varmistetaan muun muassa, että sovelluksen käyttämän sovellussarjan (Application pool) asetukset ovat tuotantokäyttöön sopivat. Lisäksi esimerkiksi varmistetaan, että vanhat TLS- ja SSL-salausprotokollaversiot on poistettu käytössä ja käytössä on vain uusimmat TLS-versiot eli TLS 1.2 ja 1.3, koska vanhemmat versiot eivät ole enää suojaustasoltaan riittävän turvallisia.

Sovelluksien konfiguraatioissa on eroja riippuen esimerkiksi käytetystä ohjelmistokehyksestä ja ympäristön tyypistä, joten osa toiminnoista on toteutettu sovelluskohtaisesti. Esimerkiksi .NET Framework -pohjaiset tuotteet vaativat erilaiset asetukset kuin .NET Core -pohjaiset tuotteet. Myös testi-, esituotanto ja tuotantoympäristöillä on eroja konfiguraatioissa, sillä testi- ja esituotantoympäristössä ei välttämättä ole vielä käytössä esimerkiksi HTTPS-varmennetta. Lisäksi esimerkiksi vähemmän käytössä olevan testiympäristön ei tarvitse olla valmiiksi käynnissä ja odottamassa käyttäjiä, vaan se voidaan käynnistää vasta, kun sitä käytetään ja näin säästetään palvelinresursseja.

Ympäristöjen konfiguraatiosta tarkistetaan siis muun muassa:

- Tuotantoympäristössä ei ole käytössä käyttöönottovaiheen vaillinaisia asetuksia, jotka johtuvat esimerkiksi HTTPS-varmenteen puuttumisesta.
- HTTPS-asennuksessa on käytössä kaikki HTTPS-asennuksen lisäkonfiguraatiot, joilla varmistetaan, että esim. evästeille asetetaan Secure-määritys ja autentikaation uudelleenohjaukset käyttävät HTTPS:ää läpi koko autentikaatioketjun.
- Sovellukselle on konfiguroitu riittävän pitkä istuntoevästeen elinikä. Esimerkiksi nostettu se 20 minuutin oletuksesta 2 tuntiin.

4.3.2 Valvonta ja hälytykset

Sovelluksien sisäiseen valvontatoimintoon toteutetaan toiminto, joka osaa monitoroida sovelluksen tilaa ja raportoida mahdollisia virheitä sekä varoituksia erilaisista tilanteista niin sovelluksen kuin palvelimenkin osalta.

Itse sovelluksen tilan osalta tarkkaillaan riippuvuuksien, eli sovelluskomponenttien, välisiä yhteyksiä. Virheitä ja varoituksia raportoidaan esimerkiksi myös siitä, jos sovelluksen oletuskonfiguraatio ei ole määriteltyjen suositusten mukainen tai sovelluksen tietokantarakenne ei ole ajan tasalla. Lisäksi valvotaan sovelluksen käyttämää ja eri sovelluskomponenttien välisiä HTTPS-varmenteiden vanhentumista, lokitietojen muodostumista ja tietokannan edellisen varmuuskopion ajankohtaa.

Palvelimen osalta valvontakomponentti osaa raportoida käyttöjärjestelmän ja ohjelmistokehyksen tietoturvapäivitysten tilan, vähäisen levytilan sekä korkean käyttöasteen muisti- ja prosessorikäytön osalta.

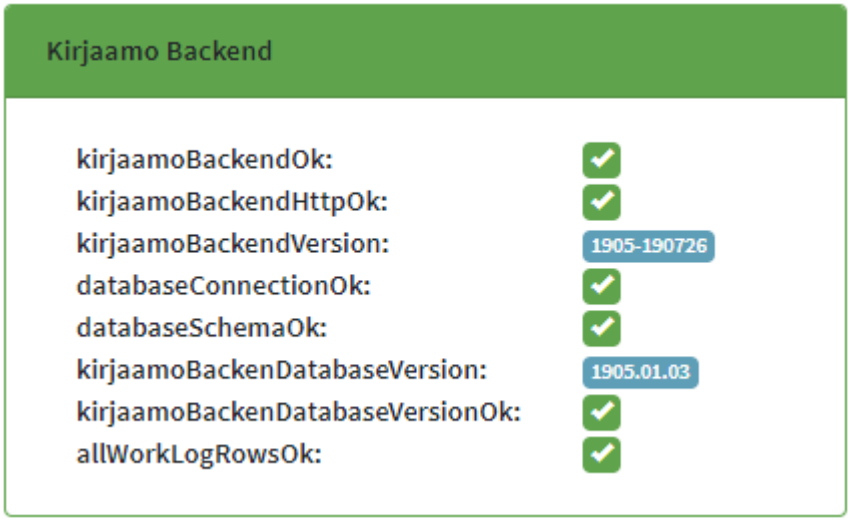
4.3.3 Tilasivujen laajennus ja visualisointi

Nykyisistä sovelluskohtaisista tilasivuista tarvitaan suojattu ja laajennettu versio, jossa on laajemmin tietoja palvelun ja palvelimen tilasta. Näitä laajempia tietoja ei voi näyttää julkisesti kaikille, koska osa tiedoista saattaisi helpottaa esimerkiksi hakkerointia ja hyökkäyksiä. Mepco-sovelluksiin toteutetaan toiminto, joka raportoi valvontajärjestelmälle sovelluksen tilan käynnistymisen ja virhetilanteiden yhteydessä.

Alkuperäiset sovellusten tilasivut olivat vain tilatietojen JSON-tuloste (kuva 4). JSON-tuloste on koneellisesti helposti luettavissa, mutta ihmissilmällä siitä on vaikea hahmottaa mahdollisia virheitä. Tästä syystä tilasivuille toteutetaan käyttöliittymä, joka osaa selkeästi korostaa mahdolliset ongelmakohdat värikoodein (kuva 5).


```
"kirjaamoBackendStatus": {
  "kirjaamoBackendOk": true,
  "kirjaamoBackendHttpOk": true,
  "kirjaamoBackendVersion": "1905-190726",
  "databaseConnectionOk": true,
  "databaseSchemaOk": true,
  "kirjaamoBackenDatabaseVersion": "1905.01.03",
  "kirjaamoBackenDatabaseVersionOk": true,
  "allWorkLogRowsOk": true
},
```


Kuva 4. Katkelma tilasivun JSON-tulosteesta.



Kuva 5. Tilasivun visuaalisen käyttöliittymän ensimmäinen versio.

4.4 Tietoturva

Valvontasovellus auttaa varmistamaan, että kunkin palvelimen ja sovellusasennuksen konfiguraatio noudattaa myös tietoturvamielessä hyväksi havaittuja asetuksia niin sovelluksien kuin palvelintenkin osalta. Konfiguraatioiden oikeellisuus on tärkeää, ja se on myös yksi OWASP TOP 10 2017 -listan kymmenestä kohdasta (1 s. A6:2017-Security Misconfiguration), joten siihen on syytä panostaa.

Sovellustasolla varmistetaan, että tuotantoympäristössä kaikki debug-asetukset, eli kehittäjä- ja virheenjäljitysasetukset, ovat poissa käytöstä. Näin varmistetaan, etteivät tarkat virheilmoitukset anna esimerkiksi mahdolliselle vihamieliselle käyttäjälle palvelimesta tai sovelluksen rakenteesta ylimääräistä tietoa, mikä lisäisi hyökkäyspinta-alaa.

Lisäksi varmistetaan, että sovellustasolla on kaikki nykyaikaiset tietoturvaa parantavat HTTP-otsakkeet (HTTP Header) käytössä. Tietoturvaa parantavia HTTP-otsakkeita ovat esimerkiksi (5):

- Content Security Policy (CSP). Otsakkeen avulla voidaan rajoittaa lähteitä, mistä selain saa ladata sivulle sisältöä.
- HTTP Strict Transport Security (HSTS), varmistaa, että selain kutsuu sivua aina HTTPS:n yli, vaikka käyttäjä yrittäisi kutsua sivua HTTP:n yli.

Palvelintasolla varmistetaan, että palvelinkomponentit ovat ajan tasalla ja esimerkiksi IIS:n asetuksia on kovennettu.

Valvontasovelluksen oman tietoturvan tarvitsee olla myös hyvällä tasolla, jotta se ei lisää hyökkäyspinta-alaa tai helpota tarjoamillaan tiedoilla hyökkäysten tekoa. Täten tarkempien tilatietojen haun rajapinta pitää suojata hyvin.

Tietoturvamielessä opinnäytetyön aikana tunnistettiin myös muita tarvittavia ominaisuuksia, mutta niitä ei käydä tässä työssä läpi, jotta ratkaisun tietoturva ei vaarannu.

4.5 Dokumentointi

Opinnäytetyön aikana valvontaportaalin käytöstä ja sovelluskohtaisen monitoroinnin käyttöönotosta luodaan alustavat ohjeistukset, jotka tallennetaan yrityksen Wikiin. Ohjeistuksia ja dokumentointia laajennetaan, kun portaalin käyttöä laajennetaan asteittain pilotoinnin aikana.

5 Käyttötapaukset

Sovellusta määritellessä ilmeni erilaisia käyttötapauksia, jotka kuvattiin toimintokortein. Toimintokortteihin määriteltiin jokaiselle toiminnolle lyhyt kuvaus, käyttäjäryhmät, esiehdot toiminnon tekemiselle, vaatimukset sekä mahdollinen tuotos. Esittelen tässä luvussa muutaman toiminnon käyttötapauskuvauksen.

Ensimmäinen käyttötapaus (taulukko 1) on uuden asennuksen lisäys järjestelmään. Sen tekee yleensä asentaja, mutta sen voivat tehdä myös muut rooliryhmät. Yhdellä asiakkaalla voi olla monta palvelinta tai yhdellä palvelimella monta asiakasta, joten asennukselle määritellään yksi tai useampi tunniste. Nämä asennukset ovat yleensä passiivisesti julkisen tilasivun kautta monitoroitavia, koska automaattisesti tilatietoja lähettävät asennukset lisätään automaattisesti.

Taulukko 1. Toimintokortti 1 – Uuden asennuksen lisääminen käsin syöttämällä

Toiminnon nimi	**Uuden asennuksen lisääminen käsin syöttämällä**
Lyhyt kuvaus	Käyttäjä lisää uuden asennuksen tiedot järjestelmään. Tietoja lisätessä määritellään asiakas, palvelin ja tietokanta.
Käyttäjäryhmät	Kaikki rooliryhmät
Esiehdot	- Käyttäjä on kirjautunut järjestelmään - Valvontaportaalilla on pääsy kohdejärjestelmään, eli se on joko julkisesti tavoitettavissa tai suojattu yhteys on muodostettavissa
Vaatimukset	- Käyttäjällä on tarvittavat tiedot asennuksen syöttämistä varten - Validointi sille, että asennus voidaan lisätä
Tuotos	- Uusi asennus on lisätty järjestelmään

Toinen käyttötapaus (taulukko 2) on uuden asennuksen lisäys valvontajärjestelmään automaattisesti. Se tapahtuu, kun asentaja tai muu sovellusta palvelimella konfiguroiva henkilö asettaa sovelluksen raportoimaan tilatietojaan automaattisesti. Sovellusasennuksen tiedot lisätään kantaan automaattisesti, kun sovellus raportoi tilatietojaan ensimmäistä kertaa, eikä sen perustietoja ole vielä tietokannassa.

Taulukko 2. Toimintokortti 2 – Uuden asennuksen lisääminen automaattisesti

Toiminnon nimi	**Uuden asennuksen lisääminen automaattisesti**
Lyhyt kuvaus	Mepco-sovellus on konfiguroitu raportoimaan tilaansa automaattisesti valvontaportaalille, eikä asennusta ole vielä määritelty valvontaportaaliin ja se lisätään automaattisesti.
Käyttäjäryhmät	Asentajat, jotka määrittelevät sovelluksen raportoimaan valvontaportaalille.
Esiehdot	- Mepco-sovelluksella on pääsy valvontaportaaliin, eli se on joko julkisesti tavoitettavissa tai suojattu yhteys on muodostettavissa.
Vaatimukset	- Validointi sille, että asennus voidaan lisätä - Tuki yhteyskäytännöille
Tuotos	- Uusi asennus on lisätty järjestelmään

Kolmas käyttötapaus (taulukko 3) on sovelluksen tilan raportointi proaktiivisesti, kun sovellus on konfiguroitu raportoimaan tilaansa valvontaportaalille automaattisesti.

Taulukko 3. Toimintokortti 3 – Sovelluksen tilan raportointi proaktiivisesti

Toiminnon nimi	**Sovelluksen tilan raportointi proaktiivisesti**
Lyhyt kuvaus	Mepco-sovellus on konfiguroitu raportoimaan tilaansa automaattisesti valvontaportaalille ja se lähettää tila- ja virhetietoja.
Käyttäjäryhmät	Sisäinen toiminto, jonka raportti on suunnattu käyttötukea ja asentajia varten.
Esiehdot	- Mepco-sovelluksella on pääsy valvontaportaaliin, eli se on joko julkisesti tavoitettavissa tai suojattu yhteys on muodostettavissa.
Vaatimukset	- Raportti lähetetään aina kun sovellus tila muuttuu tai tapahtuu virhe - Tilatiedon pitää tulla validissa JSON-muodossa - Tuki yhteyskäytännöille
Tuotos	- Uusi tilatieto tai virheraportti portaalissa

Asennuksen tilan tarkistaminen (taulukko 4) on mahdollisesti eniten käytetyin ominaisuus valvontajärjestelmässä. Sen voivat tehdä kaikki sidosryhmät. Käyttäjä näkee sovelluksen viimeisimmän tilan ja tilahistorian sekä mahdolliset virheet ja varoitukset.

Taulukko 4. Toimintokortti 4 – Asennuksen tilan tarkistaminen

Toiminnon nimi	**Asennuksen tilan tarkistaminen**
Lyhyt kuvaus	Käyttäjä tarkistaa asennuksen tilan, version ja ongelmahistorian
Käyttäjäryhmät	Kaikki rooliryhmät
Esiehdot	- Käyttäjä on kirjautunut järjestelmään - Käyttäjällä on oikeus asiakkaaseen
Vaatimukset	- Asennettu tuotteet, joiden valvontaa tuetaan - Tuotekohtaisesti sovellusversiot ja niiden yhteensopivuus - Tuotekohtaisesti sovelluksen tila ja mahdolliset varoitukset - Sovelluskonfiguraatio vastaa version vaatimuksia - Mahdollinen konfiguraatioautomaatio on onnistunut
Tuotos	- Raportti asennuksen tilasta käyttöliittymällä - Raportti asennuksen tilasta rajapinnan kautta - Raportti asennuksen virheistä (vain valvontaportaalissa)

Hälytys uudesta virhetilanteesta (taulukko 5) on toiminto, jolla asentajat saavat halutessaan ilmoituksen uusista virhetilanteista selaimen ilmoitustoiminnon kautta.

Taulukko 5. Toimintokortti 5 – Hälytys uudesta virhetilanteesta

Toiminnon nimi	**Hälytys uudesta virhetilanteesta**
Lyhyt kuvaus	Asentaja saa ilmoituksen uudesta virhetilanteesta, joka on raportoitu valvontajärjestelmään.
Käyttäjäryhmät	Asentajat
Esiehdot	- Käyttäjä on kirjautunut järjestelmään - Käyttäjä on tilannut ilmoitukset
Vaatimukset	- Asennetut tuotteet, joiden valvontaa tuetaan - Mahdollinen konfiguraatioautomaatio on onnistunut - Valvontayhteys toimii
Tuotos	- Asentaja saa ilmoituksen virhetilanteesta

Viimeinen esiteltävä käyttötapaus on DevOps-asennusautomaation suorittama tilan tarkistus rajapinnan kautta (taulukko 6). Toiminnon kautta asennusautomaatio voi tarkistaa sovelluksen tilan ja tulkita, onnistuiko asennuksen tai palvelinmuutoksen kaikki vaiheet.

Taulukko 6. Toimintokortti 6 – Asennusautomaatio tarkistaa tilan rajapinnan kautta

Toiminnon nimi	**Asennusautomaatio tarkistaa tilan rajapinnan kautta**
Lyhyt kuvaus	Asennusautomaatio tarkistaa tilan rajapinnan kautta, kun on suorittanut palvelimella sovellusasennuksia tai muita palvelinmuutoksia.
Käyttäjäryhmät	Asennusautomaatio ja mahdolliset muut automaattiset valvontajärjestelmät
Esiehdot	- Asennusautomaatiolla on pääsy valvontajärjestelmään
Vaatimukset	- Asennettu tuotteet, joiden valvontaa tuetaan - Valvontayhteys toimii
Tuotos	- Asennusautomaatio saa tiedon, onko asennuksen tila kunnossa

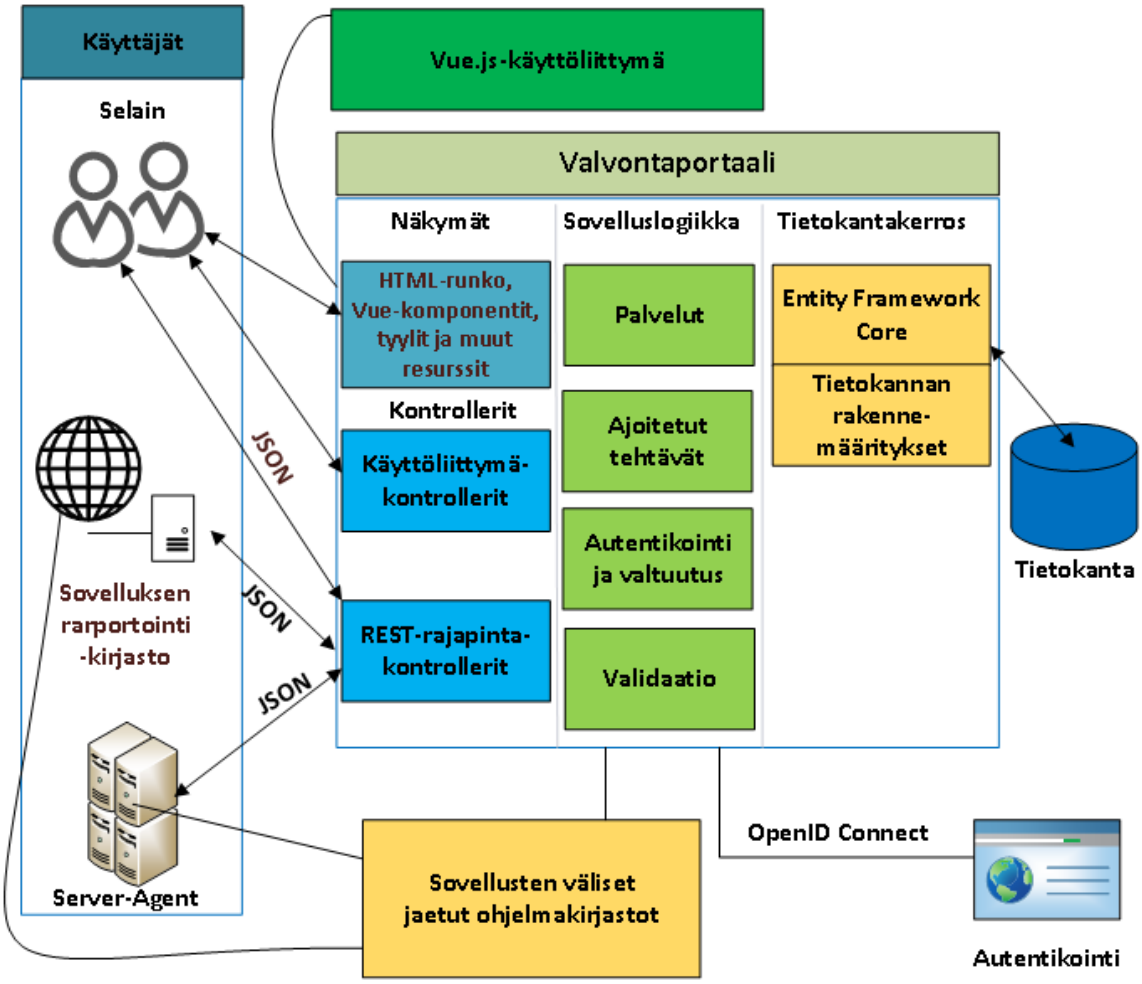
6 Sovelluksen rakenne ja toteutus

6.1 Valvontajärjestelmän rakenne

Valvontaohjelmisto koostuu eri sovelluskomponenteista:

- Valvontaportaali, johon tilat raportoidaan ja josta niitä voi selata.
- Server Agent, joka raportoi palvelimen tilaa valvontaportaalille.
- Sovelluksiin sisäänrakennetuista tilasivuista ja raportointitoiminnoista.

Sovelluksen arkkitehtuuri (kuva 6) perustuu rajapintoihin, joita hyödynnetään eri sovelluskomponenttien välisessä viestinnässä.



Kuva 6. Järjestelmän arkkitehtuurikuva.

Valvontaportaali koostuu Vue.js-kirjastolla toteutetusta selainkäyttöliittymästä sekä palvelintoteutuksesta, joka toteutettiin ASP.NET Core -ohjelmistokehyksen avulla. Valvontaportaali tarjoaa REST-mallin (Representational State Transfer) mukaiset API:t (Application Programming Interface), eli rajapinnat, sekä käyttöliittymälle toiminnoille että Server Agentille ja sovelluksien raportointitoiminnoille, jotta ne voivat raportoida tilatietojaan. Lisäksi valvontaportaali voidaan konfiguroida kutsumaan julkisia tilasivuja ajastetusti säännöllisin väliajoin tai manuaalisesti käyttöliittymän kautta.

Valvontaportaalin tietokantana käytettiin Microsoftin SQL Server -relaatiotietokantapalvelinta, joka tunnetaan myös lyhenteellä MSSQL. MSSQL oli työnantajan vaatimus tietokannaksi, koska sitä käytetään myös Mepco-tuoteperheen tuotteissa. Tietokantaa käytettiin Entity Framework Core -tietokantakirjaston avulla.

Server Agent -sovellus toteutettiin myös .NET Core -sovelluksena. Se asennetaan kaikille sovelluspalvelimille, jossa se kerää tiedot palvelimen tilasta ja tiedot asennetuista Mepco-sovelluksista ja niiden asetuksista sekä raportoi ne rajapinnan yli valvontaportaalille.

Sovelluskohtainen tilan raportointi toteutettiin omana kirjastonaan, joka kytketään kaikkiin valvottaviin sovelluksiin mukaan. Kirjasto raportoi tietyin väliajoin valvontaportaalille sovelluksen tilan sekä raportoi mahdollisista virhetilanteista sovelluksen suorituksen aikana. Kirjasto päivitetään sovelluspäivitysten yhteydessä**.**

6.2 Käytännön tietoturva

Valvontaportaaliin raportoidaan palvelimen tietoturvan kannalta kriittisiä tietoja, joten kaikki liikenne valvontaportaalin komponenttien välisissä REST-rajapinnoissa on salattua. Tiedot lähetetään salattuina ja allekirjoitettuina JWE-paketteina (JSON Web Encryption).

Valvontaportaaliin on toteutettu myös eri tasoisia käyttöoikeuksia, ja käyttäjän näkemät tiedot vaihtelevat oikeuksien mukaan. Esimerkiksi vain asentajat ja HOT-koodarit näkevät palvelimen virheistä tarkemmat tiedot.

Käyttäjähallinnassa, eli käyttäjien tunnistuksessa ja luvituksessa, hyödynnettiin ASP.NET:n omaa ASP.NET Identity -käyttäjähallintaa. Sen avulla määritettiin eri rooleja, joita käyttäjille voidaan valtuuttaa. Yhdellä käyttäjällä voi olla yksi tai useampi rooli ja roolien avulla voidaan rajata esimerkiksi oikeuksia eri asiakkaiden tietoihin eri lailla.

6.3 Kehitystyökalut

Sovelluksen toteutukseen käytettiin työnantajan tarjoamia Microsoftin kehitystyökaluja. Palvelinpuolen C#-ohjelmakoodin kehitykseen käytettiin Visual Studio 2019:ta. Selainpuolen TypeScript- ja JavaScript-koodia kehitettiin uudella avoimen lähdekoodin Visual Studio Codella, joka tarjoaa Visual Studiota monipuolisemmat työkalut moderniin Ja-

vaScript-kehitykseen. Versiohallintana käytettiin työnantajan käyttämää Azure DevOps:n Team Foundation Serveriä (TFS). Tietokannan käsittelyyn käytettiin Microsoft SQL Server Management Studion versiota 18.

6.4 Palvelinpään (backend) toteuttaminen

Tässä luvussa esitellään teknologiat, joita hyödynnettiin palvelinpään, eli ns. backend-pään toteutuksessa, ja niiden käyttöä varsinaisessa toteutuksessa.

6.4.1 .NET Core ja ASP.NET Core

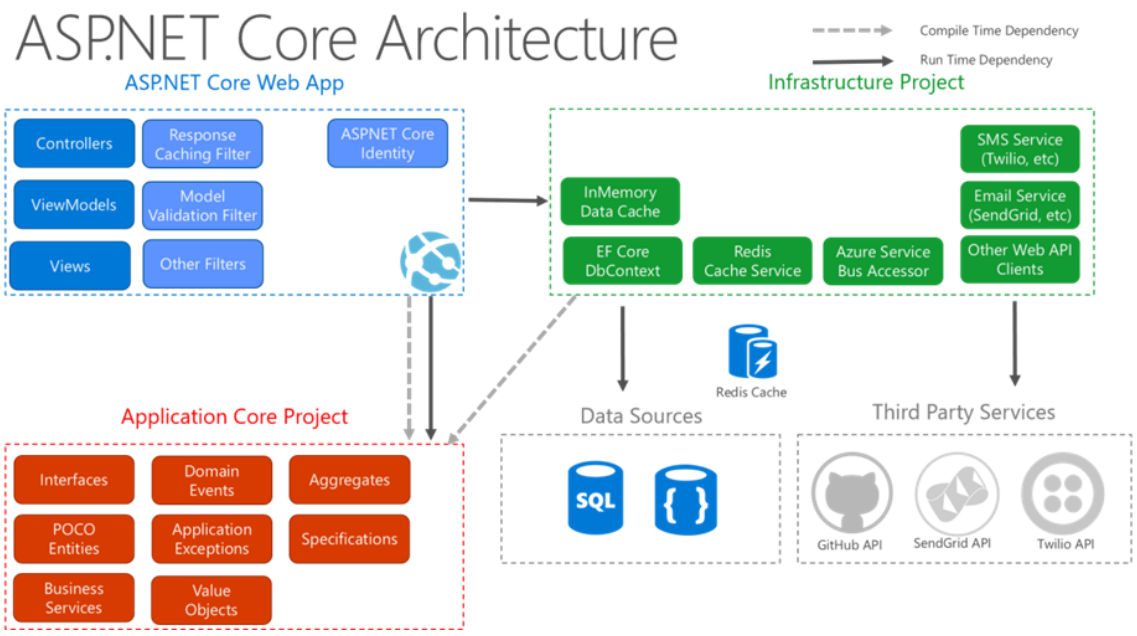
.NET Core on Microsoftin kehittämä alustariippumaton ja avoimen lähdekoodin vastine .NET Frameworkille. .NET Coren kehitys on Microsoftin vetämää, mutta avoimen lähdekoodin myötä sen kehitykseen voi osallistua kuka vain.

.NET Core ei sisällä vastineita kaikille .NET Frameworkin ominaisuuksille, vaan muun muassa vanhentuneita ns. legacy-ominaisuuksia ei ole toteutettu .NET Coreen. Aiemmin .NET Core ei tukenut myöskään Windowsin graafisia käyttöliittymäominaisuuksia (GUI), mutta versio 3.0 toi niille tuen, joka toimii vain Windowsin kanssa. Se mahdollistaa Windows Forms- ja Windows Presentation Foundation (WPF) -käyttöliittymäominaisuuksien käyttämisen ja siten Windows-työpöytäsovellusten kehittämisen. (6 s. 11.)

Web-kehityksen kannalta oleellisin ero .NET Coren ja .NET Frameworkin välillä on ASP.NET MVC ja ASP.NET Web API -rajapintojen puuttuminen .NET Coresta. Ne on yhdistetty ja refaktoroitu uudeksi ASP.NET Core -ohjelmistokehykseksi, joka on toteutettu .NET Coren päälle. (6 s. 11.)

ASP.NET Core on avoimen lähdekoodin versio Microsoftin kehittämästä ASP.NET-ohjelmistokehyksestä. Merkittävin ero ASP.NET Coressa on sen alustariippumattomuus. Perinteisiä ASP.NET-sovelluksia pystyy suorittamaan natiivisti vain Windowsilla, kun taas ASP.NET Corella kehitettyjä sovelluksia voi suorittaa Windowsin, Linuxin, Mac OS:n sekä Docker-konttien päällä. ASP.NET Coren kehityksessä on panostettu hyvään suorituskykyyn, ja se onkin yksi nopeimmista web-ohjelmistokehyksistä. (7.)

ASP.NET Core on optimoitu nykyaikaiseen web-ohjelmistokehitykseen. Se on rakenteeltaan modulaarinen (kuva 7), joten sovellus voi hyödyntää siitä vain tarvitsemansa ominaisuudet, mikä vähentää sovelluksen resurssitarpeita ja parantaa sovelluksen suorituskykyä sekä lisää tietoturvaa. (8 s. 3.)



Kuva 7. ASP.NET Core -sovelluskehyksen esimerkkiarkkitehtuuri. (8 s. 31)

ASP.NET Corelle on tarjolla paljon erilaisia kirjastoja, joiden avulla on helppo toteuttaa muun muassa autentikaatio, käyttöoikeuksien luvitus ja REST-rajapinnat suojauksineen (8 s. 59-63).

Valvontaportaalin palvelinpuoli toteutettiin hyödyntäen ASP.NET Core -ohjelmistokehystä. Sovelluksen kehitys aloitettiin käyttäen ASP.NET Coren versiota 2.0 ja päivitettiin työn edetessä aina uusimpaan ASP.NET Core -versioon, joka oli työn valmistumishetkellä 3.1.

ASP.NET Core valikoitui käytetyksi ohjelmistokehykseksi, koska yrityksessä on käytössä pääasiassa .NET-alustan ratkaisuja. ASP.NET Core on niistä web-ohjelmistokehityksen kannalta modernein ja monipuolisin vaihtoehto, jonka vuoksi se oli luonnollinen valinta tässä työssä käytettäväksi.

Tässä projektissa hyödynnetään laajalti .NET Coren valmiita kirjastoja ja ominaisuuksia, joita voi helposti laajentaa ja konfiguroida tarpeiden mukaan. Kirjastoja on käytetty esimerkiksi käyttäjien tunnistamisessa ja luvituksessa, kutsujen validoinnissa, rajapinnoissa sekä tietokannan käsittelyssä.

Seuraavaksi esittelen muutamien toiminnallisuuksien käyttöä tässä toteutuksessa. Ominaisuudet otetaan käyttöön ja konfiguroidaan yleensä .NET Core -projektin Startup-luokan ConfigureServices-metodissa.

Projektissa on käytössä .NET Coren Antiforgery-ominaisuus, jonka avulla voidaan torjua CSRF-hyökkäysyrityksiä (Cross-Site Request Forgery), eli estää kutsumasta sovelluksen käyttöliittymän rajapintoja toisen sivuston koodin kautta. Kuvassa 8 Antiforgery-ominaisuudelle konfiguroidaan käytettävä evästeen ja HTTP-otsakkeen asetukset.

```
services.AddAntiforgery(options =>
{
    options.Cookie.Name = $".{Settings.GeneralSettings.AppCookieBasename}.Antiforgery";
    options.Cookie.Path = Settings.GeneralSettings.AppUrlCookiePath;
    options.Cookie.SameSite = Settings.GeneralSettings.CookieSameSiteMode;
    options.Cookie.SecurePolicy = Settings.GeneralSettings.CookieSecurePolicy;
    options.HeaderName = "X-XSRF-TOKEN"; // Default is X-CSRF-TOKEN, note C -> X
    options.SuppressXFrameOptionsHeader = Settings.GeneralSettings.AntiforgerySuppressXFrameOptionsHeader;
});
```

Kuva 8. Antiforgery-ominaisuus konfiguroidaan projektin Startup-luokan ConfigureServices-metodissa. Esimerkissä ominaisuudelle konfiguroidaan käytettävä evästeen ja HTTP-otsakkeen asetukset.

Lisäksi Antiforgeryn varsinaisesta validoinnista huolehtiva suodatin pitää lisätä joko kontrollerin attribuutiksi tai globaalisti sovelluksen suodattimiin (Filters). Tässä projektissa käytettiin valmista AutoValidateAntiforgeryTokenAttributea, joka lisättiin sovelluksen MvcOptions-asetuksiin, joka näkyy myöhemmin kuvassa 10.

Lisäksi sovellukselle on räätälöity kustomoidut filtterit poikkeusten hallintaan ja muun muassa kirjautuneen käyttäjän hallintaan. Kuvassa 9 on esimerkkinä toteutus suodattimesta, joka asettaa kutsun kulttuurin käyttäjän kielen mukaan.

```
public class ResourceUICultureFilter : IAsyncResourceFilter
{
    1 reference
    public async Task OnResourceExecutionAsync(ResourceExecutingContext context, ResourceExecutionDelegate next)
    {
        var requestUserAccessor = context.HttpContext.RequestServices.GetRequiredService<RequestUserAccessor>();
        var user = await requestUserAccessor.GetUserFromSession(false);
        Thread.CurrentThread.CurrentUICulture = user?.GetLanguageCultureInfo() ?? new CultureInfo( name: "fi-FI");
        await next();
    }
}
```

Kuva 9. Kuvan suodattimessa haetaan kutsun käyttäjä (user) ja asetetaan UI-kulttuuri (CurrentUICulture) käyttäjän kielen mukaan.

Kuvassa 10 on esimerkki siitä, miten suodattimet konfiguroidaan käyttöön. Lisäksi kuvassa näkyy, kuinka sovelluksen tuottamille vastauksille voi määrittää omia erillisiä välimuistiprofiileja.

```
services // IServiceCollection
    .AddMvc( setupAction: options =>
    {
        //CacheProfiles: Käytetään harvemmin muuttuvissa tiedoissa tunnin välimuistiaikaa
        //Voidaan pakottaa nollaus vaihtamalla X-Cache-Identifier-headerin arvoa
        var listCacheSetting = 60.0; //TODO: Muuta konfiguroitavaksi
        var listCacheSeconds = Convert.ToInt32(TimeSpan.FromMinutes(listCacheSetting).TotalSeconds);
        options.CacheProfiles.Add("GetListCacheTime", new CacheProfile
        {
            Duration = listCacheSeconds,
            VaryByHeader = "X-Cache-Identifier"
        });
    })
    .SetCompatibilityVersion(CompatibilityVersion.Version_3_0) //Give me all of the 3.0 behavior
    .AddNewtonsoftJson()
    .AddMvcOptions(options =>
    {
        options.Filters.Add( item: new CustomExceptionFilterAttribute());
        options.Filters.Add( item: new SessionUserFilter());
        options.Filters.Add( item: new ResourceUICultureFilter());

        //Anti-Forgery (CSRF/XSRF)
        if (!Settings.GeneralSettings.DisableAntiForgerySecurityChecks)
        {
            options.Filters.Add( item: new AutoValidateAntiforgeryTokenAttribute());
        }
    });
```

Kuva 10. Kuvassa .NET Coren MVC-asetuksiin konfiguroidaan sovelluksen käyttämä välimuistiprofiili ja lisätään sovellukselle räätälöidyt suodattimet käyttöön. AddMvc-osuudessa luodaan ja konfiguroidaan räätälöity välimuistiprofiili (GetListCacheTime), jonka kestoksi (Duration) asetetaan 60 minuuttia ja kerrotaan VaryByHeader-asetuksella, että välimuisti on voimassa vain, mikäli uudenkin kutsun X-Cache-Identifier-HTTP-otsake on sama kuin vastausta välimuistittaessa. Näin välimuistin mitätöinti voidaan tarvittaessa pakottaa vaihtamalla kyseisen HTTP-otsakkeen arvoa. AddMvcOptions-kohdassa lisätään sovelluksen suodattimiin sovellukselle räätälöidyt suodattimet ja lisäksi aiemmin esitellyn Antiforgeryn validointisuodatin (AutoValidateAntiforgeryTokenAttribute).

Sovelluksen kontrollereissa ja palveluissa hyödynnetään .NET Coren sisäänrakennettua riippuvuusinjektiota (Dependency Injection), jonka avulla luodaan luokan tarvitsemista riippuvuuksista instanssit automaattisesti, kun luokka luodaan esimerkiksi kutsua varten.

```
public class UiController : BaseController
{
    private readonly ILogger<UiController> _logger;

    0 references
    public UiController(ILogger<UiController> logger, IServiceProvider serviceProvider) : base(serviceProvider)
    {
        _logger = logger;
    }

    [Route( template: "/")]
    [Route( template: "op/{vuejsQuery}")]
    [HttpGet]
    0 references
    public async Task<IActionResult> Index(string vuejsQuery)
    {
        _logger.LogDebug( message: "INDEX");
        var user = await GetRequestUser();
        return View(ViewModelFactory.GetViewModel<DefaultViewModel>(user, SetView));
    }
}
```

Kuva 11. Yksinkertainen käyttöliittymäkontrolleri, jonka avulla renderöidään pohja-HTML Vue.js-sovellusta varten. Kontrollerin rakentaja saa riippuvuutensa (logger, serviceProvider) riippuvuusinjektion kautta.

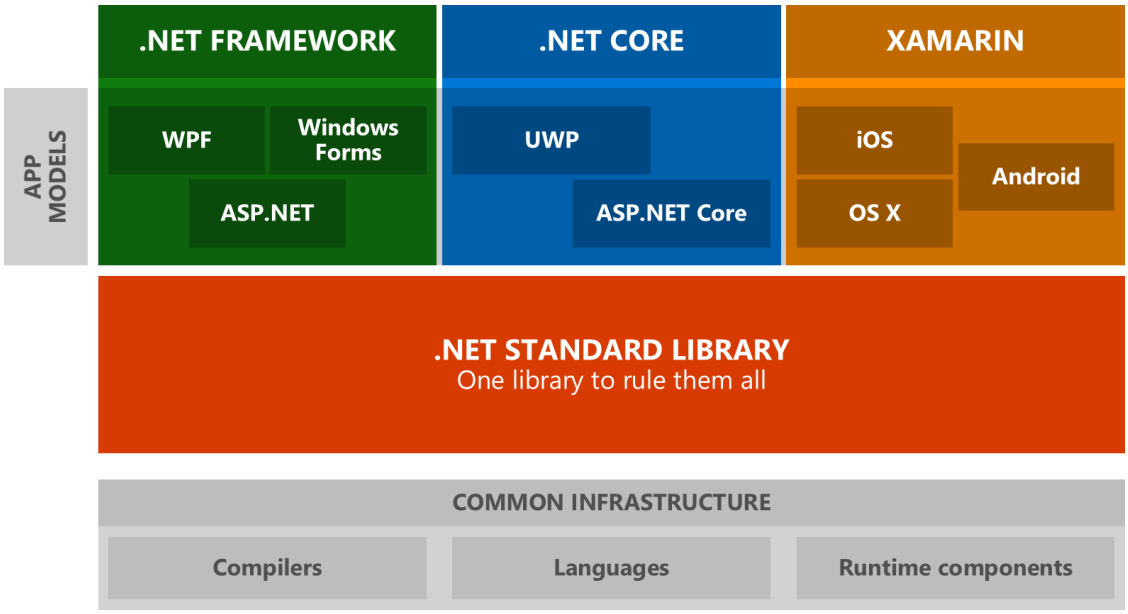
Koska sovelluksen käyttöliittymä on toteutettu SPA-toteutuksena Vue.js:llä, niin MVC-mallin mukaisen näkymän tehtäväksi jää palvelintoteutuksen osalta luoda vain perus-HTML-runko, johon Vue.js asettaa luomansa näkymän. Kuvassa 12 esitellään, kuinka Vue-sovelluksen instanssi käynnistetään.

```
@using Newtonsoft.Json;
@using SharedObjects.Configuration

@model MepcoProjectStatusTracker.ViewModels.ViewModelBase
@{
    var vueSettings = DevelopmentSettings.MepcoVueAppsSettings;
}

<div id="vue-ui-root">
    @await Html.PartialAsync( partialViewName: "_PartialHotModuleReplacementSwitch", Model)
</div>

@section scripts {
    <script>
        var VALIDATION_RULES = @Html.Raw(JsonConvert.SerializeObject(ValidationRules.RegexPatterns));
    </script>

    @*Käytetään HMR (Hot Module Replacement) versiota kehityksessä, jos käytössä.*@
    @if (Model.IsHotModuleReplacementEnabled && vueSettings.HmrEnabled)
    {
        <script src="@vueSettings.GetHotModuleReplacementUrlForEntryJS("chunk-common")" defer asp-append-version="true"></script>
        <script src="@vueSettings.GetHotModuleReplacementUrlForEntryJS("chunk-vendors")" defer asp-append-version="true"></script>
        <script src="@vueSettings.GetHotModuleReplacementUrlForEntryJS("uiApp")" defer asp-append-version="true"></script>
    }
    else
    {
        <link href="~/Scripts/vue-build/build/css/uiApp.css" asp-append-version="true" rel="stylesheet" />
        <script src="~/Scripts/vue-build/build/js/manifest.js" defer asp-append-version="true"></script>
        <script src="~/Scripts/vue-build/build/js/vendor.js" defer asp-append-version="true"></script>
        <script src="~/Scripts/vue-build/build/js/chunk-common.js" defer asp-append-version="true"></script>
        <script src="~/Scripts/vue-build/build/js/uiApp.js" defer asp-append-version="true"></script>
    }
}
```

Kuva 12. Kuvassa näkyy, kuinka Vue-ohjelma istutetaan .NET Coren näkymään. Ensin luodaan div-elementti id:llä ”vue-ui-root”. Lisäksi lisätään sivulle käyttöliittymäsovelluksen (uiApp) käännetty koodi ja sen tarvitsemat muut pakolliset riippuvuudet (manifest, vendor, chunk-common). Muut tarvittavat komponentit ladataan tarvittaessa erikseen.

6.4.2 .NET Standard

.NET Standard on spesifikaatio, joka kuvaa kaikkien eri .NET-alustojen tukemat rajapinnat. Siten .NET Standardia toteuttavaa ohjelmakirjastoa on mahdollista käyttää kaikilla eri .NET-alustojen välillä. .NET Standardin tarkoituksena on yhtenäistää .NET-alustoja ja estää siten pirstoutumista. (9.) Kuva 13 havainnollistaa .NET Standardin asemaa .NET-ohjelmistokehityksessä.



Kuva 13. .NET Standardiin pohjautuvaa kirjastoa voi käyttää kaikilla .NET-alustoilla.

Taulukossa 7 on havainnollistettu määriteltyjen rajapintojen lukumäärän kasvua eri .NET Standard -versioiden välillä. Uusin .NET Standard -versio on 2.1 ja se sisältää 37 118 yhteistä rajapintaa. Tämän työn kirjastoissa käytettiin .NET Standard -versiota 2.0, koska se on uusin .NET Standard, jota myös .NET Framework tukee.

Taulukko 7. Eri .NET Standard -versioiden määrittelemien rajapintojen lukumäärä (10).

Versio	Rajapintojen lukumäärä	Kasvu-%
1.0	7 949	
1.1	10 239	29 %
1.2	10 285	0 %
1.3	13 122	28 %
1.4	13 140	0 %
1.5	13 355	2 %

1.6	13 501	1 %
2.0	32 638	142 %
2.1	37 188	14 %

Tässä työssä .NET Standardia hyödynnettiin jaetuissa kirjastoissa, koska osa Mepco-tuotteista käyttää vielä .NET Frameworkia. Jaetun .NET Standard -kirjaston avulla samaa ohjelmakirjastoa voitiin hyödyntää molemmissa alustoissa saumattomasti.

6.4.3 Entity Framework Core

Sovelluksen tietokantakäsittely toteutettiin käyttäen Microsoftin kehittämää Entity Framework Corea (EF Core), joka tarjoaa ohjelmointirajapinnan relaatiotietokannan käsittelyyn. EF Core on alustariippumaton avoimen lähdekoodin versio Entity Frameworkista (11). Se mahdollistaa .NET-oliomallien käyttämisen tietokantaa vasten, eli tarjoaa ORM-rajapinnan (Object-relational mapping) tietokannan ja oliomallin välille.

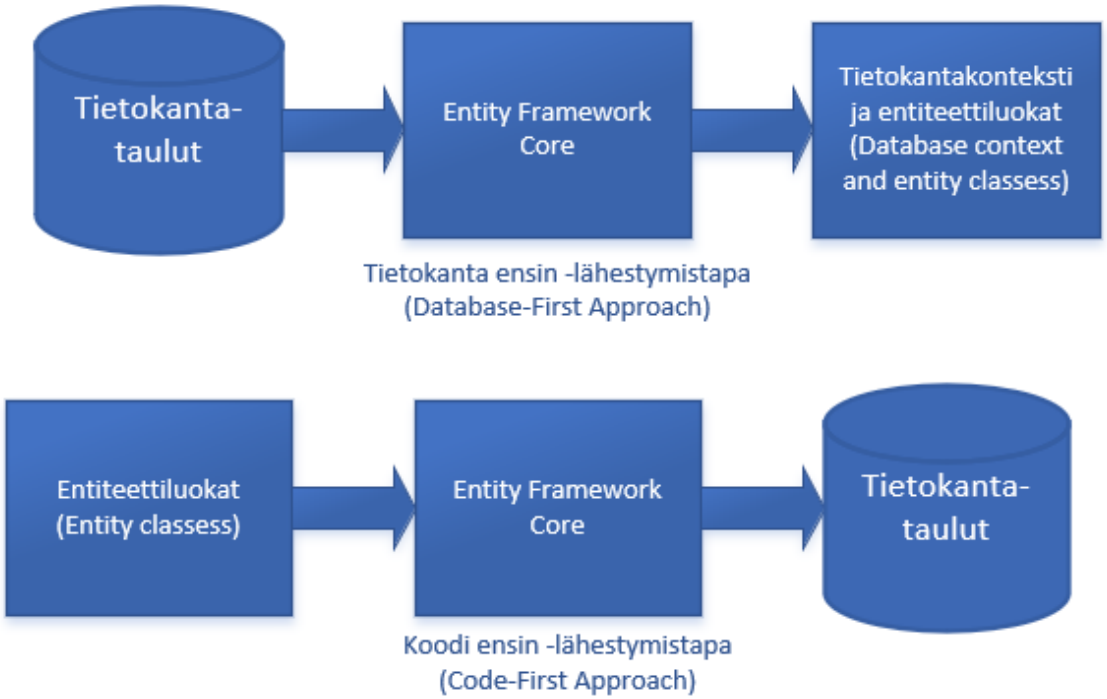
EF Coressa tietokantakyselyitä luodaan hyödyntäen .NET-alustojen tukemia Language-Integrated Query -kyselyjä (LINQ), joista EF Core muodostaa tarvittavat SQL-kyselyt. Kuvassa 14 on havainnollistava esimerkki LINQ:n käytöstä EF:n kanssa.

```
var statusList = await DbContext.InstallationStatuses
    .Include(s => s.Installation)
    .Include(s => s.Installation.Customer)
    .Include(s => s.Installation.Server)
    .OrderByDescending(s => s.Modified ?? s.Created)
    .Take(30)
    .ToListAsync();
```

Kuva 14. Esimerkki EF Coren kyselyn luomisesta LINQ:n avulla. Kyselyssä haetaan viimeisimpiä tilamuutoksia, kyselyyn otetaan mukaan tilataulun relaatioita Include-komennolla, järjestetään ne aikaleiman mukaan laskevaan järjestykseen ja otetaan 30 uusinta tilamuutosta. EF Core muodostaa tästä tietokantakyselyn automaattisesti.

EF Core tukee kahta lähestymistapaa kehityksessä (kuva 15) (12):

- Code-First – tietokanta luodaan oliomallien perusteella
- Database-First – oliomallit luodaan tietokannan perusteella.



Kuva 15. Havainnekuva Database-First- ja Code-First-lähestymistavoista (12).

EF Coren suositeltu tapa on käyttää Code-First-lähestymistapaa, jossa ohjelmoijan ei tarvitse luoda tietokantatauluja, vaan EF Core muodostaa tarvittavat taulut automaattisesti luotujen oliomallien pohjalta. Tietokantarakenteen muuttuessa kehittäjä generoi EF Coren työkalujen avulla migraatiotiedoston, jonka avulla tietokanta päivitetään ohjelmallisesti, kun sovellusta päivitetään.

Tässä työssä EF Core helpotti tietokantarakenteen muodostamista ja kyselyiden tekoa. Haasteita se taas toi migraatioiden ja suorituskyvyn myötä. Migraatioissa pitää olla tarkkana, että ne tehdään oikein, jotta olemassa olevaa tietoa ei esimerkiksi menetetä. Suorituskykyä on myös tarkkailtava, koska LINQ:n avulla on helppo tehdä liian monimutkaisia ehtoja, joista EF Core ei pysty luomaan riittävän tehokkaita tietokantakyselyitä. Pahimmillaan EF Core suorittaa ehtorajaukset kokonaan ohjelmakoodissa, jolloin saatetaan ladata koko tietokantataulun sisältö sovelluksen muistiin ja tehdä tietueiden rajaus vasta siellä. EF Coren versiossa 3.0 tosin parannettiin sekä EF Coren LINQ-muunnoksien tukea, että estettiin sellaisien ehtojen suorittaminen, joiden rajaus tapahtuisi vasta ohjelmakoodissa.

Työssä tietokantamigraatiot luotiin muokkaamalla tietokantakontekstin entiteettiluokkia sekä määrittämällä pääluokkaa. Pääluokassa (kuva 16) voidaan määritellä, mitkä entiteettiluokat tietokantarakenteeseen kuuluvat. Pääluokassa voi määrittää entiteettiluokkien välille myös relaatioita ja muita monimutkaisempia riippuvuuksia kuin mitä suoraan entiteettiluokista voi määrittää. Tällainen määritys on esimerkiksi se, että mitä tapahtuu, kun vaikka palvelin poistetaan, mutta sillä on vielä asennuksia olemassa; poistetaanko myös asennukset, merkataanko asennuksien palvelin vain tyhjäksi vai estetäänkö poisto. Pääluokassa voi myös uudelleen määrittää EF:n perustietokantakontekstin perustoimintoja, ja räätälöidä ne sovelluksen käyttöön. Tässä projektissa muokattiin tallennusmetodeja (SaveChanges, SaveChangesAsync) huolehtimaan, että tiedot rivin luojasta, luontiajasta, muokkaajasta ja muokkausajasta tallentuvat kantaan automaattisesti oikein.

```
public class StatusDatabaseContext : DbContext
{
    public readonly ITenantProvider TenantProvider;
    public DbSet<MpstInstallation> Installations { get; set; } = null!;
    public DbSet<MpstServer> Servers { get; set; } = null!;

    public StatusDatabaseContext(
        DbContextOptions<StatusDatabaseContext> options,
        ITenantProvider tenantProvider
    ) : base(options)
    {…
    }
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {…
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        new TenantQueryFilterConfigurator<MpstInstallation>(this).Configure(modelBuilder.Entity<MpstInstallation>());
        new TenantQueryFilterConfigurator<MpstServer>(this).Configure(modelBuilder.Entity<MpstServer>());

        // Default valuet, Created:
        modelBuilder.Entity<MpstInstallation>().Property(b => b.Created).HasDefaultValueSql("getdate()");
        modelBuilder.Entity<MpstServer>().Property(b => b.Created).HasDefaultValueSql("getdate()");

        modelBuilder.Entity<MpstServer>()
            .HasMany(c => c.Installations)
            .WithOne(e => e.Server)
            // Estetään serverin poisto, jos sillä on asennuksia
            .OnDelete(DeleteBehavior.Restrict);
    }
```

Kuva 16. Kuvassa on yksinkertaistettu versio sovelluksen tietokantakontekstista. Kontekstille on määritetty, että kaksi entiteettioliota (Installation, Server) kuuluvat siihen ja OnModelCreating-metodissa määritellään olioiden välisiä riippuvuuksia ja muita asetuksia.

Kuvassa 17 on esimerkki tietokantaoliosta. Tietokantaluokassa määritellään mahdollinen avainkenttä ja pakolliset kentät. Lisäksi luokassa voidaan määritellä esimerkiksi myös mahdolliset relaatiot muihin tauluihin, mutta tässä esimerkissä niitä ei ole.

```
public class MpstErrorStacktrace : MpstEntityBase
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int Id { get; set; }

    /// <summary>
    /// SHA256 HASH of Stacktrace
    /// </summary>
    [Required]
    [Column(TypeName = "varchar(64)")]
    public string StacktraceHash { get; set; } = null!;

    [Required]
    [Column(TypeName = "varchar(max)")]
    public string Stacktrace { get; set; } = null!;

    public string? ExceptionFullClassName { get; set; }
}
```

Kuva 17. Esimerkki yksinkertaisesta entiteettiluokasta, jolle on määritetty avainkenttä ja kolme muuta kenttää. Entiteettiluokka perii MpstEntityBase-luokasta itseensä myös yhteiset kentät. Tässä projektissa yhteisiä kenttiä ovat lisääjä- ja muokkaajatiedot ajankohtineen.

Kun tietokantakontekstia tai sen olioita on muutettu, pitää generoida migraatiotiedostot muutoksista (kuva 18).

```
REM Muista päivittää versio migraation luonnin jälkeen.
dotnet tool restore
dotnet ef migrations add Mpst-2020.01.03 --context SharedObjects.Models.Database.StatusDatabaseContext -o DatabaseMigrations
pause
```

Kuva 18. Migraatiotiedostot luodaan dotnet-ef-kehitystyökalun avulla.

Syntyneiden migraatiotiedostojen avulla tietokanta päivitetään yksinkertaisesti kutsumalla tietokantakontekstin Migrate- tai MigrateAsync-komentoa (kuva 19), jolloin EF suorittaa migraation mukaiset päivityskyselyt kohdetietokantaan.

```
public async Task<JsonResult> UpgradeDatebase()
{
    try
    {
        await DbContext.Database.MigrateAsync();

        if (DbContext.Products.Count() < 6)...

        return Json( data: true);
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, message: "UpgradeDatebase failed!");
        return Json( data: false);
    }
}
```

Kuva 19. Tietokannan päivitys tapahtuu kutsumalla tietokantakontekstin Migrate- tai MigrateAsync-komentoa.

Entity Framework helpotti tietokantarakenteen luomista, kun luokkien rakenteen pystyi määrittelemään ensin ja luomaan sitten sen pohjalta varsinaisen tietokantarakenteen automaattisesti migraatioiden avulla. Hyviä puolia on myös se, että muutokset pitää tehdä vain yhteen paikkaan, eikä tarvitse huolehtia tietokannan rakenteen päivityskyselyiden luonnista. Huonoja puolia on se, että muutosten migraatioiden kanssa pitää olla todella tarkkana, jotta ne eivät aiheuta tietojen menetyksiä. Migraatioiden generointityökalu tosin kyllä varoittaa tilanteista, joissa dataa saattaisi kadota (kuva 20), joten ohjelmoijan on helppo havaita nämä tilanteet ja korjata ne ennen kuin on liian myöhäistä.

```
C:\TeamProjects\HRM\Tools\MepcoProjectStatusTracker\MepcoProjectStatusTracker>dotnet ef migrations add Mpst
-2020.01.05 --context SharedObjects.Models.Database.StatusDatabaseContext -o DatabaseMigrations
Build started...
Build succeeded.
PGR: MPSTSettings updated! DatabaseSettings: OK
An operation was scaffolded that may result in the loss of data. Please review the migration for accuracy.
Done. To undo this action, use 'ef migrations remove'
```

Kuva 20. Kuvassa näkyy EF:n antama varoitus migraatiosta, jonka seurauksena saattaa kadota olemassa olevia tietoja.

6.5 Asiakaspään (frontend) toteuttaminen

Valvontaportaalin käyttöliittymä on toteutettu käyttäen Vue.js-nimistä JavaScript-kirjastoa. Ohjelmointikielenä on käytetty TypeScript-versiota 3.7 ja muita nykyaikaisia web-kehitystekniikoita. Tässä luvussa esitellään näitä tekniikoita sekä niiden käyttöä.

6.5.1 Vue.js

Vue.js, joka tunnetaan myös lyhyemmällä nimellä Vue, on alun perin Evan Youn luoma JavaScript-kirjasto, jonka avulla voidaan toteuttaa sovelluksien näkymäkerros. Vue on kevyt ja suorituskykyinen, sillä sen ydin on hyvin pienikokoinen. Pienestä koostaan huolimatta Vue on hyvin joustava, sillä sitä voidaan tarvittaessa laajentaa erinäisin lisäkirjastoin.

Vuea voi käyttää niin osana jo olemassa olevaa web-sovelluksen käyttöliittymää kuin myös toteuttaa sillä koko web-sovelluksen käyttöliittymän esimerkiksi SPA-toteutuksena. Vue osaa päivittää näkymää automaattisesti oliomallin muuttuessa, mutta tekee sen hyvin suorituskykyisesti päivittämällä vain sen osan sivusta, joka tarvitsee päivittää. (13 s. 3, 12.).

Vue-tiimi on kehittänyt tarjolle viralliset lisäkirjastot nimeltä Vue Router ja Vuex, joiden avulla voidaan helposti toteuttaa moderneja niin sanottuja yhden sivun käyttöliittymiä, eli SPA-käyttöliittymiä (Single-page application). (13 s. 3, 101)

Vue Router mahdollistaa näkymäkomponenttien näyttämisen ja vaihtamisen osoitteen perusteella, mikä helpottaa selaimen historiatoimintojen hyödyntämistä näkymien välillä, vaikka sovellus onkin toteutettu SPA-periaatteella. Tällöin käyttäjä voi siis käyttää selaimen eteen- ja taaksepäin-näppäimiä normaalisti. Vastaavasti käyttäjä voi asettaa tietyn osoitteen selaimen kirjanmerkkeihin ja pääsee kirjamerkin kautta näkymään, jossa se oli, kun se luotiin.

Vuex puolestaan tarkoitettu keskitettyyn tilanhallintaan, eli sen avulla kaikki komponentit voivat tarvittaessa päästä kiinni keskitettyyn tilaan, eli lukea sen tietoja, muokata niitä, ja tilan muuttuessa kaikki komponentit saavat tiedon muutoksista automaattisesti.

Vue-näkymiä muodostetaan luomalla Vue-juurikomponentti, joka asetetaan korvaamaan jokin tietty HTML-koodin (Hypertext Markup Language) elementti (kuva 21). Juurikomponentti ja sen alaiset muut komponentit voivat sisältää muita Vue-komponentteja. Vue-komponentteja voi luoda joko itse tai käyttää muiden kehittämiä komponentteja.

```
<div id="vue-ui-root">
</div>


import router from '@/UI/vue-router/index'

...

// eslint-disable-next-line no-new
new Vue({
  el: '#vue-ui-root',
  router,
  // store,
  render: (h): VNode => h(RouterRootView)
})
```

Kuva 21. Koodikatkelma siitä, miten Vue-näkymän juurikomponentti asetetaan korvaamaan olemassa oleva HTML-elementti sivulta. Korvattava elementti kerrotaan antamalla Vue-juurikomponentin asetuksissa el-propertyn arvona elementin id. Kuvassa näkyy myös, miten erikseen määritetty Vue Router -määritys kerrotaan Vuen juurikomponentille antamalla konfiguraatioon router-olio.

Vue-komponentti koostuu kahdesta osasta, esityskerroksesta ja käyttäytymiskerroksesta (kuva 22). Esityskerroksessa kuvataan HTML-mallin ja tarvittaessa CSS-tyylimäärittelyn (Cascading Style Sheets) avulla, miltä komponentti näyttää ja missä kohtaa komponenttia mikäkin tieto näytetään. Käyttäytymiskerroksessa on komponentin JavaScript- tai TypeScript-ohjelmakoodi, jossa kuvataan objektissa komponentin tila ja määritellään metodit, joilla komponentin tilaa voidaan muokata. (13 s. 12.)

Jokaiselle Vue-komponentille voidaan määrittää omat muuttujat dataolioon, joiden arvoja komponentti muuttaa ja joita komponentti voi jakaa myös eteenpäin omille alikomponenteilleen. Komponentille voidaan määrittää myös, mitä props-arvoja se voi ottaa vastaan ylemmältä ns. parent-komponentilta. Dataolion ja props-arvojen pohjalta voidaan laskea komponentin sisäisiä ns. computed-metodeita, joiden laskema arvo lasketaan vain kerran ja asetetaan komponentin välimuistiin. Vue osaa valvoa Vue-komponenttien reaktiivisten olioiden osalta muutoksia automaattisesti, joten välimuistettua arvoa käytetään, kunnes jonkin computed-metodin käyttämistä reaktiivista lähdearvoista muuttuu, jolloin computed-arvo lasketaan automaattisesti uudelleen.

```
<template>
  <span class="badge" :class="`badge-${rowStatusClass}`">
    {{ rowStatusText }}
  </span>
</template>

<script lang="ts">
import { Component, Prop, /* Watch, */ Vue } from 'vue-property-decorator'
import { getTranslation2 } from '@/utils/MepcoFunctionCollection'
@Component({
})
export default class MvgCustomFormatterRowActionButtons extends Vue {
  /**
   * Props:
   */
  @Prop({ type: Object, required: true })
  private readonly rowItem!: MvgRowItem

  @Prop({ type: [String, Number], required: true })
  private readonly headerId!: string

  /**
   * Computed:
   */
  get rowStatusClass (): string {
    return this.rowItem?.[this.headerId] ?? 'light'
  }

  get rowStatusText (): string {
    return getTranslation2(`statusLabel.${this.rowStatusClass}`)
  }
}
</script>
```

Kuva 22. Esimerkki yksinkertaisesta Vue-komponentista, joka on toteutettu TypeScriptillä. Komponentissa on esityskerros, eli HTML-osuus sekä käyttäytymiskerros, eli TypeScript-ohjelmakoodi. Komponenttia käytetään muotoilemaan taulukossa tilarivin tila.

Vue-komponentin HTML-mallista voidaan viitata suoraan komponentin ohjelmakoodissa määritettyihin muuttujiin, props-arvoihin, metodeihin ja computed-arvoihin. Lisäksi HTML-osassa voidaan lisätä eri HTML-elementeille tapahtumakuuntelijoita, joiden avulla voidaan suorittaa tietty metodi, kun elementtiä esimerkiksi klikataan hiirellä.

Vue-komponentteja voi luoda joko itse tai sitten voidaan ottaa käyttöön muiden kehittämiä valmiina Vue-komponentteja ja -kirjastoja. Tässä projektissa käytettiin mm. Vue Routeria, Bootstrap-Vue-nimistä komponenttikirjastoa, jonka avulla on helppo toteuttaa Bootstrap 4 -käyttöliittymäkirjastoa hyödyntävä Vue-sovellus. Komponentit lokalisoitiin hyödyntäen käyttäen Vue i18n -nimistä lokalisointikirjastoa.

Vue Routerin käyttö on melko helppoa. Näkymään voidaan määrittää joko yksi oletus-router-view-kohta tai sitten useampi nimetty router-view, jolloin esimerkiksi eri näkymien valikkonäkymä voi olla erilainen routen määritysten mukaan. Tässä projektissa käytettiin vain yhtä route-view'tä kuvan 23 mukaisesti.

```
<template>
  <div id="vue-router-content-root">
    <router-view />
    <div class="modal-container">···
    </div>
  </div>
</template>
```

Kuva 23. Esimerkki Vue Routerin router-view-komponentin käytöstä Vue-komponentin näkymän sisällä.

Vue Routeria konfiguroidaan määrittelemällä, mitä polkuja on olemassa ja mitä komponentteja mikäkin näkymä käyttää. Lisäksi Vue Routerin konfiguraatiossa voidaan esimerkiksi määrittää, mitä props-arvoja näkymän Vue-komponentille välitetään ja mistä ne ovat peräisin. Kuvassa 24 on esimerkkinä pari routea, joista toisen routen (/op/CustomerDetails) props-arvot parsitaan selaimen osoitteesta ja toisessa niitä ei ole ollenkaan. Props-arvot välitetään routen mukaiselle Vue-komponentin props-arvoiksi.

```
import Vue, { VNode } from 'vue'
import Router, { Route } from 'vue-router'
// ...
/* Components, async Lazy-Loading */
type AsyncVueComponent = Promise<typeof import('*.vue')>
const Dashboard = (): AsyncVueComponent => import(
  /* webpackChunkName: "chunk-Dashboard" */
  '@/UI/Dashboard.vue'
)
const CustomerDetails = (): AsyncVueComponent => import(
  /* webpackChunkName: "chunk-Customer" */
  '@/UI/Customer/CustomerDetails.vue'
)
// ...
type RoutePropTypes = NumberConstructor | StringConstructor
const routeParams: {[key: string]: {[key: string]: RoutePropTypes|RoutePropTypes[] }} = {
  'customerDetails': { 'id': Number },
  // ...
}
// ...
Vue.use(Router)
/* Vue-router config */
export default new Router({
  mode: 'history',
  base: APP_BASE_URL, // BASE URL Globaalista muuttujasta
  routes: [
    { path: '/', name: 'Dashboard', component: Dashboard },
    {
      path: '/op/CustomerDetails',
      name: 'CustomerDetails',
      component: CustomerDetails,
      props: (route): RoutePropsType => parsePropsFromUrl(route, routeParams['customerDetails'])
    },
    // ...
  ]
})
```

Kuva 24. Esimerkki Vue Routerin konfiguroinnista ja route-määritysten tekemisestä. Kuvassa määritellään ensin, mistä käytettävät Vue-komponentit löytyvät. Koska kaikkia komponentteja ei tarvita kerralla, ne ladataan asynkronisesti tarpeen mukaan, eli sitten kun kyseiseen näkymään siirrytään. Seuraavaksi määritellään routen parametrit ja tyypit,

jotka välitetään props-arvoina routen Vue-komponentille. Sitten otetaan Vue Router käyttöön (Vue.use(Router)) ja luodaan uusi instanssi Vue Routerista (new Router()), jolle annetaan parametrina asetukset, eli muun muassa route-määritykset.

6.5.2 ECMAScript, TypeScript, Babel, Vue-cli ja Webpack

Projektin käyttöliittymäkoodi kehitettiin käyttäen viimeisimpiä JavaScript-ominaisuuksia ja -tekniikoita. Uusien ominaisuuksien ja tekniikoiden, kuten esimerkiksi ECMAScript 2019 (ES10) ja TypeScript, avulla koodin laatua ja selkeyttä voidaan parantaa.

ECMAScript on JavaScriptin spesifikaatio, jossa määritellään JavaScriptin eri versioiden ominaisuudet. JavaScriptin suosio on kasvanut viime vuosina paljon, ja se äänestettiin Stack Overflow'n vuoden 2019 kehittäjäkyselyssä seitsemättä vuotta peräkkäin suosituimmaksi ohjelmointikieleksi (14). JavaScriptiä on alettu käyttämään myös selaimien ulkopuolella muun muassa Node.js:n avulla. Node.js:n suosion myötä sitä on alettu käyttää paljon myös palvelinpuolen koodin kehittämiseen. Suosion kasvu on näkynyt myös kielen kehittymisenä. ECMAScriptistä onkin tullut viime vuosina lukuisia uusia versioita, joissa JavaScriptiin on lisätty uusia kehittämistä helpottavia ominaisuuksia.

ECMAScript-standardin ensimmäiset versiot julkaistiin vuosina 1997–1999, jonka jälkeen seuraava versio, viides versio (ES5), julkaistiin vuonna 2009. Tässä versiossa tuli tärkeitä ominaisuuksia, muun muassa JSON-tuki, strict mode ja laajempi tuki taulukoille. Seuraava versio, ES6, julkaistiin vuonna 2015, jonka jälkeen standardista on julkaistu vuosittain uusi versio, jotka ovat tuoneet kieleen uusia ominaisuuksia.

Tärkeimpiä uusia ominaisuuksia ovat (15):

- lohkokohtaiset muuttujat, const ja let
- nuolifunktiot, x => x.id
- moduulit
- promises
- luokat ja periytyminen
- tuki parametrien oletusarvoille
- Uudet taulukoiden ja objektien käsittelyä helpottavat funktiot
- asynkroniset funktiot ja asynkroninen iterointi.

Uusimpien ECMAScript-versioiden versiolyhenne esiintyy keskusteluissa välillä eri muodoissa, esimerkiksi ECMAScript 2019:n virallinen versiolyhenne on ES10, mutta siihen saatetaan viitata kirjoissa ja artikkeleissa välillä myös vuosisidonnaisella lyhenteellä ES2019.

TypeScript puolestaan on Microsoftin kehittämä ohjelmointikieli, joka mahdollistaa vahvasti tyypitetyn JavaScript-koodin kirjoittamisen. Tyypitys auttaa parantamaan koodin laatua, sillä se auttaa välttämään virheitä, ja tietyllä tapaa pakottaa kirjoittamaan parempaa koodia. TypeScript noudattaa ECMAScript-standardia ja laajentaa sitä omien ominaisuuksiensa osalta.

Normaaliin JavaScriptiin verrattuna TypeScriptissä pitää määritellä jokaiselle muuttujalle ja oliolle tyyppi. Mikäli valmista sopivaa tyyppiä ei löydy, voidaan määritellä omia tyypityksiä tyypitystiedostoihin. Kuvassa 25 on esimerkkinä yksi tässä projektissa luotu tyyppi (MpstCustomer), jota käytetään kuvassa 26.

```
interface MpstCustomer {
  customerId: number,
  customerName: string,
  customerType: 'ON-PREMISE'|'SAAS'|'OTHER',
  productTypes: string[],
  hasServiceLevelAgreement: boolean
}
```

Kuva 25. Esimerkki olion tyypittämisestä TypeScriptillä.

```
/**
 * Data:
 */

formData: MpstCustomer = {
  customerId: 0,
  customerName: '',
  customerType: 'ON-PREMISE',
  productTypes: [],
  hasServiceLevelAgreement: false
}

/**
 * Methods:
 */

async addCustomer (event: MouseEvent|KeyboardEvent): Promise<void> {
  await this.$http.post(
    urlAction('add', 'ui/customer'),
    this.formData
  )
}
```

Kuva 26. Tässä on esimerkki, miten muuttujalle määritellään tyyppi. Tässä kuvassa on määritelty tyypitys (MpstCustomer) formData-oliolle sekä addCustomer-metodin parametreille ja paluuarvoille.

Selaimet eivät osaa suorittaa TypeScriptia natiivisti, joten TypeScript-kääntäjä kääntää koodin ECMAScriptin mukaiseksi koodiksi selaimia varten. Tässä projektissa TypeScript-koodi käännetään uusimpaan ECMAScript 2019-versioon (ES10), jotta voidaan hyödyntää viimeisimpiä JavaScript-ominaisuuksia.

Suurin osa selaimista ei tue myöskään ECMAScriptin uusimpia versioita, tai ainakaan kaikkia niiden tuomia uusia ominaisuuksia, joten käytössä on vielä erillinen JavaScript-kääntäjää nimeltä Babel. Babel antaa kehittäjälle mahdollisuuden hyödyntää uusien JavaScript-versioiden ominaisuuksia. Se voidaan konfiguroida muuntamaan koodi halutun ECMAScript-versioon, jota kaikki vaaditut selaimet tukevat. Tässä sovelluksessa oli tarve tukea Internet Explorerin versiota 11 (IE11), joka tukee ECMAScriptin ES5-version mukaista JavaScriptiä. Alla kuvassa 27 on yksinkertainen esimerkki siitä, miltä näyttää ES5-version mukaiseksi JavaScriptiksi käännetty TypeScript-koodi.

```typescript
// TypeScript-versio:
type MvgRowItem = { [key: string]: any }

export class ClassWrapper {
  // Props:
  rowItem!: MvgRowItem;
  headerId!: string

  get rowStatusClass(): string {
    return this.rowItem?.[this.headerId] ?? 'light'
  }
}
```

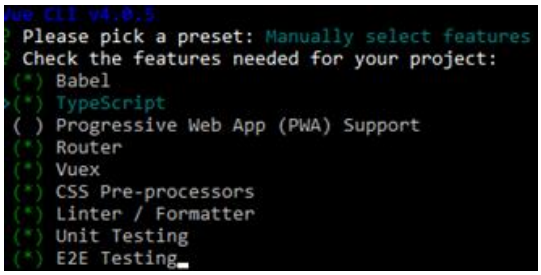
```javascript
// ES5-mukainen JavaScript:
"use strict";
Object.defineProperty(exports, "__esModule", { value: true });
var ClassWrapper = /** @class */ (function () {
    function ClassWrapper() {
    }
    Object.defineProperty(ClassWrapper.prototype, "rowStatusClass", {
        get: function () {
            var _a = this.rowItem;
            var _b = _a === null || _a === void 0
                ? void 0
                : _a[this.headerId];
            return _b !== null && _b !== void 0 ? _b : 'light';
        },
        enumerable: true,
        configurable: true
    });
    return ClassWrapper;
}());
exports.ClassWrapper = ClassWrapper;
```

Kuva 27. Esimerkki TypeScript -> ES5 -muunnetusta koodista, jota myös vanhat selaimet tukevat.

Babelin avulla voidaan ottaa käyttöön myös JavaScriptin tulevia, vielä määrittelyvaiheessa olevia, kokeellisia ominaisuuksia. Tässä projektissa on käytössä mm. C#-kielestä tutut Optional Chaining (a?.b) ja Nullish Coalescing (??) -ominaisuudet, joita on käytetty kuvan 25 rowStatusClass-koodissa. Optional Chaining mahdollistaa syvälle olion rakenteeseen viittaamisen ilman, että tarvitsee joka tasossa varmistaa, että arvo on määritetty eikä se ole null. Nullish Coalescing taas mahdollistaa ei määritettyjen ja null-arvojen korvaamisen tietyllä ns. Fallback-arvolla, joka tulee muuttujan arvoksi, mikäli arvoa ei ole määritetty tai se on null. Aiemmin JavaScriptissä on voinut käyttää tai-ehtoa vastaavaa toiminnallisuutta varten, mutta se muuntaa myös muut ns. epätosi-arvot (0, false, tyhjä merkkijono) Fallback-arvoon. Näiden kahden uuden operaattorin avulla voi luoda siistimpiä ja selkeämpiä ehtoja ja muuttujamäärityksiä, ja antaa kääntäjän huolehtia tarvittavista apumuuttujista, kun viitataan syvälle olion rakenteeseen.

Projekti käyttää myös työkaluja nimeltä Vue-CLI (Command line interface) ja Webpack. Webpack tarvitaan, koska nykyaikaisessa web-sovelluksessa on paljon JavaScript-koodia, ja käytössä on paljon ulkopuolisia kirjastoja. Webpack osaa tunnistaa ulkopuolisista kirjastoista turhan koodin, jota kohdeprojekti ei käytä, ja jättää sen pois käännetystä lopputuotteesta. Webpack osaa myös paloitella koodia eri tiedostoihin, jolloin eri näkymien koodia voidaan ladata tarvittaessa suorituksen aikana pala kerrallaan. Turhan koodin poistaminen ja pilkkominen useampaan tarvittaessa ladattavaan palaan pienentää käännetyn ohjelmakoodin kokoa merkittävästi ja nopeuttaa siten koodin suoritusta selaimessa.

Vue-CLI on luotu helpottamaan Webpackin käyttöä Vue.js-sovelluksissa. Vue-CLI:llä voi luoda Vue.js-projektille pohjakonfiguraation suorittamalla komennon ”vue create projektin-nimi”. Tämän jälkeen Vue-CLI kysyy käyttäjältä, minkälainen projekti halutaan luoda (kuva 28).



Kuva 28. Vue-CLI tukee useita eri tekniikoita, jotka voi määrittää projektin käyttöön.

Kehittäjä voi muokata ja laajentaa luotua pohjakonfiguraatioprojektin tarpeiden mukaan muokkaamalla vue.config.js-nimistä tiedostoa (kuva 29). Tässä konfiguraatiotiedostossa määritellään Vue-CLI:lle ja Webpack:lle projektissa käytetyt kehitys- ja tuotantokääntöasetukset ja mahdolliset Webpack-lisäosat. Tuotantokääntöasetuksilla voidaan määritellä esimerkiksi se, miten käännetyt tiedostot nimetään ja mitä esi- ja jälkikäsittelyitä niille tehdään. Tässä projektissa käytetiin esimerkiksi Webpack-lisäosaa, joka huolehtii siitä, että eri komponenttien käännönaikaiset niin sanotut pala-id:t säilyvät samoina, jotta tiedostojen sisältö ei turhaan muutu ja näin voidaan hyödyntää pidempiaikaisempaa välimuistia.

```
module.exports = {
  runtimeCompiler: true,
  lintOnSave: false,
  publicPath: process.env.NODE_ENV === 'production' ? assetsSubDirectory : `${devProtocol}://${devHost}:${devPort}`,
  outputDir: outputDir,
  devServer: {…}
  pages: {…}
  chainWebpack: (config) => {…}
  css: {…}
  configureWebpack: {
    resolve: {
      extensions: ['.js', '.vue', '.json'],
      alias: {
        '@': resolve('src')
      }
    },
    output: {…}
    performance: {…}
    // Optiomoinnit käytössä vain production-modessa, koska dev-tila ei tarvitse niitä
    ...process.env.NODE_ENV === 'production' && {
      optimization: {
        splitChunks: {
          chunks: 'all',
          cacheGroups: {
            vendor: {…}
            common: {…}
            bootstrapVue: {…}
            // ...
          }
        }
      },
      plugins: […
      ]
    },
    pluginOptions: {…}
}
```

Kuva 29. Esimerkki vue.config.js-tiedoston rakenteesta.

Projektin asetuksiin vaikuttavat myös muut mahdolliset konfiguraatiotiedostot, joita voivat olla esimerkiksi staattisen koodianalyysin työkalujen konfiguraatiot. Tässä projektissa käytettiin staattisen koodianalyysiin ESLint- ja Stylelint-nimisiä työkaluja.

ESLintin avulla validoidaan, että kirjoitettu JavaScript- ja TypeScript -koodi noudattavat asetettuja sääntöjä. Tässä projektissa käytettiin myös Vue-tiimin Vue-komponenteille kehittämää eslint-plugin-vue-nimistä ESLint-lisäosaa, joka osaa validoida ja asettaa säännöt myös Vue-komponenttien templatelle, eli HTML-osalle. Templaten osalta voidaan pakottaa esimerkiksi tietty järjestys HTML-elementtien parametreille. Lisäksi projektissa käytettiin muun muassa eslint-plugin-vue-a11y-nimistä lisäosaa, joka auttaa kehittäjää luomaan Vue-komponenteista saavutettavia, eli sellaisia, että esimerkiksi näkövammaiset voivat niitä käyttää. Tämä tarkoittaa esimerkiksi sitä, että toiminnollisen ikonin kerrotaan olevan oikeasti painike, jota voi painaa ja annetaan sille tekstimuotoinen otsake, joka kertoo, mitä painike tekee. Käytettävät säännöt ja lisäosat määritellään ESLintin konfiguraatiotiedostossa, jonka rakenteesta on esimerkki kuvassa 30.

```
module.exports = {
  root: true,
  parser: 'vue-eslint-parser',
  parserOptions: {
    parser: '@typescript-eslint/parser',
    ecmaFeatures: {
      legacyDecorators: true
    }
  },
  env: {
    node: true
  },
  extends: [
    'plugin:@typescript-eslint/recommended',
    'plugin:vue/strongly-recommended',
    '@vue/standard',
    '@vue/typescript',
    // Static AST checker for accessibility rules on elements in .vue.
    'plugin:vue-a11y/recommended'
  ],
  globals: {…}
  plugins: [
    'vue', // required to lint *.vue files
    'compat' // Lint the browser compatibility of your code
  ],
  settings: {
    polyfills: […]
  },
  // add your custom rules here
  rules: {…}
  overrides: […]
}
```

Kuva 30. Esimerkki ESLint-konfiguraatiotiedoston rakenteesta.

Stylelintin avulla validoidaan kirjoitetun CSS-tyylimäärittelyn rakenne noudattaa asetettuja sääntöjä. Tässä projektissa sääntöjen lisäksi käytettiin muun muassa stylelint-no-unsupported-browser-features-nimistä lisäosaa, jonka avulla varmistetiin, että kirjoitettu CSS on sellaista, että kaikki erillisessä konfiguraatiotiedostossa määritetyt selaimet tukevat sitä. Esimerkiksi IE11:n CSS-tuki on melko rajallinen nykyaikaisiin selaimiin verrattuna. Käytettävät säännöt ja lisäosat määritellään Stylelintin konfiguraatiotiedostossa, jonka rakenteesta on esimerkki kuvassa 31.

```
// Synchronizing .editorconfig and stylelint:
const editorconfig = require('editorconfig').parseSync('./editorconfig')

module.exports = {
  root: true,
  extends: [
    'stylelint-config-standard'
  ],
  syntax: 'scss',
  plugins: [
    'stylelint-scss',
    'stylelint-no-unsupported-browser-features'
  ],
  rules: {…
  }
}
```

Kuva 31. Esimerkki Stylelint-konfiguraatiotiedoston rakenteesta.

Lisäksi tässä projektissa projektille on luotu .editorconfig-tiedosto (kuva 32), jossa määritellään projektissa käytettävät tiedostoasetukset, esimerkiksi sisennystapa, merkistökoodaus, rivinvaihdon tyyli. Editorconfig on useiden eri kehitystyökalujen tukema tapa määritellä nämä asetukset, ja myös projektissa käytetyt Visual Studio 2019 ja Visual Studio Code tukevat tätä tiedostoa. Lisäksi projektin ESLint- ja Stylelint-konfiguraatiot lukevat sisennysasetukset tästä tiedostosta.

```
root = true

[*]
charset = utf-8
indent_style = space
indent_size = 2
end_of_line = lf
insert_final_newline = true
trim_trailing_whitespace = true
```

Kuva 32. Esimerkki .editorconfig-tiedoston asetuksista. Tiedostossa voidaan määritellä eri tiedostotyypeille omat asetuksensa. Tässä projektissa määriteltiin kaikille samat asetukset, jota [*]-merkintä ilmaisee.

7 Yhteenveto

Insinöörityön tavoitteena oli suunnitella ja toteuttaa Mepco-tuotteiden SaaS-palvelun valvontaan räätälöity sovellus ja kehittää sovelluskohtaiset tilasivut. Tavoitteena oli helpottaa ongelmien ratkomista ja auttaa ratkaisemaan niitä ennalta ehkäisevästi.

Työn tekeminen aloitettiin vuonna 2017, kun havaitsin asentajien kanssa käydyissä keskusteluissa, että virheiden selvittäminen pohjautui pitkälti sovelluslokeista löytyviin poikkeuksiin tai muihin virheviesteihin. Esimerkiksi autentikaation virheiden selvittäminen oli usein haastavaa, kun ongelman saattoi aiheuttaa useampi eri asetus. Virheiden syyn selvittäminen taas vaati usein koodin näkemistä, jotta vika voitiin tarkemmin paikallistaa. Näiden keskusteluiden pohjalta Mepcon web-sovelluksille luotiin yksinkertainen tilasivu, josta ilmeni JSON-muodossa, saako käyttöliittymäsovellus yhteyden muihin sovelluskomponentteihin ja saavatko esimerkiksi taustapalvelimet yhteyden tietokantaan.

Tilasivujen kertomia tietoja on myöhemmin laajennettu iteraatioittain havaintojen ja palautteen mukaan. Tilasivuista haluttiin kehittää näkymä, josta näkee yhdellä silmäyksellä kaikkien riippuvuuksien tilan. JSON-muotoinen tilasivu sai käyttöliittymän, jossa ilmaistaan eri komponenttien tila värikoodein ja korostetaan virheen aiheuttaja punaisella/keltaisella tiedolla. Lisäksi tilasivulla on asennuksesta joistain lisätietoja, joista voi olla apua, kun sovelluksen konfiguraatio on sinänsä kunnossa, mutta toimii väärin. Esimerkiksi, jos joku toiminnallisuus on kytketty konfiguraatiossa kokonaan pois päältä, joka pitäisi olla päällä tiettyä ominaisuutta varten. Tilasivu kertoo myös ennaltaehkäisevästi tietoja esimerkiksi lähiaikoina vanhentuvasta HTTPS-varmenteesta.

Tilasivujen jälkeen tarve valvontaportaalille ilmeni, kun toimin itse käyttötuen apuna HOT-koodarina. Havaitsin tukipyyntöjä ratkoessani, ettei tiedossamme ollut kunnolla sovelluksen päivityshistoriaa tai konfiguraation muutoshistoriaa. Siten HOT-koodarien, ja kuin myös muiden sidosryhmien, piti tehdä enemmän työtä, jotta pystyimme selvittämään, mitä versiota asiakas on milloinkin käyttänyt ja mikä konfiguraatio on silloin ollut, jos vastaavanlainen ongelma oli esimerkiksi jo korjattu. Lisäksi käyttötukeen tuli turhaan tukipyyntöjä ongelmista, jotka johtuivat väärästä oletuskonfiguraatiosta, esimerkiksi istunto katkesi liian nopeasti tai muutaman megatavun liitetiedostoa ei voinut lähettää lomakkeen kautta. Nämä konfiguraation korjaukset käsipelein vaativat työtä, jonka asentaja tai parhaimmillaan sovellus itse, voi suorittaa päivitysten yhteydessä, mikäli esimerkiksi asennuksen yhteydessä tarkistettava tilasivu kertoo kyseisestä ongelmasta. Helpottamalla ongelmatilanteiden selvitystä ja korjaamalla niitä ennakoivasti voidaan laskea tukipyyntöjen määrää ja nostaa asiakastyytyväisyyttä.

Valvontaportaali helpottaa myös palvelinten seuraamista, kun sen kautta näkee, ovatko palvelimella käyttöjärjestelmän ja ohjelmistokehysten päivitykset asennettuina. Näin voidaan varmistaa, että asennusautomaatio toimii kaikilla ohjelmistopalvelun palvelimilla ja tietoturva on kunnossa.

Opinnäytetyön aikana toteutettiin valvontaportaalin ja sovelluskohtaisen raportoinnin perusominaisuudet. Ominaisuudet helpottavat sovellusten vianselvitystä ja tietoturvallisuudesta huolehtimista. Haasteina toteutuksessa oli se, että eri sovellukset käyttävät eri ohjelmistokehystä, osassa on .NET Core ja osassa .NET Framework, mikä lisää tarkistettavien asioiden määrää, ja lisäksi kirjasto piti toteuttaa kahta eri ohjelmistokehystä huomioiden. Lisäksi tietoturva ja valvontakirjaston laatu piti huomioida tarkasti, jotta valvonta ei aiheuta uusia tietoturvahuolia tai ongelmia sovelluksen toiminnassa.

Valvontaportaalin tarve on osin muuttunut vuosien varrella, kun palvelinten monitorointia on lisätty ja tietoturvapäivitysten asennusta on automatisoitu. Valvontaportaalin kautta pystytään varmistamaan, että automaatio toimii. Lisäksi se tarjoaa tarkkoja tietoja eri sovellusten tiloista, tilamuutoksista ja sen kautta voi reagoida ongelmatilanteisiin jo ennen kuin asiakas ongelmasta raportoi. Lisäksi sovelluksiin ja tilasivuun integroitu oletuskonfiguraation validointi- ja päivitystoiminto vähentävät virheellisestä konfiguraatiosta johtuvia tukipyyntöjä. Näiden toimintojen avulla voidaan sekä reagoida paremmin virheitilanteisiin että täyttää varmemmin SLA-sopimusten palvelulupaukset, ja sitä kautta saadaan parannettua asiakastyytyväisyyttä.

Jatkokehityksen kannalta valvontaportaalia voisi laajentaa, ja se voisi jatkossa korvata kokonaan nykyisen erillisen sovelluksen, jossa on tietoja asiakkaista ja heille asennetuista sovelluksista, jotta tiedot olisivat yhdessä paikassa ja ne päivittyisivät osin automaattisesti. Jatkossa valvontaportaalin tarve kasvaa, kun Mepco-sovelluksien skaalautuvuusominaisuuksia kasvatetaan mm. hyödyntämällä mikropalveluita nykyistä paremmin.

Lähteet

1. OWASP Foundation. 2017. OWASP Top Ten 2017 - The Ten Most Critical Web Application Security Risks. Verkkoaineisto. <https://www.owasp.org/index.php/Category:OWASP_Top_Ten_2017_Project>. Luettu 06.07.2019.

2. Accountor HR Solutions Oy. 2019. Mepco | Moderni HR- ja palkkaohjelma. Verkkoaineisto. <https://mepco.fi/>. Luettu 10.10.2019.

3. Ristimäki, Pekka. 2019. Turvallisen sovelluskehityksen käsikirja. Verkkoaineisto. <https://vrk.fi/documents/2252790/13063677/Tukimateriaali_VRK_turvallisen+sovelluskehityksen_k%C3%A4sikirja.docx>. Luettu 01.04.2019.

4. Shirhatti, Sourabh. 2018. Hardening IIS. *OWASP.* Verkkoaineisto. <https://www.owasp.org/index.php/Hardening_IIS>. Luettu 10.09.2018.

5. Ristimäki, Pekka. 2019. Turvallisen sovelluskehityksen käsikirja - Tekninen liite 7. Verkkoaineisto. <https://vrk.fi/documents/2252790/13063677/Tukimateriaalit_VRK_turvallisen_sovelluskehityksen_k%C3%A4sikirja_Liite+7_Tekninen+liite_eng_fin.docx>. Luettu 01.04.2019.

6. Price, Mark J. 2019. *C# 8.0 and .NET Core 3.0 – Modern Cross-Platform Development.* Birmingham, United Kingdom : Packt. 9781788478120.

7. Microsoft. 2019. What is ASP.NET Core? Verkkoaineisto. 2019. <https://dotnet.microsoft.com/learn/aspnet/what-is-aspnet-core>. Luettu 22.09.2019.

8. Smith, Steve ja Microsoft. 2019. *Architect Modern Web Applications with ASP.NET Core and Azure.* [toim.] Maira Wenzel. Redmond : Microsoft.

9. Landwerth, Immo. 2016. Introducing .NET Standard. *.NET Blog.* Verkkoaineisto. <https://devblogs.microsoft.com/dotnet/introducing-net-standard/>. Luettu 18.10.2019.

10. .NET Foundation. 2019. .NET Standard FAQ. Verkkoaineisto. <https://github.com/dotnet/standard/blob/master/docs/faq.md>.

11. Microsoft. 2016. Overview of Entity Framework Core - EF Core. Verkkoaineisto. <https://docs.microsoft.com/en-us/ef/core/>.

12. Entity Framework Tutorial. Entity Framework Core. Verkkoaineisto. <https://www.entityframeworktutorial.net/efcore/entity-framework-core.aspx>.

13. Ratcliffe, Stuart. 2018. *ASP.NET Core 2 and Vue.js – Full Stack Web Development with Vue, Vuex, and ASP.NET Core 2.0.* Birmingham, United Kingdom : Packt. 978-1-78883-946-4.

14. Stack Overflow. 2019. Developer Surver Results 2019. Verkkoaineisto. <https://insights.stackoverflow.com/survey/2019>. Luettu 04.11.2019.

15. W3Schools. 2019. JavaScript Versions. Verkkoaineisto. <https://www.w3schools.com/js/js_versions.asp>. Luettu 01.11.2019.