Teollisuusnosturien monitorointityökalun ja käyttöliittymän uudelleensuunnittelu

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Ammattikorkeakoulututkinnon opinnäytetyö

HAMK, Tieto- ja viestintätekniikka

Syksy 2021

Ossi Hakkarainen

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Tieto- ja viestintätekniikka, insinööri (AMK) Tiivistelmä

Tekijä Ossi Hakkarainen Vuosi 2021

Työn nimi Teollisuusnosturien monitorointityökalun ja käyttöliittymän uudelleensuunnittelu

Ohjaaja Toni Laitinen

Opinnäytetyön tavoitteena oli selvittää tilaajayritykselle, miten olisi mahdollista toteuttaa järkevästi toimiva web-pohjainen käyttöliittymä teollisuusnosturien monitorointijärjestelmää varten. Työn suunnittelun lisäksi tarkoituksena oli tehdä prototyyppiversio käytettävästä järjestelmästä.

Opinnäytetyö jakautuu kahteen eri osuuteen. Ensimmäinen osuus sisältää teoriaa, jossa kerrotaan yleisiä asioita C#-, JavaScript- ja SQL-kielistä, sekä perehdytään niiden kirjastojen yleisiin komponentteihin. Työn toinen osuus koostuu varsinaisen prototyypin suunnittelusta ja toteutuksesta. Suunnittelua pohjustetaan työn alkuvaiheilla ja kokonaiskuvalla, jonka jälkeen siinä käsitellään yksityiskohtaisemmin projektin vaatimuksia. Toteutusosiossa puolestaan käsitellään varsinaisen prototyypin luomiseen liittyneitä työvaiheita ja menetelmiä. Tämän lisäksi osiossa tarkastellaan esimerkkien avulla visuaalista toteutusta web-käyttöliittymästä. Toinen osuus myös selittää yksityiskohtaisemmin, miksi erilaisia ratkaisuja päädyttiin valitsemaan eri projektin osuuksiin.

Opinnäytetyössä tuli ilmi kuinka erinäköisiä HTML- ja JavaScript-pohjaisia komponentteja voitiin käsitellä .NET-ympäristössä ja kuinka SQL-tietokantaa pystyttiin käyttämään ja hyödyntämään osana projektia. Opinnäytetyössä käsiteltiin myös tarkemmin JavaScriptillä tehtyä canvas-koordinaatistoa.

Opinnäytetyön lopputuloksena oli dynaaminen web-järjestelmä, jonka pitäisi vastata työn tilaajan odotuksia. Erillisiä uniikkeja web-sivuja prototyyppiin tuli kolmelle pääsivulle yhteensä noin 17 kappaletta. Näiden lisäksi prototyyppiin tuli muita sekalaisia sivuja, kuten asetukset-sivu, jotka eivät tulleet kokonaisuudessaan valmiiksi. Vaikka projektin prototyyppiversiossa eivät kaikki yksityiskohdat olleet täydellisiä opinnäytetyön ollessa lopussa, tuli prototyypistä kaiken kaikkiaan lähes valmis.

Avainsanat Ohjelmointi, JavaScript, C#, käyttöliittymäsuunnittelu
Sivut 38 sivua

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Information and communication Technology

Author Ossi Hakkarainen

Abstract

Year 2021

Subject Redesigning monitoring tool and UI for industrial cranes

Supervisors Toni Laitinen

The aim of the thesis was to find out how it would be possible to implement a sensible, functional web-based user interface for an industrial crane monitoring tool system. In addition to the result of design, the aim was to make a prototype version of the designed system.

The thesis is divided into two parts. The first section contains theory part that explains common knowledge about C#, JavaScript, and SQL languages and familiarizes the reader with their other most used libraries. The second part of the thesis consists of the design and implementation of the actual prototype and its user interface. The design includes the early stages of the project and the overall picture, and it discusses the requirements of the project in more detail. The implementation part is based on the actual work phases of the prototype. In addition, this part of the thesis uses examples to look at visual implementation of the web interface. The second section also explains in more detail why different solutions were chosen for different parts of the project.

The thesis explains how different HTML and JavaScript-based components can be handled in a .NET environment and how the SQL database could be used and utilized as part of a project. The thesis also describes in more detail about the canvas coordinate system that is made with JavaScript.

The result of the thesis was a dynamic web system that should meet the expectations of the client. A total of approximately 17 unique web pages were added to the prototype on the three main pages. In addition to these, other miscellaneous pages were added into the prototype, such as a settings page, which was not finished entirely. Although the prototype version of the project did not have all the fine details completed at the end of the thesis, the prototype was almost complete overall.

Keywords Programming, JavaScript, C#, User Interface design
Pages 38 pages

Sisällys

1 Johdanto

Käyttöliittymä on tärkeä osuus käyttäjäkokemusta. Hyvällä käyttöliittymällä pystytään takaamaan selkeä ja informatiivinen tiedonsiirto käyttölaitteelta käyttäjälle ja päinvastoin. Käyttöliittymän taustalla ns. backendissä tapahtuu usein monia operaatioita, joita normaali loppukäyttäjä ei tiedosta tapahtuvan. Näiden tapahtumien sulavan toiminnan varmistaminen voi olla suunnittelun ja toteutuksen kannalta haastavaa. Tästä syystä kompleksit taustaoperaatiot monien laitteiden ja teknologioiden välillä vaativat erityistä huolellisuutta niitä tehdessä.

Opinnäytetyön tavoitteena on selvittää tilaaja yritykselle, miten toteuttaa järkevästi toimiva web-pohjainen käyttöliittymä teollisuusnosturien monitorointijärjestelmää varten. Itse nosturimalli, jonka monitorointiin käyttöliittymää mm. käytetään, on ns. WTE (Waste To Energy) -nosturi, joita käytetään jätteenpolttolaitoksissa. Jätteenpolttolaitos on laitos, joka tuottaa energiaa erinäisiä jätteitä polttamalla. Uuden käyttöliittymän tulisi ulkonäöltään ja toiminallisuudeltaan jotakuinkin vastata aikaisempaa monitorointijärjestelmää, joka oli rakennettu teollisen käyttöpaneeliratkaisun ympärille. Monitorointijärjestelmällä tarkoitetaan nosturin käyttöliittymän sitä osaa, jota käytetään nosturin automaatiosyklien monitorointiin ja niihin liittyvien asetusten säätämiseen. Lisäksi käyttöliittymän kautta voidaan monitoroida ja konfiguroida jätteenpolttolaitoksen bunkkeri-, eli varastointialueen eri osia. Yksinkertaistettuna monitorointijärjestelmän selvitys ja toteutus tarkoittaa projektin kartoittamista ja prototyyppiversion eteenpäin toteutusta.

Koska vanha exe-pohjainen järjestelmä on edelleen päivittäisessä käytössä, uutta käyttöliittymää pitää miettiä sekä vanhan, että uuden käyttäjän kantilta. Työtä tehdessä on hyvä olla myös mielessä webtekniikan rajoitteet ja hyödyt. Työn aikana pitäisi tulla selville, mitä tietokantaoperaatioita ja muita asiakas- ja palvelinkommunikaatioita pitää tehdä järkevän järjestelmän aikaansaamiseksi. Työssä käsitellään erityisesti niitä teknisiä ratkaisuja, joita onnistuneen lopputuloksen saavuttaminen edellyttää. Opinnäytetyössä tuloksena syntyy prototyyppi / proof of concept versio käytettävästä järjestelmästä. Työtä toteutetaan pääsääntöisesti C#-, JavaScript- ja SQL-kielien avulla. Ohjelmointiympäristönä toimii Microsoftin Visual Studio 2019.

2 Tietokantojen käsittely

Kun lähdetään käsittelemään tietokantoja, on hyvä muistaa minkälaisen datan ja järjestelmän kanssa ollaan tekemisissä, jotta voidaan valita oikea tietokantatyyppi. Yksi yleisimmistä tietokantamalleista mitä käytetään, on edelleen relaatiotietokanta. Relaatiotietokanta on nimensä mukaisesti loogisesti järjestelty kanta, jossa entiteetit ovat relaatiossa, eli suhteessa toisiinsa (Bryla, 2004, s. 3).

2.1 SQL ja relaatiotietokanta

SQL lyhenne muodostuu sanoista Structured Query Language, eli suomeksi Strukturoitu kyselykieli. SQL on yksi tunnetuimpia ei-proseduraalisia kieliä. Ei-proseduraalinen kieli tarkoittaa käytännössä, että käyttäjän tarvitsee vain määrittää mitä tehdään, ei miten. SQL mahdollistaa myös käskyjen upottamisen valittuun koodikieleen. Upotettujen käskyjen vastauksia voidaan koodissa käyttää hyväksi asettamalla vastaukset muuttujiin. Tämän lisäksi ohjelmointikielen koodissa on mahdollista rakentaa dynaamisesti kyselyitä, joita lähetetään tietokantaan suoritettavaksi. (Hovi, 1998, ss. 9 - 10)

Relaatiotietokanta perustuu asiakokonaisuuksiin, joita esitetään relaatiotietokannoissa tauluina. Taulut sisältävät sarakkeita ja rivejä. Sarakkeen ja rivin erot ovat selvät; sarake sisältää samassa tietotyypissä olevaa dataa, kun taas puolestaan rivi sisältää relaatiossa olevaa dataa, joka ei ota kantaa siihen mitä tietotyyppiä viereinen kenttä sisältää. (Hovi ym., 2005, s.8) Sarakkeen nimi, esim. henkID (Kuva 1) on kyseisen sarakkeen selventävä tekijä, jonka alla kaikki tietotyypit ovat kokonaislukuja. Ei-relaatiopohjaisissa tietokantamalleissa taas talletettu data saattaa muistuttaa esim. JSON dataformaattia.

Kuva 1 . Esimerkki tietokannan riveistä ja sarakkeista.

henkID	Sukunimi	Etunimi	Osoite	Kaupunki
1	Meikäläinen	Matti	meikalaisenkatu 1	Helsinki
2	Meikäläinen	Mikko	meikalaisenkatu 1	Helsinki
9	Olematon	Oskari	nollakatu 0	Tampere

henkID	Sukunimi	Etunimi	Osoite	Kaupunki
1	Meikäläinen	Matti	meikalaisenkatu 1	Helsinki
2	Meikäläinen	Mikko	meikalaisenkatu 1	Helsinki
9	Olematon	Oskari	nollakatu 0	Tampere

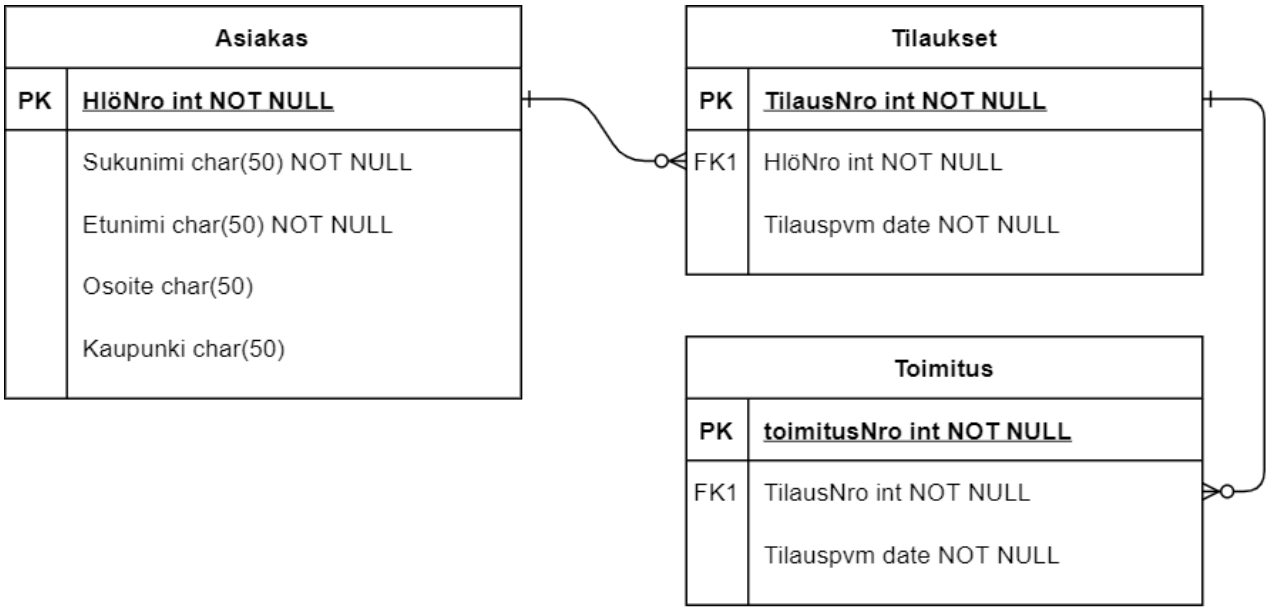
Perus- ja viiteavaimet (eng. primary key, foreign key) ovat oleellinen osa relaatiotietokantamallia. Perusavain toimii datan yksilöllisenä tunnisteena ja viiteavain, kuten nimestäkin voidaan päätellä, viittaa toisen taulun perusavaimeen tai yksilöityyn tunnisteeseen. Näiden avulla taulujen välissä pystytään muodostamaan yhteyksiä ja riippuvuuksia toisistaan. (Hovi ym., 2005, ss. 8 - 9)

2.2 Käsiteanalyysi ja sen tarkoitus

Kun tehdään käsiteanalyysiä, suorituskykyyn vaikuttavia asioita ei mietitä, vaan asioita käsitellään ideaalitilanteina ja kaavaillaan likimääräisesti mitä tietokantaan halutaan. Käsiteanalyysin työstäminen on yksi aloitus kohtia projektia ja tietokantaa suunniteltaessa. Kaavioilla kuvataan ja havainnollistetaan kantaan tulevaa ja lähtevää tietoa. Tästä syntyy lopulta käsitemalli (eng. conceptual model), jonka perusteella varsinaista oikeaa tietokantaa on hyvä lähteä työstämään. Käsitemalli on graafinen suunnitelma. Yleisin tapa kuvata käsitemallia on erilaisina käsitekaavioina (eng. Entity Relationship Diagram Model). (Hovi ym., 2005, ss. 32 - 33)

Aiemmin mainitut perus- ja viiteavaimet tulevat näkyviin käsitekaaviossa (Kuva 2).

Kuva 2. Esimerkki ER-kaaviosta.



3 Palvelinpuolen ja asiakaspuolen ohjelmointi

Vaikka ohjelmointia tapahtuu samassa sovelluksessa, sen palvelin- ja asiakaspuolen ohjelmointi eroavat paitsi ohjelmointikielissään myös käyttötarkoituksissaan ja ongelmissa. Nämä voivat myös yleisesti erota käyttöjärjestelmäympäristöissään. Palvelimet ovat vuorovaikutuksessa asiakaspuolen (selaimen) kanssa HTTP:n avulla. Tapahtumista verkkosivulla, kuten linkin painamisesta, haun suorittamisesta ja lomakkeen lähettämisestä välitetään HTTP-pyyntö asiakkaalta palvelimille. Nämä pyynnöt sisällyttävät tunnisteita, joilla voidaan määrittää haluttu toiminto esim. objektin poistamiseksi. (Mozilla, n.d)

3.1 Palvelinpuolen ohjelmointi

Palvelinpuolen ohjelmointi on kätevää, koska se mahdollistaa tietojen tallentamisen tietokantaan, sivustojen dynaamisesti rakentamisen ja HTML- ja muun tyyppisien tiedostojen palauttamisen. Datan palautus onnistuu myös renderöintitarkoituksiin sopivilla asiakaspuolen verkkokehyksillä. Tällä voidaan vähentää palvelimen taakkaa ja dataliikenteen määrää. Jos tiedot on koottu tietokantaan, voidaan ne helposti käsitellä muiden järjestelmien yhteydessä ja avulla. Tällöin päivitys tuotteille onnistuu helposti. Erityisen huomattavaa tämä on verkkokaupoissa tai varastoissa, joissa moni osasto saattaa käyttää samoja tuotteita ja tavaroiden saldojen määrää. Palvelimella on monia käyttötarkoituksia ja sitä on mahdollista käyttää moniin erilaisiin tietokantaoperaatioihin. Palvelimella voidaan suunnata eri työkalujen tuloksia esim. useammalle erilaiselle laitteelle, joka vastaanottaa palvelimelta tietoja. (Mozilla, n.d)

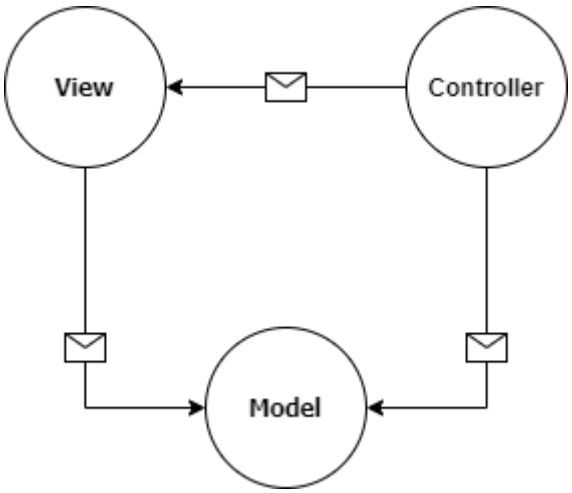
3.1.1 C#

C# on olio-ohjelmointipohjainen kieli. C#:n ominaisuuksiin kuuluu sen tyyppiturvallisuus, joka takaa monesta muustakin ohjelmointikielestä tutun syntaksirakenteen. Muuttujien rakennetyyppien avulla (esim. readonly-kentät) käyttäjä pystyy lisäämään turvallisuutta ja vähentämään virheiden mahdollisuutta. Näiden lisäksi C# mahdollistaa objektien tallentamisen siten, että niitä on mahdollista käyttää dynaamisesti. Lisäksi se mahdollistaa sisäisten objektien käytön kevyille rakenteille. C#:n ASP.NET sisältää monia vaihtoehtoja palvelinpuolen ratkaisuihin, joista yksi on MVC ratkaisumalli. (Microsoft, 2021.-c)

Model View Controller eli MVC on datamalleja käyttävä suunnittelukuvio. MVC:n avulla pystytään erittelemään käyttöliittymä, datamallit ja erilaiset applikaation logiikat. MVC:tä käyttävässä web-sivussa ohjain vastaa erilaisista pyyntöreitityksistä ja toimii mallin kanssa tekemällä monenlaisia toimintoja. Näiden lisäksi ohjain määrää ja hakee näkymän, jossa itse sivu renderöidään datan perusteella. (Microsoft, n.d. -a)

Havainnollistavan diagrammin (Kuva 3) kautta voidaan havaita, miten data liikkuu MVC-mallin sisällä.

Kuva 3. MVC-diagrammi.



MVC-mallin käyttäminen selkeyttää koodin kirjoitusta ja ylläpitoa. Helppolukuisuutta ja skaalautuvuutta voidaan lisätä entisestään käyttämällä Partial views tyylistä rakennetta. Partial views tarkoittaa nimensä mukaisesti osittaista näkymää. Osittaisen näkymän hyöty tulee selkeydessä. Yhden tiedoston koodia voidaan jakaa useampaan lohkoon, jolloin yksittäinen lohko ei sisällä liikaa sisältöä. Samalla samankaltaista koodia ei toisteta kirjoittamalla sitä uudestaan, vaan yksittäisiä koodilohkoja voidaan uudelleen käyttää ns. avainsanoina/komponentteina. (Smith ym., 2021)

MVC-mallia ovat hyödyntäneet monet eri ohjelmointikielet ja ohjelmistot vuosien aikana. C#:ssa mallia käyttää ASP.NET Core MVC-viitekehys ja razor pages, jotka hyödyntävät sitä sivun luonnissa. ASP.NET Coren viitekehys käyttää razor view engineä käyttöliittymän näkymän renderöintiin. Koska razor on joustava ja kompakti C#-koodin käsittelyssä, pystytään sillä sekoittamaan asiakas- ja palvelinpuolen koodia keskenään. Razor myös mahdollistaa dynaamisesti generoidun sisällön luonnin palvelimella. Razor pages- ja MVC-mallit on rakennettu kysely- ja vastauslogiikan päälle käsittelemään UI tapahtumia.(Microsoft, 2021. -b)

Razoria ei kuitenkaan saa sekoittaa Blazor-viitekehykseen, joka kuitenkin voi käyttää razor rakennetta, mutta ei ole razor pages yhteensopiva (Microsoft, n.d. -b).

Razor-syntaksissa CSHTML:ään voidaaan määritellä omia koodilohkoja, joiden rakenne saattaa muistuttaa <script> tagin sisältöä normaalista HTML-rakenteesta. Määriteltävälle lohkolle asetetaan @-merkki indikoimaan lohkon aloituskohta (Kuva 4). Lohkon lopetuksen sen sijaan määrää aaltosulkumerkkin lopetus.

Kuva 4. CSHTML esimerkki.

```
@{
    var tervehdys = "Hello, ";
    var nimi = "John";
    var vkoPv = DateTime.Now.DayOfWeek;

    var viesti = tervehdys +nimi+ "! Today is "  + vkoPv;
}

<p> @viesti</p>
```

```
Hello, John! Today is Sunday
```

Aaltosulkumerkki ei kuitenkaan välttämättä lopeta koodia, jos koodin rakenne jatkuu loogisesti, kuten ehtolauseilla. Tällöin esimerkiksi else ei vaadi uutta @-merkkiä vaan CSHTML-syntaksi ymmärtää koodin jatkumisen viimeisen ehdon aaltosulkumerkkiin asti. CSHTML:ssä voidaan käyttää myös melkein kaikkea C#:sta tuttuja rakenteita.

Koodissa (Kuva 5) voidaan havaita @-merkillä tuotu asiakaslistan. Listaa voidaan iteroida silmukkarakenteiden avulla. Koska objektit on mahdollista tuoda CSHTML-tiedostoon myös merkkijono-tyyppisenä, tarkoittaa tämä sitä, että tiedostoon ei tarvitse tehdä erikseen koodisyntaksia, vaan se voidaan generoida kokonaan C#-tiedostossa.

Kuva 5. Silmukkarakenne, CSHTML.

```
@model Models.asiakkaat

    <h1>Asiakkaat</h1>

    @foreach (var asiakas in model.asiakkaat)
    {
        <p>@asiakas</p>
    }

    <!--  tai normaali for loop  -->

    @for (var i = 0; i < model.asiakkaat.Length; i++)
    {
        var asiakas = model.asiakkaat[i];
        <p>@asiakas</p>
    }
```

Vaikka CSHTML-tiedostoon voidaan tehdä ohjelmiston logiikkaan vaikuttavia komponentteja, ei ole suositeltavaa kirjoittaa sinne suuria määriä koodia. Yhtenä syynä tähän on koodin skaalautuvuus, jossa uudelleenkäytettävyys saattaa kärsiä, mikäli jokaiselle tiedostolle on luotu logiikka paikallisesti sen itsensä ympärille. Toisena syynä on koodin selkeämpi ulkoasu. Ulkoasulla tarkoitetaan ohjelmiston funktionaalisuuden ja graafisten elementtien erottamista eri tiedostoihin ihmissilmälle luettavampaan muotoon. Näistä syistä yleisenä käytänteenä on C# Action- tai vastaavan kooditiedoston hyödyntäminen siistin ja järkevän ulkoasun ylläpitämiseksi.

3.1.2 LINQ

LINQ eli Language-Integrated Query on C#:ssa toimiva kirjasto, jonka avulla voidaan tehdä SQL tyylisiä kyselyitä kokoelmaan objekteja. Kyselyillä voidaan tehdä perustason operaatioita, kuten järjestys- , filtteröinti- ja ryhmittelyoperaatioita. Eri operaatioiden avulla isompikin data saadaan formatoitua oikeaan muotoon. LINQ tukee kaikkea mahdollista, aina SQL server tietokannasta XML-dokumenteihin asti, sekä myös IEnumerablea tai geneerisiä IEnumerable<T>-rajapintoja tukevia kokoelmia. (Microsoft, 2021.-d)

Kuvassa 6 nähdään yksinkertainen esimerkki LINQ:lla tapahtuvasta suodatuksesta. Kuvassa alkuperäinen nimilista esitellään matriisisina ennen kyselyä, jonka jälkeen itse kysely

muodostetaan. Vaikka esimerkissä kyselylle määritetään spesifinen datatyyppi (IEnumerable<string>), ei tämä ole pakollista, vaan kyseisessä kohdassa myös var-avainsana on validi. Huomioitavaa implisiittisessä muuttujatyypissä on kuitenkin se, että sen tyyppi määritellään kyselyn palautuksen perusteella, joka voi johtaa ei haluttuihin epäkohtiin ja virheisiin myöhemmin koodia kirjoittaessa. Haku etsii nimilistasta kaikki J-kirjaimella alkavat nimet, jonka jälkeen se valitsee nimen ja palauttaa tuloksen. Tulos on tässä tapauksessa uusi suodatettu lista nimiä. Kun kysely on muodostettu, voidaan sitä käyttää esimerkiksi silmukkarakenteen avulla.

Kuva 6. LINQ-suodatus nimilistalle.

```
using System;
using System.Collections.Generic;
using System.Linq;

class LINQEsimerkki
{
    static void Main()
    {

        // Alkuperäinen data
        string[] nimiMatriisi = new string[] {"", "Aatami", "Juhani", "Johannes", "Olavi","Maria","Helena" };

        // Määritellään haluttu kysely
        IEnumerable<string> nimiHaku =
            from nimi in nimiMatriisi
            where nimi.StartsWith("J")
            select nimi;

        // Iteroidaan kaikki kyselyn hakutulokset
        foreach (var nimi in nimiHaku)
        {
            Console.Write(nimi + ", ");
        }
    }
}
// Output: Juhani, Johannes,
```

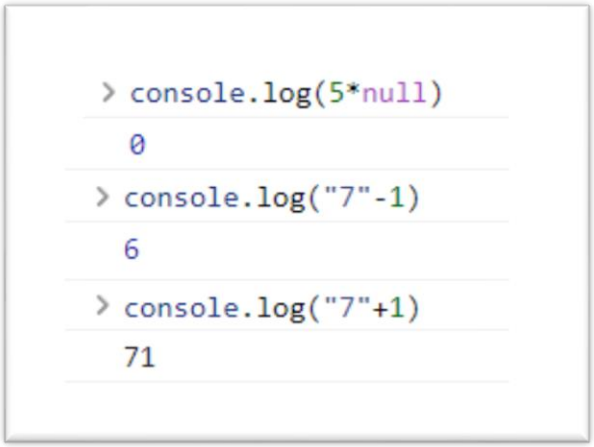
Iteroidessa listaa, kysely ”aktivoituu” samalla tavalla kuin funktiokutsu, vasta tällöin data haetaan määritellystä paikasta. Lykätyn toteutuksen hyviin puoliin kuuluu parannettu suorituskyky, koska arvoa kysellään vasta silloin kun sitä todella tarvitaan. Jos lykättyä toteutusta ei haluta käyttää, voidaan se pakottaa kutsumalla ToArray tai ToList methodia. (Microsoft, 2021.-a)

3.2 JavaScript

JavaScript on asiakaspuolen ohjelmointikieli. Se pyörii isäntäympäristön sisällä, yleisimmin nettiselaimessa (Stefanov, S. 2008. s.23). JavaScript on dynaamisesti tyypitetty, jonka tarkoituksena oli alun perin tehdä siitä aloittelijaystävällinen. Todellisuudessa aloittelijoiden tekemiä virheitä voi olla vaikea huomata, koska järjestelmä ei erikseen mainitse niistä ennen ohjelmakohdan suoritusta. JavaScriptin nimi juurtaa alkunsa Javan suureen suosioon. Nimensä lisäksi JavaScriptillä ei ole varsinaisesti mitään tekemistä staattisesti tyypitetyn Javan kanssa. (Haverbeke, 2014, s.6)

Ohjelmointikieli suorittaa ns. automaattista tyyppikonversiota. Tällä tarkoitetaan tyyppien sekoittamista toistensa kanssa operaattorien avustuksella. Toisin kuin staattisesti tyypitetyillä ohjelmointikielillä, kuten C#, JavaScript ei välttämättä vaadi erillistä konversiofunktion kutsumista tyypin vaihtamiseksi. Esimerkissä (Kuva 7) tyyppikonversiot tapahtuvat operaattorien avulla hieman eri tavalla JavaScriptissä kuin C#:ssa.

Kuva 7. JS operaattori tyyppikonversiot.



Jos samat operaatiot kirjoitetaan esim. C#-kielellä, ensimmäinen tulos tuottaa tyhjän, kun taas toinen operaatio tuottaa operaatiotyypistä johtuvan virheviestin. Vain ainoastaan kolmas vaihtoehto tuottaa saman operaation, jossa tyyppi vaihtuu automaattisesti merkkijonoksi. (Haverbeke, 2014, s.18)

JavaScript hyödyntää tapahtumaohjattua ohjelmointimallia, jossa tapahtumankäsittelijäfunktiot suorittavat käyttäjän syötteen perusteella toimintoja. Syötteet eli tapahtumat eivät ilmaannu omia aikojaan, vaan ne muodostuvat perinteisesti käyttäjän interaktiosta sivustolla. Koska tapahtuman käsittelijöitä suoritetaan asynkronisesti, ei niitä suoriteta välittömästi, toisinkuin tapahtumia itseään. Tämä voi johtaa siihen, että sivu ei ole latautunut, vaikka tapahtumakäsittelijää olisi jo kutsuttu. (Flanagan, 2000, ss. 162 - 163)

3.2.1 JavaScript prototyypit

JavaScriptiä voidaan kutsua prototyyppipohjaiseksi ohjelmointikieleksi. Prototyyppi käyttäytyy kuin mallipohja, josta voidaan periä metodit ja tyyppitiedot. JavaScriptin prototyyppi antaa myös mahdollisuuden lisätä uusia arvoja objekteihin muuallakin kuin konstruktorissa. Yrittämällä asettaa arvoa muualla kuin konstruktorissa ilman prototyyppiominaisuutta ei onnistu kuten kuvasta (Kuva 8) voidaan huomata. (Mozilla, 2021)

Kuva 8. Prototyyppi-ominaisuuden käyttäminen.

3.2.2 Bootstrap

Bootstrap on vaihtoehtoinen CSS ja JavaScript lisäosa, jonka ovat kehittäneet Mark Otto ja Jacob Thorton. Bootstrap tarjoaa paljon avoimeen lähdekoodiin perustuvia tyylivaihtoehtoja nettisivujen muokkaamista varten. Bootstrap keskittyy mobiili ensin periaatteeseen, jossa suurin osa sen komponenteista ja vaihtoehdoista ovat skaalautuvia älylaitteille. Bootstrapin

käytön etu on, että kaikkia boxeja ja HTML-komponentteja ei tarvitse keksiä ja kirjoittaa uudelleen, vaan ne voidaan nopeasti luoda käyttämällä valmista pohjaa. (Bootstrap team, n.d)

3.3 Logiikka (PLC)

”Ohjelmoitava logiikkaohjain eli PLC (eng. Programmable Logic Controller) on teollisuuden automaatioon käytettävä pelkistetty tietokone. Ohjaimet voivat automatisoida tietyn prosessin, koneen toiminnon tai jopa koko tuotantolinjan.” (Unitronics, n.d)
Logiikkaohjaimen keksijänä voidaan sanoa olevan Dick Morley, joka sai idean kehittää ensimmäisen PLC:n, Modicon 084, vuonna 1968 uuden vuoden krapulassa. Morleyllä oli monen samankaltaisen projektin lopetuspäivämäärää tulossa uuden vuoden päivänä. Samanlaiseen toistuvaan työhön turhautuneena hän tajusi, että kaikkiin projekteihin voisi luoda yhtenäisen laitteen, joka voisi korvata releiden suuren käytön. Tämä johti ensimmäisen logiikkaohjaimen luontiin. (Pattiengineering, n.d)

Logiikkaohjaimen päätarkoituksena on toimia rautaläheisenä yhdistettynä tietokoneena, joka voi jakaa käskyjä laitteille. Logiikan toimintatapa toimii erittäin suoraviivaisesti. Ensimmäiseksi logiikka vastaanottaa erilaista dataa eri lähteiltä kuten antureilta. Logiikka ottaa tarkasteltavaksi datan, jonka jälkeen se antaa komennon uloslähtöjä varten perustuen dataan ja ennalta määriteltyihin parametreihin. Logiikka tekee toimintaansa ns. syklissä, jota se kiertää läpi kunnes toisin käsketään. (Wayand, 2020)

4 Kehittämistyön tarkoitus ja tavoite

Projekti aloitettiin jo kesällä, jolloin raamit työlle asetettiin. Projektin tavoitteena kesällä oli saavuttaa helppo ja dynaaminen web-pohjainen käyttöliittymä, joka rakennettaisiin vanhan pohjan perusteella. Vanhassa pohjassa nosturien monitorointikäyttöliittymä oli tehty

käyttäen .exe pohjaista windows riippuvaista template-järjestelmää. Erinäisistä haasteista johtuen, kuten päällekkäisien yhteyksien takia, järjestelmää oli erittäin työläs ylläpitää. Vanha järjestelmä myös sisälsi monia staattisia, mutta kustomoitavia elementtejä, jotka mahdollisesti haluttiin korvata dynaamisilla elementeillä. Web-pohjainen käyttöliittymä mahdollistaisi monia päällekkäisiä yhteyksiä toisiaan häiritsemättä. Tällöin monitorointityökalua pystyisi käyttämään sulavasti moni operaattori samanaikaisesti. Tämän lisäksi web-pohjaisessa käyttöliittymässä pystyttäisiin rakentamaan responsiivista sivua, jonka avulla käyttöliittymä pystyisi tukemaan montaa eri resoluutiota selaimelle vaihtamatta lähdekoodia.

5 Projektin suunnittelu ja toteutus

Vanhan käyttöliittymän korvaavaa järjestelmää oltiin aloitettu tekemään jo ennen virallista opinnäytetyön aloitusta. Järjestelmä toimi MVC-mallin pohjalta, jossa sivut oli suunniteltu toimimaan synergisesti vanhan järjestelmän kanssa. Kehittämissuunta projektille kuitenkin oli painotettu uudelleenkäytettävyyteen ja skaalautuvuuteen, jota vanha staattinen järjestelmä ei pystynyt toteuttamaan. Skaalautuvuudella projektissa tarkoitettiin suurimmaksi osaksi kahta erillistä konseptia; verkkosivun responsiivisuutta mobiillilaitteilla ja projektin dynaamista skaalautuvuutta isompiin eri kokonaisuuksiin.

Verkkosivua tarkastellaan usein monella eri laitteella ja näytönkoolla, näin on myös monitorointityökalussa. Vaikka pääasiallinen keskittyminen on ns. Full HD-resoluutioissa näytöissä, on responsiivisuus (eli mukautuvuus) oleellinen elementti projektissa. Responsiivisuudesta vastaa pääasiallisesti bootstrap, joka nopeuttaa työskentelyä valmiiden tyylielementtien avulla. Yksityiskohtia voidaan hienosäätää erillisellä tyylitiedostolla, jossa uusia luokkia voidaan luoda tai vanhoja ylikirjoittaa.

Projektin dynaamisuus on vielä isompi kokonaisuus, johon kuuluu niin tyylitiedostot kuin tietokantasuunnittelu. Työmäärään vähentämiseksi on koodin hyvä automaattisesti hakea ja lisätä valittu data dynaamisesti sivulle. Tämä tarkoittaa sitä, että data pitää löytyä

dynaamisesti haettavasta sijainnista esimerkiksi tietokannasta. Koska vaihtuvat datamäärät ja elementit ovat yleisiä, pitää sivun tyyliasuun määrittää joustavat ehdot, kun väistämättömät päällekkäisyydet tulevat sivulla esille.

5.1 Suunnittelu

Kehitystyötä suunniteltaessa oli hyvä tarkastella, miten loppukäyttäjä sekä käyttöönottaja käyttää lopullista tuotetta. Projektissa aikana käyntiin viikoittaisia palavereita, joista selvisi, minkälaisia toimintoja eri sivuilla oli, ja miten ne toimivat vanhassa järjestelmässä. Kaavioiden lisäksi kyseisistä tiedoista tehtiin vaatimusmäärittely (Kuva 9), jonka avulla sivujen hahmottaminen olisi helpompaa.

Kuva 9. Alustava vaatimusmäärittely sivuille.

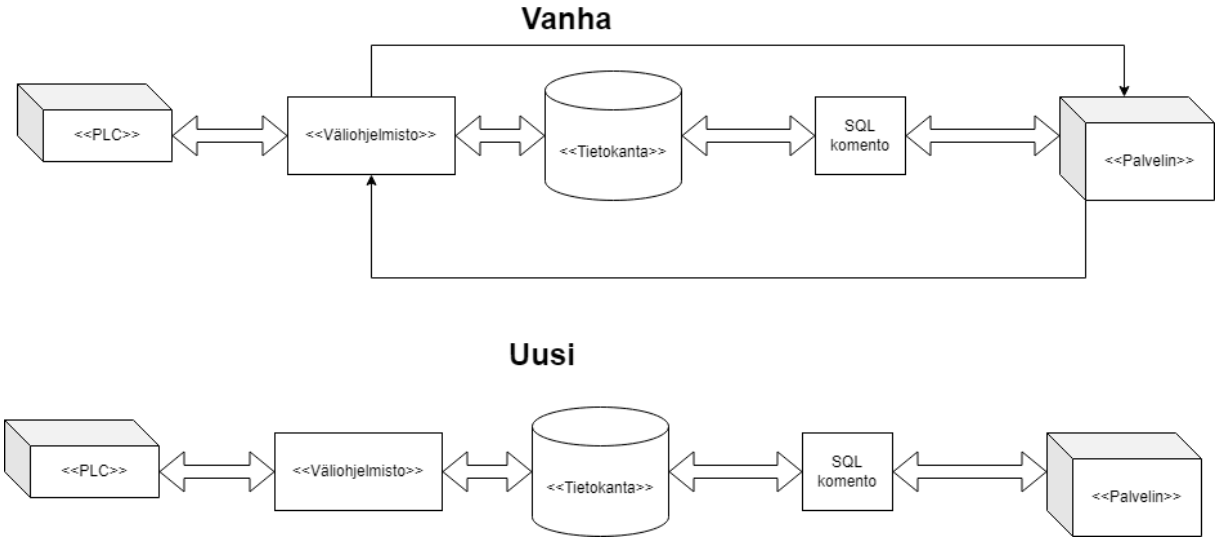
Päätoiminto	Ominaisuus	Käyttötapaus
Nosturien monitorointi	Nosturin tilan monitorointi graaffisesti	Operaattorin on nähtävä ns. yhdellä vilkaisulla yhden bunkerin nosturien mahdolliset korkeus-, painotiedot ja vikatilat
Bunkkerin objektien monitorointi	Bunkkerissa olevien paikkojen niiden tilan ja nosturien paikkojen monitorointi	Operaattorin on nähtävä bunkkerin solun tila ja missä mahdollinen nosturi tällä hetkellä bunkkerissa kulkee
Porttien hallitseminen	Erillaisien porttien ohjaus mekaniikka	Operaattorin on pystyttävä hallitsemaan erillaisia portteja ja liikenne valoja mahdollisimman yksinkertaisesti
Bunkkerin pohjapiirrustusten hallitseminen	Bunkkerin "pohjapiirrustuksia" suunnitteleminen ja aktiivisen lataaminen	Operaattorin tai vastaavan henkilön pitää pystyä suunnittelemaan bunkkerin solujen pohjapiirrustus ,jotka ohjaavat bunkkerin nostureiden toimintaa
Diagnostiikkan monitorointi	Diagnostiikka tietojen eli virheiden ja viestien näyttäminen graaffisesti	Operaattorille pitää käydä ilmi jos jossain nosturissa on vikatila menossa ja tästä tulee ilmoittaa punaisella. Erillaiset viestit tulee myös tallentaa ja niitä pitää pystyä tarkastelemaan jälkeen päin
Bunkkerin aikataulutuksen hallitseminen	Bunkkerissa olevat pohjapiirrustusten aikataulutus viikkokalenterin mukaisesti	Suunnittelija voi luoda viikkojärjestyksen missä hän voi määritellä jokaiselle päivälle erikseen oman järjestyksen missä erillaisia pohjapiirrustuksia ajetaan
Bunkkerin korkeus tietojen monitorointi ja hallitseminen	Bunkkerin pinnan korkeuden ja solun arvon monitoroiminen ja arvon ylikirjoitus	Operaattorin pitää pystyä ylikirjoittamaan virheellisiä bunkkerin korkeustietoarvoja ja näitä arvoja pitää pystyä monitoroimaan helposti graaffisien elementtien ja värien avulla
Enerigan kulutus monitorointi	Energian kulutuksesta olevat olennaisien tietojen nyttäminen	Monitoroinnissa on hyvä näkyä kuinka paljon energiaa siirroissa on kulutettu ja ilmaista näitä graaffisesti
Asetusten hallinta	Asetuksia jotka muokkaavat käyttäjä kokemusta ja perus arvoja käyttöliittymässä	Käyttöönottajan pitää pystyä säätämään helposti perus asetuksia jotka löytyisivät yhdeltä sivulta koottuna
Laitteiden ja hätäpysäytyksien monitorointi	Laitteiden ja hätäpysäytyksien näyttäminen graaffisesti	Operaattorille pitää käydä ilmi jos jossainlaitteessa on hätäpysäytys aktiivisena ja tästä tulee ilmoittaa. Erillaiset viestit tulee myös tallentaa ja niitä pitää pystyä tarkastelemaan jälkeen päin

Koska entinen järjestelmä oli edelleen käytössä, pystyttiin tämän toimintaa tarkastelemaan ja tarvittaessa parantelemaan. Selviä ongelmakohtia projektin lähtövaiheilla löytyi datan kulusta eri projektin kohdissa. Kommunikaatio logiikkaohjaimen kanssa ei ollut selvillä, joten

tätä piti selvittää. Ensimmäisiä laajoja konsepteja mitä lähdin työstämään, olivat käsitekaaviot, joiden avulla pohjapiirustuksia ja suunnitelmia olisi hyvä hahmotella.

Projektin tavoitteet olivat selvillä, mutta yksittäisien sivujen ja datan virtaus oli vielä epäselvää, joten näitä kohtia yritettiin selventää tekemällä UML-relaatioita datan ja sivujen välillä. Edellisessä ohjelmistossa, rajoitteet pakottivat tekemään haastavampia datan kulun malleja, mutta koska uudessa käyttöliittymässä näitä rajoitteita ei ollut modernimman web-tekniikan ansiosta, oli mahdollista yksinkertaistaa datan kulkua. Yksinkertaistettu kaavio (Kuva 10) kuvaa datan kulkua projektissa.

Kuva 10. Kaavio datan kulusta projektissa.



Uuden mallin mukaisessa järjestelyssä tietokanta ja palvelin olisivat suuremmassa roolissa keskustelun ja tiedon vaihtumisen kannalta. Tapahtumat päivitettäisiin tauluihin, joista logiikka (PLC) pystyisi niitä hakemaan väliohjelmiston avulla. Eroavaisuutta vanhaan tulisi siinä, että yksi ns. turha välikäsi ja siinä tulevat omat mutkat jäisivät tällä mallilla välistä ja tiedon kulku pysyisi aina erittäin yksinkertaisena. Yksinkertaisuus helpottaisi selvästi koodin ylläpitoa ja sen virheenjäljitystä.

Tietokannan tauluista ja niiden relaatiosta toisiinsa luotiin käsitekaavion periaatteita noudattava malli helpottamaan uuden järjestelyn kanssa. Alkurakenteiden jälkeen uusia alustavavia rakenteita auttavia Er-kaaviota tehtiin esimerkiksi Layout kokonaisuuteen, jossa

dataa tultaisiin hakemaan logiikka ohjaimelta ei-määrätyistä paikoista. ER-kaaviolla saatiin selkeytettyä datan kulkua logiikalta kolmeen eri tietokanta tauluun. Er-kaavion periaatteen vastaisesti, samaan konseptiin suunniteltiin alustavasti minkälaisia kyselyjä tauluista pitäisi suorittaa. Tämä selkeytti hahmottamis- ja luomisprosessia ennalta tuntemattomien taulujen kanssa.

5.2 Prototyyppi version toteutus

Kuten aiemmin mainittu, prototyyppiversiota oli lähdetty työstämään jo ennen virallista tietokantasuunnitelmaa. Sivujen määrää ja niiden vanhaa sisältöä pystyttiin katselemaan vanhasta käyttöliittymästä. Sivuja olisi kokonaisuudessa viisi pääsivua, joiden alaisuudessa sivujen määrä riippuisi sivun tarkoituksesta sekä nosturien ja bunkkerien määrästä. Prototyyppiin kaikkia sivuja ei ollut tarkoitus saada täysin valmiiksi, vaan tarkoituksena oli tarkastella vanhaa käyttöliittymää ja tämän perusteella hahmotella web käyttöliittymää ja miettiä kuinka uudet sivut voitaisiin tehdä.

Projektissa meni aikaa koodin arkkitehtuuriin tutustuessa ja omatoimisessa opiskelussa aiheeseen liittyvistä menetelmistä. Version hallinta oli tärkeä osa prototyyppiversion toteutusta. Versionhallinnalla taattiin koodin säilyvyys ja jaettavuus. Uudet versiot koodista käytiin läpi vähintään viikoittain, jolloin koodi tarkasteltiin ja koodiin tehtiin mahdolliset uudet korjausehdotukset.

Testatessa ja projektia kehitettäessä tehtiin projektiin vanhan pohjan kautta uusi tietokantarakenne, jota pystyttiin hyödyntämää turvallisesti. Tietokanta ja sen taulujen sarakkeet saattoivat vaihtaa muotoaa päivittäin. Tästä syystä tietokantaan ei aluksi ollut järkevää tehdä erillisiä klustereita ja optimoituja indeksejä. Toinen syy klustereiden ja optimoitujen indeksien tekemättä jättämiselle oli niiden siirtämisen haastavuus viralliseen tietokantaan.

5.2.1 Palvelimen ja tietokantayhteyden käyttö

Kommunikoinnin aikaansaamiseksi palvelimen ja tietokannan kanssa oli luotava erillinen tietokantayhteys. C#:ssa tietokantayhteyden luomiseen voitiin käyttää System.Data.SqlClient kirjastoa/nimiavaruutta. Konfigurointiasetuksiin määritettiin erikseen tietokantapolku, jotta tietokantakyselyitä käyttävät funktiot voivat käyttää polkua helposti. Esimerkkikysely (Kuva 11) näyttää, kuinka perushakukysely voidaan muodostaa merkkijonona.

Kuva 11. Esimerkkikysely (Microsoft, n.d. -b).

```
private static void ReadOrderData(string connectionString)
{
    string queryString =
        "SELECT OrderID, CustomerID FROM dbo.Orders;";
    using (SqlConnection connection = new SqlConnection(
               connectionString))
    {
        SqlCommand command = new SqlCommand(
            queryString, connection);
        connection.Open();
        using(SqlDataReader reader = command.ExecuteReader())
        {
            while (reader.Read())
            {
                Console.WriteLine(String.Format("{0}, {1}",
                    reader[0], reader[1]));
            }
        }
    }
}
```

Esimerkkissä kyselyfunktion parametrinä toimii ”connectionString”(Kuva 12). Tämä on lista parametrejä, kuten salasana, tietokannannimi ja käyttäjätunnus.

Kuva 12. Connection String esimerkki.

$$Server = myServerAddress; Database = myDataBase; UserId = myUsername; Password = myPassword;$$

Haettu data tallennettiin paikallisesti ohjelmistossa ASP.NET Core MVC-mallin mukaisesti malliin (Model). Mallin tallennusformaatti toimi hyvin tietokanta- ja listamaisesti, jossa yhden otsikon alle kerättiin siihen sarakkeeseen sopivat tietueet. Mallin dataa voitiin tämän

jälkeen helposti suodattaa esimerkiksi LINQ avulla. Normaali kysely mallista oli melkein yhtä yksinkertaista kuin englannin lukeminen. Rakenne muodostui usein where funktiosta (

Kuva 13) ja nuolifunktion avustamalla indeksin valinnalla.

Kuva 13. Yksinkertainen where-lause.

```
modelData.Where(x=>x != null && x.ID == selectedID);
```

Tilanteesta riippuen oli myös mahdollista, että koodissa tarvittiin yhdistelmiä eri mallien rakenteista. Koska mallit vastasivat SQL tietokannan rakennetta pitkälle, pystyttiin tarvittavat JOIN komennot toteuttamaan LINQ avulla. Kuva 14 demonstroi kuinka kolmen mallin yhdistäminen tapahtuu käytännössä.

Kuva 14. LINQ-kysely.

```
IEnumerable<'a> allObj = from loc in allLocations
                         join bCell in b2
                            on loc.location equals bCell.location into bLocationJoin
                         from bCell in bLocationJoin.DefaultIfEmpty(new BLayoutCells { })
                         join bLayout in b1
                            on bCell.layoutID equals bLayout.layoutID
                            into bIDJoin
                         from bLayout in bIDJoin.DefaultIfEmpty(new BLayout { })
                         select new { loc = loc, bCell = bCell, bLayout = bLayout };
```

Asetelmassa oletetaan, että mallit ovat jo käytössä. Tässä tapauksessa mallit tai taulut, joista data haetaan, ovat: päämalli allLocations, sekä apumallit b1 ja b2. Kuten normaalissa SQL kyselyssä joukot liitetään toisiinsa ID tunnisteen tai vastaavan yksikäsitteisen arvon avulla. Tämän lisäksi kyseinen komento täyttää tyhjät rivit tyhjillä mallivaihtoehdoilla. Liitos toteutetaan molempiin apumalleihin, jonka jälkeen mallit valitaan yhdistettyyn anonyymipohjaiseen objektiin. Tällä tavalla lokaatio mallin yhdistelmää voidaan esimerkiksi käyttää yhdessä silmukkarakenteessa, eikä käyttäjän tarvitse miettiä erikseen silmukan indeksejä, joita sisältyisi mallien määrän verran.

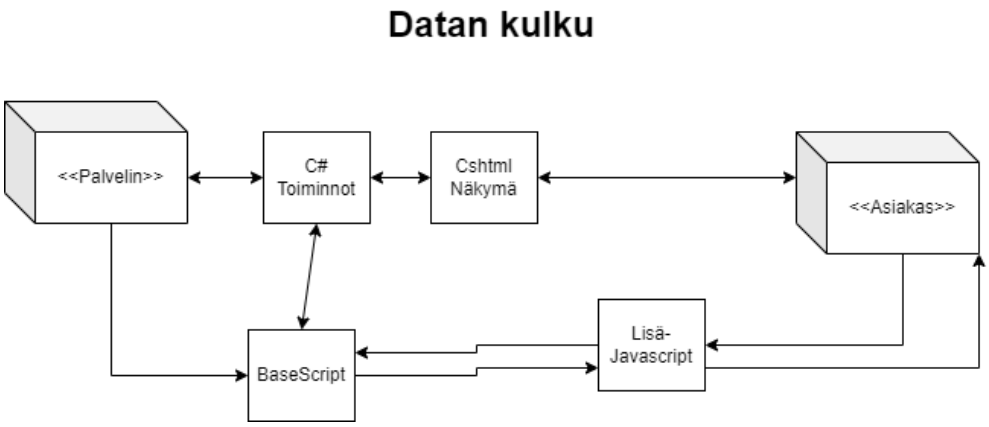
Kun data oli suodatettu ja haluttu data valittu, oli sen käyttö nopeaa sekä asiakaspuolella visualisoinnissa ja serveri puolella laskennassa.

5.2.2 Asiakas- ja palvelinkommunikaation implementointi

Normaali kommunikaatio asiakkaan ja palvelimen välillä tapahtui kustomoidulla Razorin syntaksin avulla (kts. luku 3.1.1). Razor yhdessä C#-toimintojen kanssa vastasi suuremmasta osasta palvelimen ja asiakkaan välisestä viestittelystä. C#-toimintojen avulla pystyttiin vähentämään CSHTML:llään liiallisen funktionaalisen koodin muodostumista. Tämän lisäksi erotellulla pystyttiin lisäämään dynaamisuutta ja koodin uudelleen käyttöä. Serveri ja erillisessä JavaScript tyylisessä kommunikaatiossa luotettiin tehtyyn alustavaan mallipohja (base) skriptiin.

Diagrammi (Kuva 15) kertoo, kuinka palvelin välittää tietoa asiakkaalle.

Kuva 15. Datan kulku palvelimelta asiakkaalle.



Ns. mallipohja-skriptin avustuksella palvelinpuolen tapahtumasta pystyttiin kutsumaan tarpeen tullen myös asiakaspuolen selaimessa toimivaa JavaScript tiedostoa. Jotta kuitenkin js tiedostoa pystyttäisiin hyödyntämään monessa eri tapahtumassa ja kontekstissa, selaimen ja palvelimen piti pystyä välittämään skriptin käynnistyksen lisäksi muunlaistakin dataa. Datan välitys aloitettiin muodostamalla viesti. Viestin muodostumisen jälkeen tämä välitettiin normaaliin tapaan asiakaspuolen skriptille. Skripti tunnistaa viestin eri käskyt ja näiden perusteella muodostaa konstruktorin arvot. Jos datan välitys tapahtui ensimmäisen kerran jälkeen, ja skripti oli jo ”aktiivisena” tällöin ainoastaan viestin sisällön data otettiin uudelleen käyttöön.

Normaalit C# kommunikointiviestit toimivat lähettäjien perusteella, jotka triggeröivat erilaisia funktioita. Funktioille (Kuva 16) kirjoitettiin ja sidottiin event-tyylisiä tapahtumalaukaisijoita, joiden avulla lähettäjän ja sen arvon saaminen oli alun jälkeen nopeaa. Arvojen tarkastelu ja niiden triggeröinti tapahtui samankaltaisesti kuin normaaleissa .NET tapahtumissa. Arvon muuttuessa ja kynnyksen ylittyessä tapahtuma laukaistiin, jolloin paikallinen tapahtumafunktio sai kutsun.

Kuva 16. Net Event-funktio.

```
using System;

public class Program
{
    public static void Main()
    {
        Console.WriteLine("Hello World");

        var c = new Counter(0);
        c.ThresholdReached += _ThresholdReached;
        c.Add(1);

    }

    static void _ThresholdReached(object sender, EventArgs e)
    {
        Console.WriteLine("Kynnys ylitettiin.");
    }
}
```

Muokattua funktio kutsua (Kuva 17) helpottaa se, että jokaiselle ei erikseen tarvitse määrittää rakennetyyppejä ensimmäisen luonnin jälkeen. Normaaleja tapahtumafunktioita voitiin tämän jälkeen soveltaa syvällisempiin funktioihin, luomalla esimerkiksi lähettäjä arvon tai muuttuvan arvon perusteella olevia tilakoneita. Tämän tyyppisillä tilakoneilla pystyttiin tekemään monenlaisia ajastuksia ja sivunvaihdoksia tilanteen mukaan.

Kuva 17. Muokattu funktio kutsu.

```
// Initialisointi
public Button refreshButton = new Button("Refresh", "");

  //Tapahtuman Bindaus
  refreshButton.valueChanged = _refreshButtonClicked;

  // Sender funktio
  private void _refreshButtonClicked(BaseBaseControl sender, bool value)
    {
        redrawRequest = true;
    }
```

Koodiesimerkeistä voidaan huomata samankaltaisuuksia toistensa kanssa.

Monimutkaisemmasta ei-valmiiksi tehdystä kommunikaatiosta, voidaan hyvänä esimerkkinä käyttää canvaksen toimintaa, johon liittyi myös serverin ja canvaksen välinen kommunikaatio. Canvakseen haluttiin sisällyttää sivujen perusteella erilaisia 2d-elementtejä. Serveri ja canvaksen välisessä kommunikoinnissa viestin pituus ja rakenne on tärkeä. Tyypillisen viestin täytyy sisältää mm. korkeus, leveys, paikka x, paikka y, ja käsiteltävän objektin tyyppi jne. Pakollisista tiedoista tuleekin jo huomattava määrä dataa, jotka on siirrettävä. Tämän takia viestiin ei kannata pakata kuin pakolliset objektit. Esimerkiksi jos halutaan päivittää tietoja yhdestä solusta, tulee objektien tiedot säilyttää molemmissa asiakas ja palvelinpuolella, tällöin vältetään iso datavirta objekteille, joita pitää jatkuvasti päivittää ruudulle uudella datalla. Tyypillinen datan määrä yhdelle objektille voi olla merkkijonon määrä esim. >100 merkkiä tarkoittaa jo 0,1 kilotavun kokoista määrää. Vaikka viestin koko ei itsessään tunnu suurelta voi suurempien viestien jatkuva käyttö monessa paikassa samanaikaisesti haitata operaation nopeaa toimintaa. Kaava 1 demonstroi, kuinka yksi merkki on yhden tavun kokoinen, jolloin voidaan todeta edellä mainittu yksikkömuunnos oikeaksi.

Kaava 1. Merkin koko.

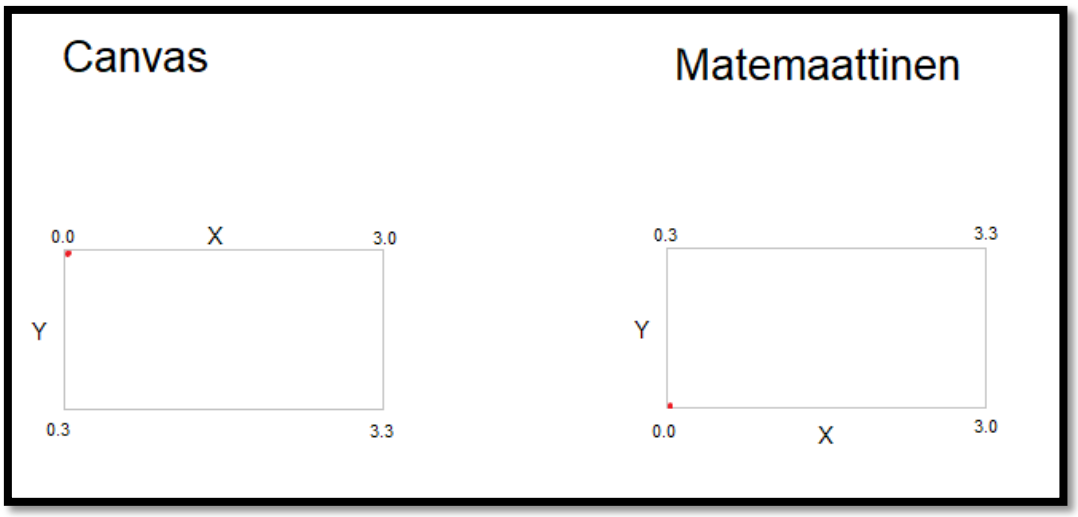
$$1\ char = 1\ byte$$

Objektin koko voidaan huomata jo 100 objektilla. Jos oletetaan, että viesti ei ole optimoitu ja jokainen objekti sisältää 100 merkkiä, tarkoittaa tämä jo 10 000 merkkiä / per haluttu päivitysnopeus. Jos vielä oletetaan, että liikenne menee molempiin suuntiin ei optimoidulla viestillä, on jatkuva yhteys per sekunti 2*10kt. Tämä fiktiivinen yhteys tarkoittaa 0.16 megabitin data määrää vain yhdelle asiakkaalle. Viestin liiallisen määrän vähentämiksi voidaan asiakaspuolelta pyytää päivitystietoja, esimerkiksi pelkän objektin tunnisteen (ID) perusteella. Tämä leikkaa jo huomattavasti liikenteen määrää, koska tunnisteet ovat harvoin yli 10 merkkiä pitkiä.

5.2.3 Piirtotapahtumien ja paikkakoordinaattien ongelmien ratkaisu

Iso osa projektin graaffisista piirto-operaatioista tapahtui canvas elementin kautta. Canvas mahdollisti omien elementtien piirtämisen valmiiksi määriteltyyn paikkaan. Eri elementtien luonnista voitiin tehdä yksinkertaistetut funktiot, joilla esim. kolmion tai kuvan piirto voitiin toteuttaa mutkattomasti. Viesti kertoi, montako elementtiä pitäisi piirtää, ja kyseisien elementtien objektien ominaisuudet. Näiden avulla pystyttiin muodostamaan pohjakokonaisuus nosturin alueelle. Yksi ongelmista, joka piti ratkaista, oli koordinaattien ja hiiren paikkaerot. Eroavaisuudet syntyivät monesta eri syystä. Esimerkkinä yhdestä näistä oli vaatimus pohjan ”väärinpäin” näyttämiseen. Syy vaatimukselle löytyy canvaksen piirron aloituskoordinaatista. Canvaksessa ja yleisesti ottaen monessa ohjelmointiin ja tietoteknisissä liittyvissä koordinaatistoissa (Kuva 18) voi nähdä y-akselinen väärästä päästä aloittamisen.

Kuva 18. Koordinaatisto vertailu.



Ensimmäinen versio pohjan kääntämisestä oli luotu käyttämällä y-akselissa unaarista negaatiota muuttuja-arvon edessä. Menetelmällä tehty kääntäminen kuitenkin loi ongelmia hiiren osoittimen ja canvaksen koordinaattien välillä. Korjauksena tähän epäkohtaan löytyi ratkaisu, jossa ennen viestin käsittelyä koordinaatti olisi jo käännetty oikeaan formaattiin asettamalla korkeuskoordinaatti negaatiolla varustettuna ja lisäämällä tähän canvaksen oma korkeuskoordinaatti. Kaavalla (Kaava 2), kuvan oikea koordinaatti y-akselilla saadaan näkyviin.

Kaava 2. Y:n laskeminen kaavalla.

$$y = (-(y1) + canvasY)$$

Kaava voidaan nyt kääntää JavaScript koodiksi. Kuvassa (Kuva 19) näkyvä funktio ottaa viestin y arvon, parsii sen negatiiviseksi kokonaisluvuksi, ja lisää tähän canvaksen oman korkeuden, asettaen akselin kohdalleen.

Kuva 19. Y:n korkeuskoordinaatti.

```
y: (-(parseInt(msgy)) + this.canvas.height)
```

Akselien oikein asettelun jälkeen hiiren paikkakoordinaatit toimivat normaalisti. Yksittäisen ”normaalin” valinnan lisäksi, valintaa tuli pystyä soveltamaan pidemmälle, jolloin hiiren valinnan moitteeton toiminta oli tärkeää. Vanhasta järjestelmästä tutuista valinnoista, oli eräs valinta ns. ”maalausvalinta” tai selkeämmin sanottuna monen ruudun valinta. Valitut valintaruudut maalatuivat asetetulla värillä tai kuvilla, joiden yhdistelmää pystyttiin muokkaamaan yläreunasta plus ja miinus painikkeiden avulla. Valitut solut saatiin tarkastelemalla käyttäjän ensimmäisen klikkauksen ja toisen klikkauksen paikkakoordinaatteja verrattuna solujen koordinaatteihin. Tällä tavoin yhdestä solusta voitiin varmistaa, että se oli sisältynyt käyttäjän kaksiulotteiseen valintaan. Tarkistusta tehtiin relatiivisen yksinkertaisella funktiolla ja ehtolauseiden yhdistelmällä. Ensimmäiseksi haluttiin tietää mihin suuntaan kahden klikin tapahtuma kohdistui. Jolloin tarkastettiin ensimmäisen klikkauksen paikka ja verrattiin sitä kakkostapahtumaan kuvassa (Kuva 20) näkyvien ehtolauseiden avulla.

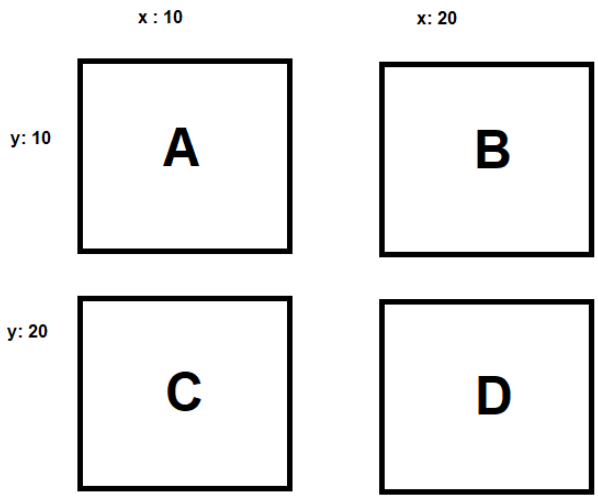
Kuva 20. Valinnan ehtolausekkeet.

```
if (rect2.y < rect1.y) vertical = "up";
else vertical = "down";

if (rect2.x < rect1.x) horizontal = "left";
else horizontal = "right";
```

Kuvasta (Kuva 21) pystytään havainnollistamaan edelliset ehtolauseet (Kuva 20). Valitsemalla esim. ensimmäiseksi valinnaksi D (20, 20) ja toiseksi valinnaksi A (10, 10) tarkoittaa tämä ehtolauseiden mukaan (10 < 20), ylös ja vasemmalle.

Kuva 21. Solujen monivalinta.



Suunnan saamisen avulla, paikan tarkistamista pystyttiin hyödyntämään tarkemmin. X ja y koordinaatteja voitiin nyt vertailla suunta-alueen koordinaatteihin uusilla ehtolauseilla (Kaava 3), enempää virhekohtia miettimättä.

Kaava 3. Uudet valinta-alueen ehtolauseet

$$Solu.x \leq Valinta1.x \;\&\&\; Solu.x \geq Valinta2.x$$

$$\&\&$$

$$Solu.y \leq Valinta1.y \;\&\&\; Solu.y \geq Valinta2.y$$

Uusien ehtolauseiden avulla pystyttiin nyt värjäämään solun haluamalla värillä tai tallentamaan sen ID myöhempään käyttöä varten. Viestin lähettämisen jälkeen, ID:n käsitteleminen palvelimella oli jokseenkin suoraviivaista toimintaa, josta palautus takaisin canvakselle oli riippuvainen solun omista arvoista.

5.2.4 Suurennettavan paikkakoordinaatiston kehittäminen

Canvaksen paikkakoordinaattien määrä saattoi kasvaa riippuen projekteista moninkertaiseksi, jolloin ruututilaa piti allokoida jotenkin. Tilanpuutetta oli helpointa ratkaista pienentämällä kaikkia kuvia saman verran. Pienentämällä kaikkia kuvia huomattavasti kuitenkin loi toisenlaisen ongelman, missä käyttäjän oli vaikea nähdä mitä kuvia tietyissä elementeissä oli. Koska kaikki tieto haluttiin näkyviin yhdelle välilehdelle, päädyttiin canvakseen rakentamaan suurennusominaisuutta (Zoom). Käyttämällä valmiiksi määritettyä canvaksen konstruktorielementtejä, peruszoomauksen luonti onnistui ilman suurempia ongelmia. Konstruktoriin luontiin tapahtumankuuntelijat, jotka vastasivat hiiren näppäin painalluksien rekisteröinnistä. Tapahtumakuuntelijoiden luonnin jälkeen, voitiin kutsua käyttäjän keskirullan pyörityksen perusteella aktivoituvaa funktiota, johon välitettiin parametrinä tapahtuma (event) objekti. Objektin wheelDelta ominaisuuden avulla pystyttiin laskemaan zoomausarvo. Tämän jälkeen arvo syötettiin uuteen funktioon, jossa asetettiin rajat zoomaukselle. Funktiossa zoomaus tapahtui pääasiallisesti kahden valmiin canvas-funktion avulla (translate, scale). Lisäksi zoomauksessa toimi kustomoitu oma funktio, joka otti parametrinä edelliset tallennetut paikkakoordinaatit.

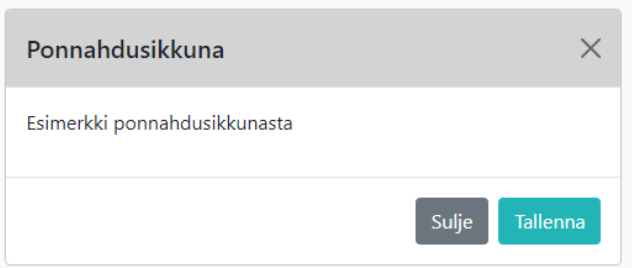
Haasteeksi zoomauksessa osoittautui sen tallennus. Sivua piti aluksi päivittää jatkuvasti, jotta canvaksen koordinaattien kuvat ja symbolit saatiin vaihdettua. Tämä tarkoitti sitä, että aina kun sivu päivittyi, meni zoomaus ”default” arvolle. Tallennuksen teki erittäin hankalaksi se, että tarkennuksen kohta canvaksella saattoi olla missä tahansa käyttäjän kursorin määrittelemässä kohdassa, kun sivua haluttiin päivittää. Relatiivisten sijaintien ylläpitämisestä vastaava matriisi tarvitsi aina edellisen sijainnin koordinaatit, joka olisi tarkoittanut, että jokainen käyttäjän tekemä liike zoomauksessa olisi pitänyt kirjata ja pitää muistissa. Sijainnin ylläpitämistä helpotti, kun alkuperäistä koodia kommunikaatiossa muutettiin nykyiseen järkevämpään suuntaan. Uudistuksena sivupäivitystä vaativat objektit tallennettiin ja kutsuttiin vain id:n avulla. Tällä tavoin pystyttiin suurennuksessa käyttämään paikallisia edellisiä arvoja, ja näin ollen kiertämään ongelma.

5.2.5 Canvaksen lisäominaisuuksien käsittely palvelinpuolella

Luvussa 5.2.3 käsiteltyjen valintaominaisuuksien lisäksi canvaksesta piti pystyä klikkauksen avulla kutsumaan erillisiä funktioita. Näihin funktioihin kuului ponnahdusikkunan tyyppinen lisäikkuna. Yleisesti lisäikkunalla yritettiin saada täydennystä canvaksen tietoihin. Lisäikkunan avulla voitiin selkeyttää käsiteltävää tietoa, esimerkiksi suurentamalla fonttia tai kuvia, sekä antamalla käyttäjälle lisätietoja soluun liittyen.

Ikkunan tapahtuman luonti tapahtui luomalla solulle tyyppi, joka oli klikattavassa muodossa ja ponnahdusikkuna tyyppinen. Kun kyseiset parametrit oli annettu solulle, voitiin asiakaspuolen skriptistä, tapahtuman jälkeen lähettää palvelimelle tunniste (ID). Palvelin havaitsee tunnisteen ja sen perusteella tekee uuden ikkunan. Itse ponnahdusikkunan luonti sisältöineen tapahtui samankaltaisesti kuin muutkin sivut. Ainoana eroavaisuutena oli bootstrap:llä ja tyylin kanssa luotu muokattu läpinäkyvyys ominaisuus. Läpinäkyvyys ominaisuuteen (Modal) oli mahdollista tyyliominaisuuksien kautta säätää tekstin ja ponnahdusikkunan kokoa. Käyttämällä saatua tunnistetietoa ponnahdusikkunaan päivitettiin data, josta pystyttiin suodattamaan vain halutut tiedot näkyviin. Lähes jokainen tehty ponnahdusikkuna sisälsi poistumisnapin, mikä sijaitsi yleensä oikeassa yläreunassa. Nappia painamalla ikkuna piilotettiin käyttämällä ”visible” attribuuttia. Jos ponnahdusikkuna (Kuva 22) sisälsi muita nappeja, pystyttiin nappien tapahtumat sitomaan erillisiin funktioihin.

Kuva 22. Ponnahdusikkuna.



Canvaksessa ponnahdusikkunoiden käyttö toi helpotusta moneen kohtaan, jossa vanhassa järjestelmässä luotettiin nyt canvaksessa tehtävään ominaisuuteen. Näistä merkittävimpiä helpotuksia toiminnan kannalta oli pinnan korkeuden ylikirjoitusominaisuus, jota käsiteltiin ensin JavaScript tapahtuma kuuntelijalla. Jotta jokaista solua voitiin kuunnella erikseen, piti

niihin tehdä dynaamisesti tapahtumakuuntelija klikkauksen yhteydessä, tai bindata jokaiseen yli 50 soluun tapahtumankuuntelija erikseen. Tapahtumakuuntelijalla kuunneltiin näppäinpainalluksia ”keypress” avainsanan avulla (Kuva 23).

Kuva 23. JavaScript-kuuntelija.

```
document.addEventListener('keypress', (event) => {
  let name = event.key;
  let code = event.code;
  console.log(name);
  console.log(code);
}, false);
```

JavaScript kuuntelija osoittautui kuitenkin huonoksi vaihtoehdoksi uudessa toteutuksessa monestakin eri syystä. Tapahtumakuuntelijoiden erillinen bindaaminen sekavoitti selvästi koodia. Järjestelmän piti nyt tietää mitä solua oli klikattu, välittää tämä tieto serveripuolelle, jonka jälkeen se palautti id:n ja asetti kyseiseen soluun tiedon kuuntelijasta. Vasta tämän jälkeen varsinaista tekstiä pystyttiin alkamaan editoimaan. Tekstin erillinen editoiminenkin toi haastetta tällä metodilla, koska canvaksen olisi pitänyt päivittää tekstin paikkaa koko ajan saadakseen se pysymään solun sisällä ja estämällä vanha tekstin näkyminen. Editoiminen siis olisi tarkoittanut, että jokaisesta näppäinpainalluksesta canvas piirtyisi uudelleen, tai erillinen objekti piirrettäisiin vanhan tekstin päällä, antaen illuusion päivittyvästä näytöstä. Molemmissa ratkaisuissa oli omat haasteensa. Onneksi helpotusta saatiin tähän käyttämällä, tässäkin tilassa valmiiksi raamitettua, ponnahdusikkuna valintaa. Ponnahdusikkunalla vältettiin turha canvaksen päivitys, ja koska ponnahdusikkuna komento tuli suoraan palvelin päädystä ei viestejä välitelty montaa eri kappaletta turhaan. Kuten aiemmin mainittua myös tässä kohdassa hyödyttiin ponnahdusikkunan antamasta lisätilasta, jossa pystyttiin näyttämään lisää oleellista tietoa solusta ja tässä tapauksessa editoimaan solun tietoja.

5.2.6 Graafisien-ominaisuuksien luonti

Canvaksen grafiikkaelementtien ohella sivuilla tuli olla staattisempia normaaleja HTML elementtejä. Näiden tekeminen oli hyvin sivukohtaista, mutta yleisenä nyrkkisääntönä oli niiden helppo muokattavuus koodissa, sekä tyylin vaihto tyylitiedostossa.

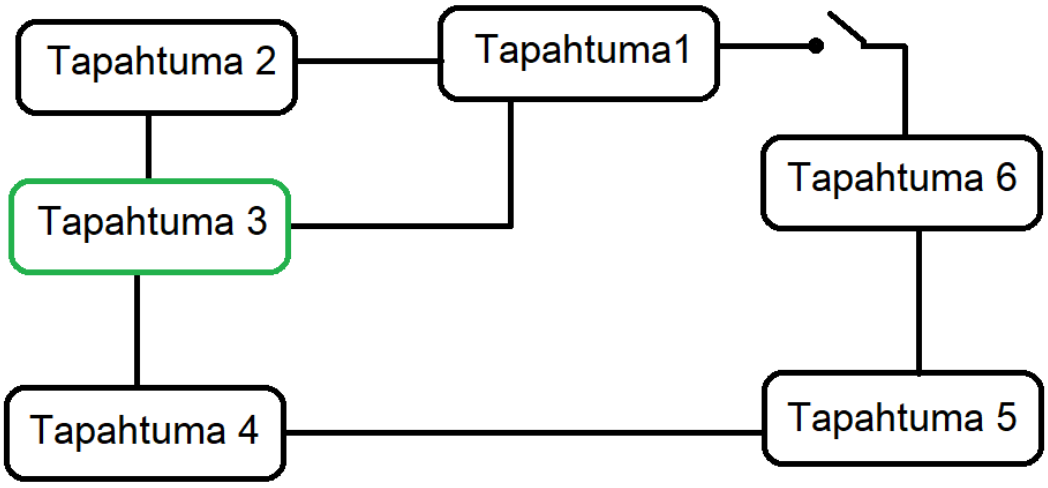
Vallitsevia HTML-elementtejä olivat html taulukot (<table>). HTML-taulukkoja oli monenlaisia, mutta tavallinen taulukkotyyppi kuitenkin koostui otsikosta, ja sen alle koostuvista tiedoista. Taulukoille oli määritetty oma C#-luokka, jonka avulla datasta pystyttiin nopeasti luomaan HTML-pohjainen taulukko. Taulukon tyyliä voitiin muokata yksittäiselle riville tai otsikolle myös nopeasti, koska CSS-luokat luotiin valmiiksi C#:n puolella. Toisinaan tuli tilanteita, jossa valmis C#-luokka ei tarjonnut tarpeeksi joustavuutta muokattavuudessaan. Tällöin HTML-koodi voitiin rakentaa dynaamisesti erikseen C#:n sisällä. Tarve, jossa kyseinen dynaaminen luonti oli pakollista, tuli aikataulun sisältä. Aikataulu koostui normaalin lukujärjestyksen tapaisesta viikkokalenterista, jossa tunteja oli tässä tapauksessa täydet 24-tuntia, seitsemälle päivälle. Päivien lisäksi taulukkoon tuli mahduttaa päiväkohtaisiin otsikkoihin, mitä ohjelmaa tietyllä päivällä ajetaan. Ohjelmia piti pystyä vaihtamaan painikkeista, jotka lopulta sisälsivät ohjelman nimen ja tulivat viikonpäivän otsikon alle. Erilaiset tuntien omat ohjelmat taas tuli merkata värein, jotta ne pystytään erottamaan toisistaan selkeästi. Syy miksi valmista luokkaa ei tässä vaihtoehdossa kuitenkaan pystytty hyödyntämään tällä kertaa, johtui tunti kohtaisista ohjelmista. Uuden järjestelmän toivomuksissa oli, että aikataulut pystyttäisiin merkkaamaan aikaisempaa joustavammin. Aikaisemmassa järjestelmässä aikatauluja ei pystytty merkkaamaan tarkemmin kuin tunnin tarkkuudella käyttöliittymässä.

Ensimmäinen versio kyseisestä joustavammasta taulukko vaihtoehdosta tehtiin jakamalla staattisesti <td> solun sisältö neljään osaan 15 minuutin intervalleja varten. Logiikka toimintojen puolella rakensi HTML-taulukon ja täytti yhden <td> tagin neljällä tagilla. Joustavuuden ja koon yhdelle diville antoi CSS määritykset. Käyttämällä minimi ja maksimi korkeuksia voitiin yhden td tagin sisältö helposti jakaa neljään osaan. Parannus ideaa kyseiseen vaihtoehtoon saatiin käyttämällä flex-grow määrittelyä ja lisäämällä solun sisältö vain silloin kun on tarve. CSS-määrittelyn pääideana oli se, että solun sisältö kasvaa aina, jos sillä on sille tilaa. Näiden tyyli vaihdoksien avulla oli mahdollista myös tehdä jopa minuuttikohtainen aikataulutus, jossa yksi div kasvoi aina sen perusteella, oliko tuntikohtaisen ohjelman sisällä väri sama. Vähentääkseen selaimen taakkaa, voitiin samat värit tarkastamaan ja liittämään yhteen div-tagiin riippuen minuuteista. Esimerkki tapauksena, jos tunnin sisällä kaikki objektit olivat samaa luokkaa, pystyttiin varmuudella

toteamaan, että solun sisältöön ei tarvita kuin yksi div sadan prosentin korkeudella, jolloin se täytti koko td:n sisällön.

Yksittäisiä graaffisia-elementtejä olivat SVG-kuvat. SVG-kuvia voitiin luoda ja muokata Inkscape nimisessä ilmaissovelluksessa. SVG-kuvien hyviin puoliin kuului sen skaalautuvuus eri resoluutiolla. Yhdelle kuvalle ei välttämättä tarvinnut määrittää kuin minimi ja maksimi korkeus- ja leveysarvot CSS:ssä, jolloin kuva itsestään venyi elementin sääntöjen mukaisesti. Vaikka suurin osa SVG-kuvista oli staattisia, pystyttiin niitä soveltamaan puolidynaamisilla SVG-kuvilla. Dynaamisuudella tarkoitettiin sitä, että SVG-kuva voitiin ladata ns. inline-muodossa. Inline- ja normaali kuvamuoto erosivat lataustavassaan. Normaalit kuvat, kuten SVG- ja PNG-tyyppiset kuvat, ladattiin ennalta määritetystä ”Images”-hakemistosta. Inline-kuva taas määriteltiin kokonaan ennakkoon tai ajon aikana koodirivillä tekstinä. Puolidynaamisuudella inline-kuvassa tarkoitettiin, että sen osia pystyttiin muokkaamaan koodilla, vaikka SVG-kuva säilytti silti suuremman osan alkuperäisestä muodostaan. Käytännössä tämä tarkoitti, että koodissa datan perusteella voitiin luoda tilakoneita, jotka esimerkiksi vaihtoivat väriä tilanteen mukaan. Datan avulla voitiin vaihtaa reunan, tai sisällön värejä, laatikon tekstiä ja logiikkaportin tilaa (auki, kiinni) (Kuva 24).

Kuva 24. Puolidynaaminen SVG-kuva.



Jos tapahtuman tilaa haluttiin merkitä tällä hetkellä tapahtuvaksi, pystyttiin tyylien avulla asettamaan väreille häivytys animaatiot. Funktiossa (Kuva 25) asetettiin läpinäkyvyys ominaisuutta pienemmälle ja häivytettiin näin nollan ja sadan prosentin välillä. Tämä loi sekunti perusteisen vilkkumisvärin.

Kuva 25. CSS-animaatio.

```
animation: hideshow 2s ease-in-out infinite;


@keyframes hideshow {
    0%, 100% {
        opacity: 1;
    }

    50% {
        opacity: 0.3;
    }

}
```

6 Johtopäätökset ja pohdinta

Opinnäytetyössä käsiteltiin käyttöliittymän ohjelmiston suunnittelua ja toteutusta. Pääasiallisena tavoitteena työssä oli kartoittaa projektia ja toteuttaa prototyyppiversiota eteenpäin. Opinnäytetyössä tuli ilmi kuinka erinäköisiä HTML- ja JavaScript-pohjaisia komponentteja voidaan käsitellä .NET-ympäristössä, ja kuinka SQL-tietokantaa voidaan käyttää ja hyödyntää osana projektia. Opinnäytetyössä käsiteltiin myös tarkemmin JavaScriptillä tehtyä paikannuskoordinaatistoa, sillä tämä oli iso osa virallisen prototyyppiversion sivujen kokonaisuudesta. Jotta opinnäytetyön osalta pystyttäisiin paremmin pysymään aikataulussa, päätettiin logiikkaohjaimen ja tietokannan välinen kommunikaatio rajata tarkoituksella pois opinnäytetyön toteutusosuudesta. Lopputuloksena opinnäytetyöstä saatiin toivottu skaalautuva web-pohjainen järjestelmä, jonka ulkoasu vastasi edellisen version tyyliä.

Kokonaisuudessaan projektissa oli paljon tekemistä, koska melkein kaikki sivun käännöksestä uuteen tekniikkaan piti tehdä alusta asti ”käsin”. Prototyypin toteutuksessa piti myös ottaa

huomioon se, että prototyypin tulisi olla olemukseltaan hyvin samankaltainen kuin sitä edeltävä versio, jotta vanhan käyttäjän ei tarvitsisi opetella koko käyttöliittymää uusiksi.

Kun mietittiin yleisesti uusia ja vanhoja käyttäjiä, katsottiin tärkeäksi, että käyttöliittymästä löytyisi kaikki kohdat intuitiivisesti. Tämän aikaansaamiseksi piti miettiä uusia tapoja toteuttaa nyt web-pohjainen käyttöliittymä. Uudessa ratkaisussa piti muistaa aina geneerisyys ja dynaamisuus, joka teki välillä helponkin komponentin tai asian yllättävän monimutkaiseksi ja aikaa syöväksi työvaiheeksi. Hyvin suunniteltu dynaamisuus elementteihin ja koodiin tosin loi monia muita etuja alun kangertelujen jälkeen. Aikaisemmin esimerkiksi tapahtumissa saattoi olla vain yhdestä neljään ennalta määritettyä vaihtoehtoa, mutta uudessa tämä saattoi olla vaikka 50, jos tarve sitä vaati. Yhteensä projektissa nähtiin noin 17 kappaletta uniikkeja sivuja. Erillisiä uniikkeja sivuja prototyyppiin tuli kahdelle pääsivulle $(x) * 6$, jossa x oli bunkkerien määrä, monitorointi sivuille $(y) * 7 + 4$, jossa y oli nostureiden määrä. Näiden lisäksi prototyyppiin tuli muita sekalaisia sivuja, kuten asetukset-sivu, jotka eivät tulleet kokonaisuudessaan valmiiksi. Vaikka projektin prototyyppiversiossa eivät kaikki yksityiskohdat olleet täydellisiä opinnäytetyön ollessa lopussa, tuli prototyypistä käyttöliittymän ulkoasun kannalta kaiken kaikkiaan lähes valmis. Itse prototyypin kehityksessä tämä olisi ollut seuraava vaihe. Projektista pystyttiin nyt jatkamaan ja viimeistelemään muita mahdollisia keskeneräisiä kohtia tarpeen tullen.

Lähteet

Bootstrap team. (n.d.). *About bootstrap.* Haettu 1.11.2021 osoitteesta https://getbootstrap.com/

Bryla, B. (2004). *Oracle Database Foundations: Technology Fundamentals for IT Success.* (1. p.). John Wiley & Sons, Incorporated.

Flanagan, D. (2000). *JavaScript- tehokäyttäjän opas* (P. Matilainen, käänt.). Suomen Atk-kustannus Oy. (Alkuperäisteos julkaistu 1997)

Haverbeke, M. (2014). *Eloquent JavaScript, 2nd Edition.* No Starch Press

Hovi, A. (1998). *SQL-Ohjelmointi Pro-kurssi.* (4. p.). Suomen ATK-kustannus Oy

Hovi, A., Huotari, J. & Lahdenmäki, T. (2005). *Tietokantojen suunnittelu & indeksointi.* (1. p.). Docendo Finland Oy

Microsoft. (n.d. -a). *ASP.NET MVC Pattern*. Haettu 1.11.2021 osoitteesta https://dotnet.microsoft.com/apps/aspnet/mvc

Microsoft. (n.d. -b). *SqlCommand Class*. Haettu 23.11.2021 osoitteesta https://docs.microsoft.com/en-us/dotnet/api/system.data.sqlclient.sqlcommand?view=dotnet-plat-ext-6.0

Microsoft. (2021.-a). *Introduction to LINQ Queries*. Haettu 24.11.2021 osoitteesta

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/introduction-to-linq-queries

Microsoft. (2021. -b). *ASP.NET Core MVC*. Haettu 7.12.2021 osoitteesta

https://docs.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-6.0

Microsoft. (2021 -c). *A tour of the C# language*. Haettu 17.1.2022 osoitteesta

https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/

Microsoft. (2021.-d). *LINQ*. Haettu 24.11.2021 osoitteesta

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/

Mozilla. (n.d). *Introduction to the server side*. Haettu 1.11.2021 osoitteesta
https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Introduction

Mozilla. (2021) *Object prototypes*. Haettu 17.11.2021 osoitteesta
https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes

Pattiengineering. (n.d). *The Father of the PLC – Dick Morley*. Pattiengineering.

https://pattiengineering.com/blog/father-plc-dick-morley/

Smith, S., JENDOUBI, M., Anderson, R., & Sauber, S. (2021). *Partial views in ASP.NET Core*. Haettu 1.11.2021 osoitteesta https://docs.microsoft.com/en-gb/aspnet/core/mvc/views/partial?view=aspnetcore-5.0

Stefanov, S. (2008). *Pact Object-oriented JavaScript : create scalable, reusable high-quality JavaScript applications and libraries.* Packt Publishing

Unitronicsplc. (n.d). *What is PLC*. Unitronicsplc.

https://www.unitronicsplc.com/what-is-plc-programmable-logic-controller/

Wayand, B. (2020). *What is a PLC?*. Mroelectric.

https://www.mroelectric.com/blog/what-is-a-plc/