RIIHI DMA -TILAUSPORTAALI

Ammattikorkeakoulun opinnäytetyö

Tieto- ja viestintätekniikka, insinööri (AMK)

kevät 2022

Jaakko Jääskeläinen

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Tieto- ja viestintätekniikan koulutusohjelma
Tiivistelmä

Tekijä Jaakko Jääskeläinen
Vuosi 2022

Työn nimi Riihi DMA -tilausportaali

Ohjaaja Petri Kuittinen

Opinnäytetyön päätavoitteena oli luoda asiakkaan ja työntekijän kannalta tehokas ratkaisu aloittaa asiakkuus päätelaitehallintapalvelun kanssa. Kyseiseen palveluun on mahdollista valita useampia järjestelmäkohtaisia asetuksia ja ylläpidettäviä sovelluksia, jotka räätälöidään asiakkaan tarpeiden mukaan.

Toimeksiantajana toimi IT-alan yritys, Riihisoft Oy. Ratkaisun tarve syntyi asiakasmäärien kasvaessa. Vanha järjestelmä vaati liian monta manuaalista ja toistuvaa työvaihetta asiakkaan ja työntekijän kannalta, eikä täten myöskään antanut vaikutelmaa pätevästä alan toimijasta.

Ratkaisun tehokkuuden mittarina käytetään aikaa, mikä tilaajan ja työntekijän osilta säästyy, vanhaan järjestelmään verrattuna. Palvelun kustannukset on myös suhteutettu uuden prosessin säästämään aikaan.

Palvelu toteutettiin Azure-pilvialustalla ajettuna web-sovelluksena, johon käyttäjille lisättiin tunnukset. Sovellus mahdollisti käyttäjille aina ajantasalla olevan katalogin valittavista asetuksista asiakasympäristöillensä. Tilaukset tallentuivat tietokantaan, mistä niitä on helppo selata ja näin asiakkaat sekä työntekijät pysyivät perillä sovituista asetuksista. Ratkaisu nopeutti myös tilausten käsittelyä, sillä määrätyssä formaatissa tallentuneet tilaukset mahdollistivat prosessien automatisoinnin. Lopuksi arvioidaan jatkokehitysmahdollisuudet ja skaalautuvuus käyttäjämäärien kasvaessa.

Avainsanat Azure, pilvilaskenta, web-sovellus, ohjelmointi, ohjelmistokehys
Sivut 56 sivua ja liitteitä 1 sivua

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Information and communication technology | Abstract
Author Jaakko Jääskeläinen | Year 2022
Subject Riihi DMA order portal
Supervisors Petri Kuittinen

The goal of this thesis was to create, from the perspective of the client and the worker, an effective solution to begin a customership with a device management service. The service in question is configurable with multiple system-specific settings and applications that are tailored to suit the client's needs.

An IT business, called Riihisoft Oy, acts as the contractor. The need for this solution came to be as client masses expanded. The old system required too many manual and repetitive tasks from the client and the worker, thus it did not give an impression of a competent company.

To measure the efficiency of the solution, time saved by the client and the worker is compared to the old system. The cost-effectiveness is also measured by comparing the added cost that the new system accumulates to the time saved.

The solution was carried out as a web application running on Azure for which the users were provided credentials. The application made it possible to serve its users the most up-to-date catalog of configurations for their client environments.The generated orders were saved into a database where they were easily accessible while also keeping the client and the worker on the same page regarding the agreed upon configurations. The solution also hastened order processing because every order was now in the same format which in turn made it possible to automate the process further. Lastly, further development and scaling of the service will be evaluated as the userbase increases.

Keywords Azure, cloud computing, web application, programming, framework
Pages 56 pages and appendices 1 pages

Sisällys

1 Johdanto 1

2 Riihi DMA -palvelun käyttöönotto 1

2.1 Prosessin kehityspolku 2

2.2 Web-sovellus 2

3 Teknologioiden ja arkkitehtuurin määrittelyt 2

3.1 Omat fyysiset palvelimet ja pilvi 3

3.1.1 Fyysiset palvelimet 3

3.1.2 Pilvipalvelut 4

3.1.3 Vaihtoehtojen vertailua 4

3.2 Web-sovelluksen arkkitehtuuri 5

3.2.1 Käyttäjärajapinta (front end) 6

3.2.2 Toimintalogiikka (backend) 6

3.2.3 Tietovarasto (storage) 6

3.3 Ohjelmistokehykset, kirjastot ja kielet 7

3.3.1 Ohjelmistokehykset 7

3.3.2 Sisällönhallintajärjestelmät 12

3.3.3 Kirjastot 15

3.3.4 Kielet 15

3.4 Työkalut ja kehittimet 20

3.4.1 Azure DevOps ja Git 21

3.4.2 Visual Studio 22

3.4.3 Nuget 22

3.4.4 MS SQL Server & SSMS 23

3.4.5 Adobe XD 24

4 Riihi DMA tilausportaali 25

4.1 SQL Server 26

4.2 ASP.NET ja Umbraco 27

4.2.1 ASP.NET ohjelmistokehyksenä 27

4.2.2 Umbraco 28

4.2.3 Palvelinsovelluksen rakenne 31

4.3 Käyttäjärajapinta 35
4.4 Sovelluksen suojaaminen 41
4.4.1 Tietoturva tasoina 41
4.4.2 Sovelluskehitys ja tietoturva 42
4.4.3 Authentikaatio 43
4.4.4 Tietokanta 44
4.5 Azure 45
4.6 Azure App Service 47
4.7 Azure SQL Server 48
4.8 Portaalin julkaisu Azureen ja käyttöönotto 49
5 Projektin tulokset 50
5.1 Asiakkaiden näkökulma 51
5.2 Ylläpidon helpottaminen 52
5.3 Jatkokehitys 53
5.4 Henkilökohtaiset tavoitteet 54
Lähteet 55

Liitteet

Liite 1 Riihi DMA -asiakaslomake

1 Johdanto

Tämän opinnäytetyön tavoitteena on luoda toimeksiantajalle järjestelmä, mikä korvaa vanhan asiakkuuksien aloitusprosessin, päätelaitehallintapalvelun käyttöönottoon. Riihi DMA on Riihisoft Oy:n tuottama palvelu, jolla asiakkaat voivat yksinkertaistaa yrityksensä päätelaitteiden hallinnan. Palvelu koostuu, Microsoftin Intunessa hallittavista, räätälöidyistä ohjelmisto- ja asetuspaketeista, joita Riihisoft Oy ylläpitää. Riihisoft Oy:n ensisijainen liiketoiminta on ohjelmistokonsultointi ja Riihi DMA on erillinen tuote omalla kokoonpanollaan. Ratkaisun kannalta on olennaista saada järjestelmä, jota on helppo käyttää, ylläpitää ja sen tulee tulevaisuudessa automatisoida mahdollisimman monta toistuvaa työvaihetta. Työssä käsitellään vanha järjestelmä, vertaillaan vaihtoehtoja ratkaisun suhteen, dokumentoidaan toteutuksen eri vaiheet ja lopuksi arvioidaan toteutetun ratkaisun hyödyllisyys.

Henkilökohtaisena tavoitteena on saada syvempi käsitys moderneista web-teknologioista, sovellusarkkitehtuurista ja tehokkaista kehitysprosesseista. Toinen tavoite on mahdollisuus vakituiseen työsuhteeseen joko Riihi DMA -tuotteen piirissä tai Riihisoft Oy:n ohjelmistokonsultoinnissa.

2 Riihi DMA -palvelun käyttöönotto

Alkuperäisessä järjestelmässä asiakkaalle lähetetään sähköpostilla listanomainen PDF-lomake, jonka he palauttavat valintojensa kanssa (Liite 1). Asiakkaiden tuli päivystää sähköpostiaan, jotta he löytäisivät tilattavat tuotteet ja työntekijän täytyi pitää kirjaa tilauksista eri sähköpostiketjuissa. Vanha järjestelmä vaati myös työntekijältä paljon tarkkaavaisuutta, sillä valittavien tuotteiden katalogi kasvoi jatkuvasti ja lomakkeiden piti olla ajan tasalla. Käsin tehtävät tarkistukset ja asiakkaan, mahdollisesti epäselvät, toiveet tekivät koko prosessista erittäin virhealttiin, joten ratkaisun tulisi yksinkertaistaa tilausten tekoa kokonaisvaltaisesti.

2.1 Prosessin kehityspolku

Lähdin tilausten käsittelijöiden kanssa selvittämään kehittämisen kohteita tutkimalla edellä mainittuja pullonkauloja. Käytännössä kaikki palvelun käyttöönottoon liittyvä tapahtui käsin, sillä työntekijät olivat täysin työllistettyinä asiakkaiden parissa, joten prosessien kehittämiseen ei ollut ihmisresursseja. Tuotelistojen päivitys oli verkkolevyllä oleva jaettu tiedosto, mistä tilausvastaava kasasi sähköpostin asiakkaalle. Asiakkuudet oli jäsennelty omiin sähköpostikansioihin, joita käytiin päivystämässä. Iso osa työajasta menikin kaikkeen muuhun kuin varsinaiseen tuotteen ydintoimintaan.

2.2 Web-sovellus

Ottaen huomioon Riihisoft Oy:n ensisijaisen liiketoiminnan päätimme yhtiötä edustavan ohjaajani kanssa, että henkilökohtaisen kehityspolkuni kannalta olisi viisasta rakentaa web-sovellus tilausten käsittelyyn. Tämän takia en tule käymään läpi kilpailevia vaihtoehtoja jo valmiista tilausportaaliratkaisuista tai muista kuin web-pohjaisista lähestymistavoista.

Web-sovelluksella tarkoitetaan selaimessa toimivaa, julkisen verkon kautta liikennöivää, palvelua. Toteutuksen sovellus olisi käytännössä sivusto, johon käyttäjä kirjautuu sisään tekemään tilauksia asiakasympäristöihinsä ja tarkastellakseen aikaisempia tilauksiaan.

Laskuttaminen tapahtuu täysin sovelluksen ulkopuolella, joten mitään maksupalveluihin liittyvää ei tarvitse suunnitella tai toteuttaa sovellukseen. Katalogin tuotteet ovat muutenkin suurimmilta osin palveluun kuuluvia ja sopimuskohtaisia, joten niiden mahdollinen hinnoittelu ei sisälly projektin laajuuteen.

3 Teknologioiden ja arkkitehtuurin määrittelyt

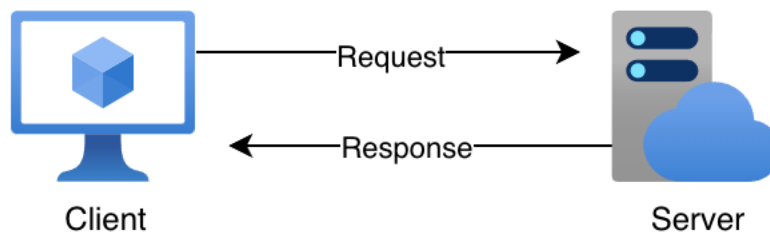
Kehittämisen aloittamista edeltävässä työvaiheessa lähdin kartoittamaan teknologiset vaihtoehdot ja arkkitehtuurin kannalta olennaiset komponentit. Varsinaisen toteutuksen ja teknologioiden määrittelyyn vaikuttivat aikaisempi osaamiseni, kilpailevien teknologioiden vertailu ja Riihisoft Oy:n linjaukset. Ohjesääntönä oli suunnitella mahdollisimman kevyt ja

helposti ylläpidettävä kokonaisuus, jota voidaan tarvittaessa täydentää uusilla ominaisuuksilla. Riihisoft Oy:n ollessa Microsoft kumppani, teknologiavalinnat tulisi ensisijaisesti hakea Microsoftin ekosysteemistä.

3.1 Omat fyysiset palvelimet ja pilvi

Web-sovellukset vaativat minimissään yhden palvelimen, joka tarjoilee sisällön käyttäjälle selaimeen. Ottamatta kantaa käyttöjärjestelmiin ja komponentteihin, palvelimien suhteen vaihtoehtoja on käytännössä oma fyysinen palvelin ja pilvi.

Kuva 1 Palvelin - asiakas kommunikaatio



3.1.1 Fyysiset palvelimet

Perinteinen tapa pystyttää ja ylläpitää palveluita oli yrityksen tiloissa oleva fyysinen palvelintietokone, josta yhtiö vastasi omilla resursseillaan koko sen elinkaaren. Pilvipalveluita ei aina ollut saatavilla, joten omat palvelimet olivatkin ainoa keino saada palveluita ylipäätään aikaiseksi.

3.1.2 Pilvipalvelut

Pilvipalvelut ovat puolestaan ulkoistettu palvelu, jossa vastuu ylläpidosta on määriteltävissä eri tasoissa. Kaikki palvelimien fyysiseen ylläpitoon liittyvä on aina palveluntarjoajan vastuulla.

Tietojen palautus katastrofitilanteissa on myös pilvessä usein sisäytetty jo perustason palveluihin. Tämän laajuutta voidaan myös määritellä erikseen, palvelun kriittisyyden tai vaikka budjetin mukaan.

3.1.3 Vaihtoehtojen vertailua

Pilvipalveluiden hyödyt ovat, päällepäin katsottuna, mittavat verrattuna muihin ratkaisuihin. Palvelujen tasoja voidaan määritellä pelkän infrastruktuurin ylläpidosta (IaaS) täysin ylläpidettyihin palveluihin (SaaS). Pilvipalveluissa ei myöskään ole alkupään kustannuksia palveluiden pystyttämiseen vaativasta laitteistosta tai lisensseistä. Resursseihin pääsyyn ei myöskään vaadita fyysisiä tiloja, jotka olisivat yhteydessä yhtiön verkkoon. Jo näistä johtuen pilvipalvelut ovat nykyään nopeiten kasvavin tapa uusien palveluiden pystyttämiseen. (Eurostat, 2021)

Pilvipalvelun käyttämättä jättämisen valinta on selkeä silloin, kun fyysinen pääsy laitteistokokoonpanoon ja verkkoinfrastruktuuriin on välttämätöntä. Pilvipalveluntarjoajat harvemmin antavat tähän mahdollisuutta. Tietoturvaan liittyvät linjaukset saattavat myös rajata pilvipalvelut kokonaan pois vaihtoehdoista, joten omat palvelimet on tällöin ainoa tapa toteuttaa palveluita.

Alla havainnollistava taulukko Microsoftin pilvipalveluiden eri tasoista verrattuna toisiinsa ja fyysisiin palvelimiin, vastuun näkökulmasta. (Taulukko 1)

Taulukko 1 Vastuunjako pilvipalveluissa (Microsoft Corporation, 2021, s. 216)

Responsibility	On-prem	IaaS	PaaS	SaaS
Data governance & rights management	Customer	Customer	Customer	Customer
Client endpoints	Customer	Customer	Customer	Customer
Account & access management	Customer	Customer	Customer	Customer
Identity & directory infrastructure	Customer	Customer	Microsoft / Customer	Microsoft / Customer
Application	Customer	Customer	Microsoft / Customer	Microsoft
Network controls	Customer	Customer	Microsoft / Customer	Microsoft
Operating system	Customer	Customer	Microsoft	Microsoft
Physical hosts	Customer	Microsoft	Microsoft	Microsoft
Physical network	Customer	Microsoft	Microsoft	Microsoft
Physical datacenter	Customer	Microsoft	Microsoft	Microsoft

Microsoft Customer

Nämä vaihtoehdot eivät kuitenkaan poissulje toisiaan, vaan on olemassa myös hybridiratkaisuja, missä osa laitteistosta on edelleen fyysisesti yrityksen tiloissa, mutta yhteydet sillataan pilveen. Yksityinen pilvi, eli käytännössä itsepystytettävä pilvipalvelu, on sitä varten, kun halutaan hyödyntää pilvialustan tarjoamia ominaisuuksia ja silti pitää omien palvelimien tarjoamat hyödyt.

Toimeksiantoon liittyvät vaatimukset ohjasivat, ilman pidempää pohdintaa, pilviratkaisuun. Palveluntarjoajista otimme käyttöön Microsoftin Azuren, sillä Riihisoft Oy:llä oli jo Azure-ympäristö provisioituna.

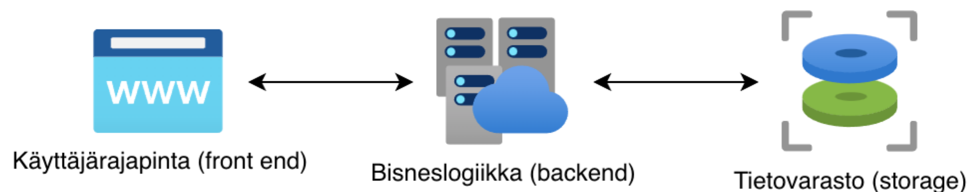
3.2 Web-sovelluksen arkkitehtuuri

Sovelluksien arkkitehtuuraalisiin kokoonpanoihin liittyy usein kiivastakin keskustelua eri koulukuntien kanssa, joten päätöksenteossa on hyvä selvittää etukäteen kaikki sovelluksen vaatimat ominaisuudet ja päättää niiden pohjalta paras lähestymistapa. Pyrimme mahdollisimman modulaariseen ratkaisuun, jotta tulevaisuudessa jatkokehittäminen olisi vaivatonta. Monoliittinen ratkaisu olisi helpoin ja nopein toteuttaa, mutta se tulisi hidastamaan kehitystä.

Tilausportaalin toiminnallisuuteen liittyy kolme tasoa, mitkä on jaettu toimialojen mukaan: käyttäjärajapinta, toimintalogiikka ja datan varastointi. (Martin, 2018, s.205) Tasojen välinen abstraktio mahdollistaa sovelluksen eri osien kehittämisen vaikuttamatta toisiin tasoihin.

Ohessa havainnollistava kuva sovelluksen korkean tason arkkitehtuurista (Kuva 2).

Kuva 2 Sovelluksen korkean tason arkkitehtuuri



3.2.1 Käyttäjärajapinta (front end)

Käyttäjärajapinnan ensisijainen toteutus on selaimessa ajettava interaktiivinen sivusto, josta käyttäjä voi hallita kaikkia hänelle suotuja toimintoja. Toissijainen toteutus on mahdollinen integraatiorajapinta, jolla tilauksien tietoja voidaan hyödyntää toisissa sovelluksissa.

3.2.2 Toimintalogiikka (backend)

Toimintalogiikka on tilausten käsittelyä varten koteloitu kokonaisuus, mikä ajetaan palvelimen puolella. Tämä taso toimii välikätenä tietojen varastoinnin ja käyttäjän näkemien tietojen kanssa.

3.2.3 Tietovarasto (storage)

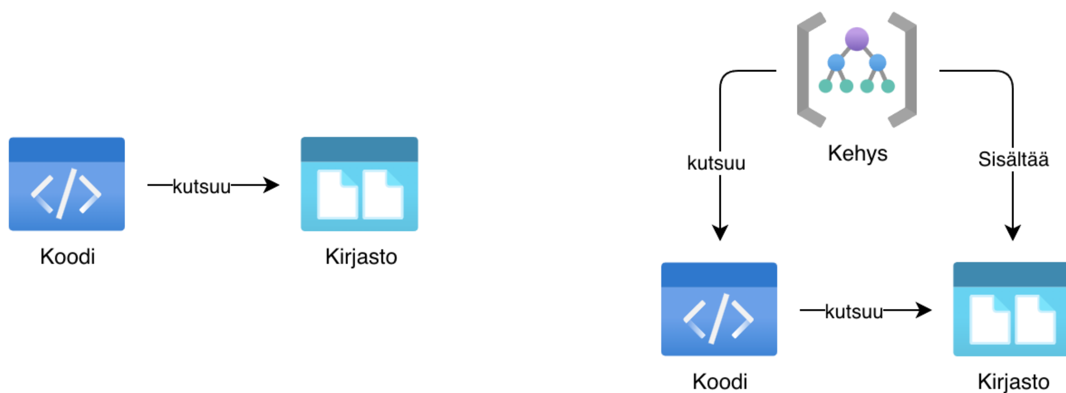
Tietovaraston vastuualueella on pysyvä säilyvyys ja mahdolliset relaatiot tietojen välillä. Tietovarasto voi sisältää jonkin verran logiikkaa liittyen tiedon tyyppiin ja mahdollisiin

kenttiin, eli skeemaan. Siellä voi usein myös olla tiedon eheyteen liittyvää validaatiologiikkaa. Tämä taso voi yksinkertaisimmillaan olla vaikka vain yksittäinen tiedosto kovalevyllä, jota toimintalogiikka päivittää.

3.3 Ohjelmistokehykset, kirjastot ja kielet

Kehittämisen avuksi on olemassa valmiita koodikokonaisuuksia, eli kirjastoja ja ohjelmistokehyksiä. Ne on luotu ratkaisemaan usein kohdattuja kehittämisen haasteita, jotta kehittäjien ei tarvitsisi tehdä kaikkea uudestaan joka projektissa. Kehykset ja kirjastot ovat lähellä toisiaan siinä mielessä, että molemmat tarjoavat kehittäjälle rajapinnan, minkä ympärille rakentaa toiminnallisuutta. Keskeinen ero kehysten ja kirjastojen välillä on hierarkia. Kun kehittäjän koodi kutsuu metodia kirjastosta, koodilla on ensisijainen hallinta. Kehysten kanssa tämä on käänteinen, eli kehys kutsuu kehittäjän koodia. Ohjelmistokehykset usein myös tarvitsevat kirjastoja toimintojensa tueksi, joten ne ja koodi kutsuvat myös kirjastojen rajapintoja.

Kuva 3 Kirjastojen ja kehysten hierarkia



3.3.1 Ohjelmistokehykset

Web-kehityksessä on syytä hyödyntää mahdollisimman paljon yleisesti käytettyjä standardeja, jotta toiminnallisuus eri selaimissa ja palveluiden välisessä kommunikaatiossa

olisi johdonmukaista. Toimeksiannon tilausportaali ei ole mitenkään uusi, tai erikoinen konsepti, joten sen rakentamista varten on jo olemassa monta eri kehystä, mistä valita. Hyvä kehys vähentää toistuvan koodin kirjoittamista ja kehittäjä voi näin keskittyä varsinaiseen toimintalogiikkaan. Kehysten kanssa työskentely ohjaa tietynlaisen koodin tuottamiseen, sillä kehys on nimensä mukaan koodin ympärillä. Tästä poikkeaminen aiheuttaa usein enemmän koodia, kuin ilman kehystä kehittäminen. On siis selvitettävä aina projektikohtaisesti, että milloin kannattaa ottaa kehys käyttöön ja milloin se tuottaa vaan tarpeetonta monimutkaisuutta.

Koska linjasimme aiemmin käyttäjärajapinnan ja palvelinpuolen logiikan erillisinä komponentteina, on nyt myös mahdollista käyttää eri kehystä kumpaankin toteutukseen.

Vertailussa huomioin lähtökohtaisesti suosituimmat kehykset, sillä ne ovat todennäköisesti parhaiten dokumentoituja, ylläpidettyjä ja toimintorikkaita. Kehysten kielirajapinnat ja Riihisoft Oy:n linjaukset vaikuttivat myös valintaani.

Palvelinpuolen toimintalogiikan tueksi valitsin Microsoftin Common Language Runtimen (CLR) päälle rakennetun ASP.NET-kehyksen. ASP.NET on avoimeen lähdekoodiin perustuva, Apache-lisenssin alla toimiva, ilmainen web-ohjelmistokehys. Se on suunniteltu tuottamaan dynaamisia web-sivuja, -sovelluksia ja -palveluita. ASP.NET koteloi http-liikennöinnin reitittämisen ja sisällön tarjoilun, jotta kehittäjän ei itse tarvitse kantaa huolta standardien sisällä pysymisestä. Valinta rajaa palvelinpuolen käyttöjärjestelmän Windows Serveriin ja http-palvelimen IIS-palvelinohjelmistoon, mitkä olivat suoraan Riihisoft Oy:n linjausten puitteissa. Vaihtoehtoina olisivat olleet Pythonin tai Javan päälle rakennetut kehykset.

Taulukko 2 Web-ohjelmistokehysten käyttö ja kasvu 2020 - 2021

Kehys	Kieli	Kehittäjä	Palvelin	Lähdekoodi	Arkkitehtuuri	Käyttö ja kasvu (2020-2021) (Stack Overflow, 2020; Stack Overflow, 2021)
ASP.NET	C#/VB	Microsoft	IIS	Avoin	MVC	15,74% (-28,1%)
Spring	Java	VMware	Tomcat	Avoin	MVC	14,56% (-11,2%)
DJango	Python	PSF	Nginx/ Apache	Avoin	MVT	14,99% (+5,6%)

Lähteenä on Stack Overflow Insights, minkä vuosittaisiin teknologiakyselyihin vastaavat kymmenet tuhannet sivuston käyttäjät.

Kyselyn vastaajat ovat voineet ilmoittaneen käyttäneensä useampaa kehystä, joten prosentit eivät ole toisiaan poissulkevia.

Käytön seurannassa kannattaa ottaa huomioon kasvu suhteessa markkinaosuuteen, sillä vanhemmat kehykset saattavat olla pidemmänkin aikaa käytössä, mutta menettävät markkinaosuuttaan joka vuosi. Kilpailukelpoisen aseman menetys voi johtaa projektin hylkäämiseen, joten äkillisesti romahtavaa kehystä ei kannata ainakaan uuteen projektiin välttämättä valita ja vanhemmankin projektin ylläpidon on syytä ottaa askeleita pahimman varalle.

Taulukon perusteella suosiota on vaikea vertailla, mutta ylipäätään kehysten mainita noin isossa kyselyssä tarkoittaa, että ne ovat mahdollisesti laajalti käytössä. Kehysten

ilmestyminen uusina niminä taulussa voidaan myös tulkita niin, että niiden käyttöaste on kasvanut.

Djangoa lukuunottamatta, kaikkien vertailun kehysten ollessa MVC-kehyksiä, varsinaiset erot ovatkin suurimmilta osin kehityskielissä ja ajoympäristöissä. .NET-kielten ollessa jo entuudestaan tuttuja ja koska ASP.NET on Microsoftin tuote, syitä poiketa ASP.NET:stä ei oikeastaan ollut.

MVC on yksi oliopohjaisen ohjelmistokehityksen lähestymistapa. Sovelluksen toiminta on jaettu MVC-mallissa kolmeen abstraktioon: data, eli malli (model), käyttäjärajapinta, eli näkymä (view) ja toimintalogiikka, eli käsittelijä (controller). Kutsut ohjataan käsittelijälle, joka juttelee mallin kanssa ja palauttaa lopuksi näkymän.

MVT on pitkälti samanlainen MVC-arkkitehtuurin kanssa. MVT:n kaksi tasoakin ovat saman nimisiä. Erot ovat siinä, että MVT:n kehys vastaa käsittelijöistä, kun taas MVC:ssä käsittelijän kirjoittaminen on kehittäjän vastuulla. MVT:n näkymä on vastuussa HTTP-viestien liikenteestä ja sapluuna (template) ajaa saman roolin, kuin MVC:n näkymä. Tämä saattaa vaikuttaa sekavalta, koska toimintahan on käytännössä sama, mutta termit menevät ristiin. Kyseessä on kuitenkin Djangon kehittäjien tulkintaan pohjautuva näkemys asiasta (Django Software Foundation, 2022).

Käyttäjärajapinnan suunnittelussa päätin ottaa kokoonpanoksi täysin selaimessa ajettavan ratkaisun. Näin sovelluksen toimintalogiikka on kokonaan eritelty ajon näkökulmastakin palvelimella ja asiakkaan laitteella ajettavaan koodiin. Ajoympäristön määrittely tällä tavalla rajaa käyttäjärajapinnan kehykset näin ollen JavaScript-pohjaisiin teknologioihin. Muitakin vaihtoehtoja on JavaScriptin tilalle, kuten WebAssembly ja VBScript, mutta ne ovat aika monimutkaisia ratkaisuja näin yksinkertaiseen sovellukseen, sekä niiden yhteensopivuus kaikilla selaimilla ei ole varmaa.

Kilpailu JavaScript-kehysten välillä on todella kiivas ja siksi uusia ilmestyy hyvin usein. Tässä trendissä on puolensa, sillä kilpailu ajaa aina kehitystä, mutta samalla projekteja hylätään ennätysvauhtia. Päädyin valitsemaan Googlen kehittämän AngularJS-kehyksen, sillä se kuului Riihisoft Oy:n linjauksissa määritellyn sisällönhallintajärjestelmän kehityskieliin. Katsoimme,

että siihen tutustuminen edistäisi minun osaamistani konsulttipuolella parhaiten. Hyvinä vaihtoehtoina oli Vue.js ja React.js, jotka molemmat tarjosivat samanlaisia ominaisuuksia tiedon elementteihin sitomisen suhteen ja olivat tilastoissa hyvin edustettuina.

Taulukko 3 Web-ohjelmistokehysten käyttö ja kasvu 2020 - 2021

Kehys	Latauskoko (kt)	Kehittäjä	Käyttö ja kasvu (2020 - 2021) (Stack Overflow, 2020; Stack Overflow, 2021)	Käyttö ja kasvu (2020 - 2021) (Greif, 2021)
Vue.js	29,3 (v2.5.16)	Vue Core Team	18,97 (+9,7%)	51% (+4%)
React.js	11,8 (v16.7.0)	Facebook	40,14% (+11,8%)	80% (+0%)
Angular	172 (v1.7.2)	Google	22,96% (-8,5%)	54% (-4%)

AngularJS:n osalta tulee myös huomioida, että Angularista on tullut uusi versio, Angular2, mikä ei ole sama kehys, mutta luvut ovat molemmissa otannoissa yhdistettynä. Tämän takia, vaikka käyttö näyttääkin isolta, se ei todellisuudessa todennäköisesti ole.

Ensimmäinen lähteistäni on sama, kuin palvelinpuolen kehysten vertailussa, eli Stack Overflow. Käytin, vertailun vuoksi, stateofjs.com -sivustoa toisena lähteenä. Heidän statistiikkansa perustuu myös kyselyyn, jossa on noin 20 000 vastaajaa eri puolilta maailmaa. Kyselyn vertailussa oli ainoastaan JavaScript-kehyksiä toisin kuin Stack Overflow:n, missä oli kokonaisvaltaisemmin eri teknologioita.

Aitoa käyttöastetta on vaikea määritellä, sillä siihen ei ole mitään, yksimielisesti hyväksyttyä, keskitettyä tietokantaa ja jokainen sivusto saattaa käyttää erilaisia tapoja laskea käyttöä.

Tämä heijastuu hyvin lähteiden tulosten välisessä erossa. Lähdekoodien latausmäärien vertailu on myös yksi tapa verrata kehyksiä, mutta paketinhallintaohjelmiakin on monia ja niidenkään luvut eivät välttämättä ole kerro koko totuutta. Pakettia voidaan ladata samaan projektiin useita satoja kertoja, mikä paisuttaisi lukuja.

Vertailun JavaScript-kehysten erot ovat projektin kannalta jokseenkin marginaalisia, sillä suurimmat eroavaisuudet ovat, toiminnallisuuksien kutsumismetodeja lukuunottamatta, konepellin alla olevassa koodissa, mihin kehittäjän ei ole tarkoitus kajota. Optimoinnin kannalta pakattu latauskoko on ehkä tärkein mittari, sillä se vaikuttaa sivulatauksen kestoon. Tämäkin mittari menettää merkityksensä ensimmäisen latauksen jälkeen, sillä selain varastoi kirjastoja välimuistiinsa, joten uudelleenlataukset tapahtuvat nopeasti.

Tuloksista voidaan olettaa, että AngularJS menettää suosiotaan. Se täytyy tulevaisuudessa ottaa huomioon, isompia muutoksia suunnitellessamme sovellukseen. Meidän tulee harkita toisen kehyksen käyttöönottoa, jos AngularJS:n käyttöaste jatkaa laskuaan ja kehitys hidastuu huolestuttavasti. En pidä Riihisoft Oy:n linjausta projektiin siltikään huonona, sillä Umbraco ei ole ilmoittanut kiinnostusta backoffice-kehyksensä vaihtoon.

AngularJS:n lisäksi valitsin Bootstrapin helpottamaan sovelluksen käyttöliittymän rakentelua. Bootstrap on CSS, JavaScript ja HTML -kirjastokokonaisuus, joka määrittelee itsensä kuitenkin ohjelmistokehykseksi sivuillaan. Se on luotu yksinkertaistamaan ja standardisoimaan web-sivujen ulkoasujen kehittämistä ja helpottamaan responsiivisuuden saavuttamista. Responsiivisuudella tarkoitetaan sivuston toimivuutta eri resoluutioisilla selaimilla. Bootstrap sisältää hyvin dokumentoidun ja laajan kirjon valmiita, muokattavia, malleja, joita yhdistelemällä sivujen rakentelu pysyy johdonmukaisena.

3.3.2 Sisällönhallintajärjestelmät

CMS (Content Management System), eli sisällönhallintajärjestelmä, on alusta sisällön tuottamiseen, hallintaan ja julkaisuun. Se koostuu yleensä kahdesta erillisestä komponentista: Sisällönhallintasovellus (CMA), mikä toimii käyttäjärajapintana varsinaisen sisällön tuottamiseen ja sisällönjulkaisusovellus (CDA), jonka tehtävä on rakentaa tuotetusta

sisällöstä lopullinen julkaistu tuotos. Sisällönhallintajärjestelmiä tarjotaan itse hallitavina ratkaisuina, kuten WordPress.org ja Drupal, sekä täysin ylläpidettyinä palveluina, kuten WIX ja WordPress.com. Ylläpidetyn ja itse hallittavan ratkaisun isoimpana erona, vastuujaon lisäksi, on että ylläpidetty palvelu ei mahdollista järjestelmän koodien muuttamista, sillä asiakkaalla ei ole pääsyä varsinaiseen ajoympäristöön tai sen koodeihin. Esimerkiksi WordPress.com tarjoaa ainoastaan CMA- ja CDA-toiminnallisuudet, eli tällöin WordPress-sovellus on palvelu (SAAS). WordPress.org on puolestaan WordPress-ohjelmisto kehitysalustana (PAAS).

Kehittäjän näkökulmasta, CMS:ien tarkoitus, on yksinkertaistaa tietomallien rakentelu ja luoda sopiva abstraktio tietovaraston ja käyttäjärajapinnan väliin, jotta ylläpidettävän koodin määrä pysyy maltillisena. Ne on myös suunniteltu vähemmän teknisten henkilöiden työkaluiksi, joilla sisällön julkaisu tapahtuisi vaivattomasti. Useammat sisällönhallintajärjestelmät tekevät käyttöönoton suhteellisen helpoksi ja tarjoavat valmiiksi kaiken olennaisen täysveristen sivustojen pystyttämiseen. Monet blogit ja verkkokaupat pyörivätkin jonkin sisällönhallintajärjestelmän päällä, sillä niiden perustoiminnalisuuksienkin kehittäminen tyhjästä ja ylläpito veisi paljon resursseja.

Kuten kehystenkin kanssa, jokaiseen projektiin sisällönhallintajärjestelmää ei välttämättä kannata implementoida. Ne ovat kuitenkin aika isoja kokonaisuuksia ja vaativat täten tutustumista niiden käytäntöihin. Kaikki ohjelmistoprojektit eivät edes tarvitse sisällönhallintaa, joten ylimääräinen monimutkaisuus vaan hidastaisi projektin etenemistä.

Tilausportaalissa sisällönhallintaominaisuudet tulevat tarpeeseen, kun sovelluksen tarjoamaa palvelua ylläpidetään. Ylläpitoon kuuluu ensisijaisesti sovellusten käyttäjien hallinta ja datan julkaisu ihmisen luettavaan muotoon. Ylläpitäjät eivät välttämättä ole ohjelmistokehittäjiä, joten käytön tulee olla mahdollisimman helppoa.

ASP.NET:in tueksi valitsin tietojen tallennustason hallintaa varten Umbraco-sisällönhallintajärjestelmän. Umbraco on avoimeen lähdekoodiin pohjautuva, MIT-lisenssin alla toimiva, ilmainen järjestelmä. Se hyödyntää ASP.NET:in perusominaisuuksia ja lisää niihin omaa lisätoiminnallisuuttaan. Umbracon ylläpidosta vastaa Umbraco Core Team.

Umbraco on myös vahvasti Riihisoft Oy:n käytössä konsultointipuolella, joten tämä projekti antaisi minulle valmiuksia mahdollisesti konsultointipuolelle siirtymisessä.

Umbracolla on muutama kilpailija ASP.NET-maailmassa. Niiden vertailussa pyrin kartoittamaan mahdollisia syitä poiketa Riihisoft Oy:n linjauksesta, vaikka valinta olikin käytännössä jo tehty.

Taulukko 4 Sisällönhallintajärjestelmien vertailua

CMS	Lisenssi	Tekninen tuki	Lähdekoodi
DotNetNuke	Ilmainen/maksullinen	Kyllä (maksullinen)	Avoin
EpiServer	Maksullinen	Kyllä	Suljettu
Sitefinity	Maksullinen	Kyllä	Suljettu
Umbraco	Ilmainen/maksullinen	Kyllä (maksullinen)	Avoin

Kaikista järjestelmistä löytyy jokin maksullinen lisenssi, mikä tarjoaa eri asteista teknistä tukea, lisenssin tasosta riippuen. Pienemmissä projekteissa voidaan yleisesti katsoa eduksi, että käytettävissä on ilmainen versio, minkä voi myöhemmin tarvittaessa päivittää tukea tarjoavaan lisenssiin. Kynnys aloittaa kehittäminen ilmaisella järjestelmällä on tällöin matalampi. Tuki on tärkeä ominaisuus, sillä se takaa ongelmatilainteisiin asiantuntijan apuun, eikä kehittäjän tarvitse turvautua ainoastaan keskustelupalstojen vastauksiin.

En kokenut tarpeelliseksi selvittää jokaisesta lisenssistä hintoja, sillä ne olivat kaikki sopimuskohtaisia, eivätkä välttämättä verrannollisia palveluidensa osalta.

Avoin lähdekoodi saattaa joidenkin projektien kannalta olla hyvinkin olennainen ominaisuus, sillä jossain vaiheessa projektia, ainoa tapa edetä voi olla haarauttaa oma versio lähdekoodista. Suljetun lähdekoodin ohjelmistot eivät tätä pääsääntöisesti salli.

3.3.3 Kirjastot

Valinnat ohjelmistokehysten suhteen vaativat omat riippuvuutensa kirjastojen osalta, mutta kehittämisen aikana tulee myös vastaan tilanteita, jossa on hyvä harkita muidenkin kirjastojen hyödyntämistä. Vanhemmilla kehittäjillä saattaa olla alkumääritysten pohjalta jo valmiiksi ajatus joistain kirjastoista, millä toiminnallisuuksia voitaisiin lähteä toteuttamaan.

Umbracon mukana tuli pitkälti kaikki mitä projektin ensimmäiseen versioon tarvitsisin, joten en lähtenyt turhaan latailemaan ylimääräisiä kirjastoja. Umbraco sisälti riippuvuudet kantayhteyksiin, taulujen indeksointiin, mallien käsittelyyn, sekä navigaatiologiikkaan. Tällä hetkellä kaikki kirjastot ovat joko uusimpia versioita tai lähes uusimpia ja Umbracon sisäinen toimintalogiikka nojaa niihin. Nähtäväksi jää tulevaisuudessa, kun kirjastoja olisi aika päivitellä, että miten hankalaksi niiden päivittäminen osottautuu.

3.3.4 Kielet

Valitut ohjelmistokehyslinjaukset rajaavat hyvin pitkälti käytettävien kielten määrää. Osa linjauksista tehtiin tosin kielten perusteella, joten suhde toimii molempiin suuntiin. Palvelinpuolen kielipaletiksi karsiutui C# ja selaimessa ajettavat kielet ovat HTML, CSS ja JavaScript.

C# on Microsoftin .NET-alustalle kehitetty, korkean tason, oliopohjainen ohjelmointikieli. Sen vahvuuksina palvelinpuolen ohjelmoinnissa on sisäänrakennettu muistinhallinta, vahva tyypitys ja komponentti-orientoitunut lähestymistapa ohjelmointiin. C# sisältää .NET-alustan kieli-integroidun kyselyominaisuuden (LINQ), jonka avulla C#:ia voidaan hyödyntää myös kantakyselyiden abstraktiona. Varsinaista ”raakaa” SQL-kyselyä ei tarvitse käsin kirjoittaa, vaan kirjasto rakentaa kyselyn ajon aikana. C# on tulkattava kieli, joten se vaatii aina .NET-ajoympäristön toimiakseen. Vaikka C# on rakennettu olemaan tehokas ja nopea kieli, se ei

kuitenkaan pyri kilpailemaan alemman tason kielien, kuten C:n tai assembly-kielen kanssa. Nykyään laitteistojen suorituskyky on sen verran halpaa, joten marginaalinen etu koodin tehokkuudessa ei ole syy harkita muuta kieltä, projektin kehittämisen hankaloitumisen kustannuksella.

Palvelinpuolen ohjelmointikieliä on C#:n lisäksi useita, kuten Java, PHP, Python ja Ruby. Periaatteessa millä tahansa kielellä voidaan rakentaa palvelinsovelluksia, mutta käytettävyyden kannalta, hyvä palvelinpuolen ohjelmointikieli edellyttää, että jokin web-ohjelmistokehys on rakennettu ja ylläpidetty sen ympärillä.

Taulukko 5 Ohjelmointikielien vertailua

Kieli	Kehittäjä	Esimerkkikehys	Paketinhallinta
C#	Microsoft	ASP.Net	Nuget
Java	Oracle	Spring	Maven
PHP	Zend Technologies	Laravel	Composer
Python	PSF	DJango	Pip
Ruby	Yukihiro Matsumoto, et al	Rails	RubyGems

Kielen tehokkuus, dokumentoinnin taso ja takuu jatkokehityksestä ovat myös erittäin olennaisia ominaisuuksia kieliä valitessa. Kielen tehokkuuden määrittely ei ole aina

yksittäinen arvo, vaan pikemminkin olennaisten mittarien vertaamista projektin tarpeisiin. Joskus kielen olennaisin ominaisuus on lajittelutehokkuus isoilla datamäärillä ja tietyillä kriteereillä. Toisinaan se voi olla ylläpidettävien rivien määrä tiettyjen toimintojen kirjoittamiseen. Tämä kaikki on aika monimutkaista, mutta isommissa projekteissa se voi olla kriittisin pohjatyö.

Taulukko 6 Ohjelmointikielien käytön, markkiosuuden ja suosion vertailua

Kieli	Käyttö ja kasvu (2020 – 2021) (StackOverflow, 2020; StackOverflow 2021)	Markkinaosuus ja kasvu (2020 – 2021) (Q-Success, 2022a)	Suosio ja kasvu (2020 – 2021) (TIOBE Software BV, 2022)
C#	27,86% (-11,3%)	9,3% (-12,3%)	3,95% (-8,6%)
Java	35,35% (-12,1%)	3,2% (-13,6%)	11,96% (-29,2%)
PHP	21,98% (-16,1%)	79,1% (+0,3%)	1,99% (-17,4%)
Python	48,24% (+9,4%)	1,4% (-7,7%)	11,72% (+20,8%)
Ruby	6,75% (-5%)	4,3% (+43,3%)	N/A

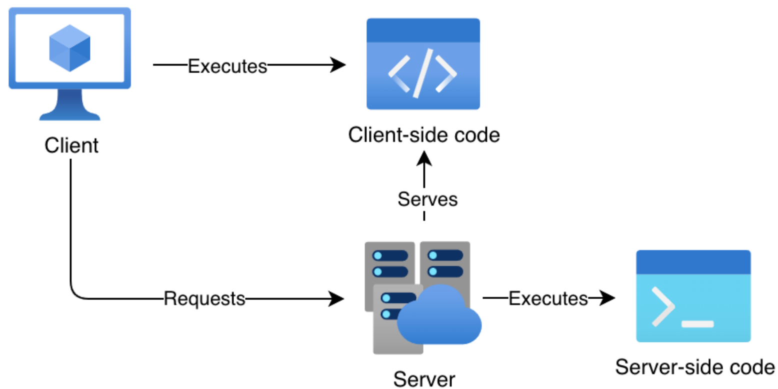
Kielen markkinaosuuteen kannattaa suhtautua jokseenkin kriittisesti, sillä osuudeltaan isoin kieli ei välttämättä ole kuitenkaan aina se paras vaihtoehto. Paljon käytetyn kielen hyvä puoli on kieltä käyttävän yhteisön luoma dokumentointi, virallisen lisäksi. Näin ongelmatilanteissa apu on yleensä helposti saavutettavissa. Toisaalta, kielen suuri markkinaosuus voi olla

puhtaasti ilmiö ajalta, jolloin kilpailua ei juurikaan ollut ja legacy-järjestelmien määrä on edelleen iso siivu käyttöasteesta. Kaikki lähteet eivät myöskään ole samassa mielessä verrannollisia, sillä osa saattaa rajata tuloksia tiettyjen kriteerien perusteella. Sen takia on hyvä ottaa muutama eri lähde trenditietoja varten ja tarkistaa otannan rajaukset.

Käytön ja markkinaosuuden olennainen ero on siinä, ettei käytön prosenttiosuudet sulje toisiaan pois, kun taas markkiosuus tarkoittaa jokaisen kielen siivua kokonaisuudesta. Stack Overflowin käytön mittari ei myöskään ole web-sovelluskehitykseen rajattu, vaikka suurin osa vastaajista olivatkin web-kehittäjiä. W3Techsin otanta rajaa vain web-sovelluksissa käytettyjen kielien osuudet. Suosio on siinä mielessä mielenkiintoinen, että se ei pohjaudu varsinaisesti kumpaankaan edellämainituista mittareista, vaan pikemminkin kielen hakuoptimointiin ja trendidataan. Tiobe kerää useammasta hakukoneesta tietyillä avainsanoilla tuloksia ja laskee niiden mukaan kielen ”suosion.” Se ei kerro, että onko kieli ”hyvä” tai että kuinka monta riviä koodia kyseisillä kielillä olisi kirjoitettu. Tiobe indeksi voi kuitenkin olla hyödyllinen työkalu projektin kielivalintaa tehdessä.

JavaScript on korkean tason, dynaamisesti tyypitetty, skriptikieli, jota tyypillisimmin ajetaan selaimessa. Noin 97,9% verkkosivuista hyödyntävät sitä selainpuolen toiminnallisuuksissa (Q-Success, 2022b.) Siksi JavaScriptin ehdoton hyöty web-kehityksessä onkin sen selainyhteensopivuus. Se toimii tehokkaasti selaimissa, eikä sen käynnistymisessä ole merkittävää viivettä, sillä modernien selainten oma JavaScript-tulkki vastaa koodin kääntämisestä ja ajosta. JavaScriptin ominaisuuksiin kuuluu myös sisäänrakennettu dokumenttiobjektimalli, joka mahdollistaa HTML-elementtien muokkaamisen ajon aikana. JavaScript oli alunperin suunniteltu ajettavaksi vain selaimissa, mutta sillä voi nykyään periaatteessa rakentaa vaikka täysin toimivan palvelinpuolenkin. Kuten millä tahansa muulla kielellä, sellaisenaan JavaScriptillä toiminnallisuuksien rakentaminen on aika työlästä. Siksi 81,4% verkkosivuista hyödyntääkin jotakin kolmannen osapuolen JavaScript-kirjastoa (Q-Success, 2022c). JavaScript-koodin ylläpitäminen vaatii paljon suunnitelmallisuutta, sillä se ei ole vahvasti tyypitetty eikä se pakota kehittäjää koodaamaan tietyllä tavalla. Selaimen puolella ajettavaan koodiin liittyy myös tietoturvariskit, sillä kaikki tieto on selainta ajavan laitteen muistissa. Tämän takia on erittäin tärkeää pitää huolta siitä, mitä tietoja käsittelee palvelimen puolella ja mitä voidaan antaa käyttäjän laitteen muistiin.

Kuva 4 Asiakas- ja palvelinpuolen koodien ajo

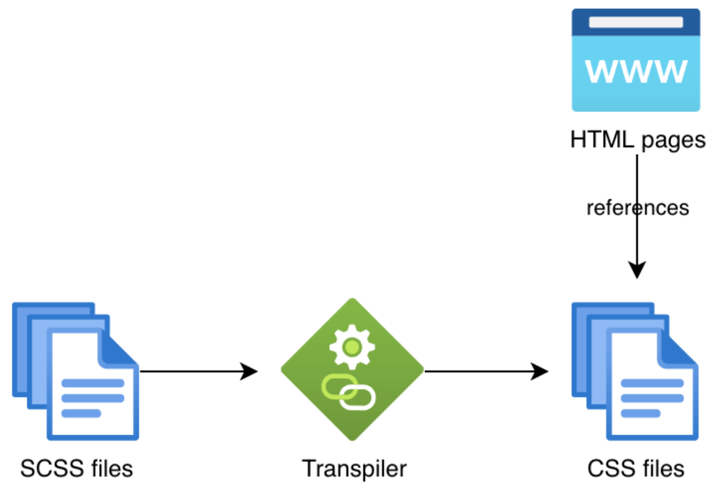


HTML on merkintäkieli, jonka vastuulla on tiedon näyttäminen selaimessa. HTML on lyhenne sanoista Hypertext Markup Language. Se koostuu sisäkkäisistä ja peräkkäisistä elementeistä, jotka selain renderöi multimedia web-sivuiksi. Verkkosivuja varten HTML:lle ei varsinaisesti ole kilpailevia vaihtoehtoja, jotka toimisivat kaikilla selaimilla. HTML-sisällön generointiin taas puolestaan on useampia eri työkaluja, kieliä ja kirjastoja. Dynaamisen sisällön tuottaminen tapahtuukin yleisimmin yhdistelmällä palvelinpuolen ja selainpuolen koodia.

Visualisointia tukemaan on HTML:n lisäksi CSS, eli Cascading Style Sheets -tyylitiedostoteknologia. Sen avulla sivuja voidaan jäsennellä luettavammiksi ja käyttökokemuksen kannalta mukavammiksi. Selain yhdistää HTML- ja CSS-tiedostot renderöidessään sivustoa. CSS ohjaa tyylejä selektorien avulla liittäen niihin tyylitietoja. Jokainen selain renderöi tyylitiedostoja kuitenkin vähän eri tavalla, joten yhteensopivuus on huomioitava selainkohtaisesti.

Isojen sovellusten tyylitiedostovaatimukset usein paisuvat aika isoiksi kokonaisuuksiksi, joten niiden hallintaan on kehitetty laajennuksia. Näiden laajennusten vastuulla on pitää ylläpidettävien tiedostojen määrä maltillisena ja tehdä mahdollisten muutosten toteuttaminen vaivattomaksi. Laajennuksista suosituimmat ovat esikääntäjät, kuten SASS ja LESS. Esikääntäjä on ohjelmisto, joka kääntää syntaksillaan luodut tiedostot CSS-tiedostoiksi.

Kuva 5 SCSS-tulkkaus



LESS ja SASS ovat lähellä toisiaan monella tapaa; molempien ensisijaiset syntaksit ovat CSS:n osajoukkoja ja molemmat tukevat muuttujien käyttöä. CSS-osajoukko tarkoittaa, että kaikki CSS on syntaksiltaan käypää myös osajoukon syntaksia. Selaimet eivät tue osajoukkojen syntakseja suoraan, joten esikääntäjiä tarvitaan tyylitiedostojen luomiseksi. Konepellin alla LESS kääntyy JavaScriptin avulla ja SASS Rubyä. Vertailun kannalta olennainen ero on käytännön ominaisuuksien välillä. LESS on rajatumpi, eikä tue esimerkiksi loogisia funktioita, muuta kuin tietyissä tilanteissa, kun taas SASS tukee laajalti loop ja ehto -logiikkaa.

Valitsin projektin tyylitiedostojen avuksi SCSS:n, eli SASS:in. Vaikka projekti on suhteellisen pieni ja nyt, tyylitiedostoidenkin suhteen, kehittämisessä on ylimääräisiä askeleita, pidin silti SCSS:ää hyvänä ratkaisuna tulevaisuuden kehittämisen kannalta.

3.4 Työkalut ja kehittimet

Työkalut ja järjestelmät web-kehittämiseen valittiin Riihisoft Oy:n valmiista linjauksista, joten vertailut on täten rajattu ensisijaisesti Microsoftin ekosysteemin tuotteisiin.

3.4.1 Azure DevOps ja Git

Kriittisin osa missään projektissa on versiohallinnan määritykset, sillä se voi suojata työntekijää katastrofaalisilta työn menetyksiltä. Riihisoft Oy:llä oli jo entuudestaan Azuren DevOps-palvelu käytössä, joten projektin hallinta pystytettiin sinne lähdekoodivaraston kanssa.

Azure DevOps Services on Microsoftin ylläpitämä pilvipalvelu projektinhallintaan ja resursointiin. Se on keskitetty portaali, mihin kaikki projektiin liittyvä voidaan kasata. DevOpsista on mahdollista pystyttää myös itselleen paikallinen kokoonpano, mikä on käytännössä täysin samaa ohjelmistoa, kuin pilvessä tarjoiltu ratkaisu. Azure DevOps -projekti sisältää hallintatoiminnallisuuksia tehtävälistoihin (Work items & Boards), dokumentointiin (Wiki), koodivarastoihin (Repos) sekä jatkuvaan toimitukseen ja integraatioon (DevOps Pipelines, Test Plans ja Artifacts). Palvelu integroituu myös saumattomasti muiden palveluiden kanssa, rajapintayhteyksiä hyödyntäen.

Git on ohjelmisto, jonka tarkoituksena on seurata muutoksia tiedostoissa. Se on erittäin tehokas työkalu ohjelmistokehityksessä, sillä se mahdollistaa ei-lineaarisen työnkulun ”haarojen” avulla. Useampi kehittäjä voi työstää samaa koodia rinnakkain ilman, että heidän tarvitsisi kaiken aikaa olla ”lukitsemassa” tiedostoja toisiltaan. Git tukee myös koodien synkronointia ulkoisiin varastoihin, jonka kautta muutokset voidaan pitää ajantasalla muidenkin kehittäjien kanssa. Muutokset ovat tällöin myös turvassa tilanteissa, jossa työasema rikkoutuu palauttamiskelvottomaksi.

Azure DevOps tukee Gitin lisäksi myös Team Foundation Version Control (TFVC) -versionhallintateknologiaa. Olennainen ero Gitin ja TFVC:n välillä on niiden arkkitehtuuri. Git on hajautettu ja TFVC on keskitetty. Hajautettu käytännössä tarkoittaa, että Git ei vaadi mitään keskitettyä varastoa koodille, joten kehittäjän työasemalla on koko muutoshistoria aina mukana. TFVC toimii niin, että kehittäjä lataa ainoastaan uusimman version koodista ja lukitsee muilta tiedostoja ”vain-luku” -tilaan muutosten ajaksi. Se ei salli tiedoston muokkaamista samanaikaisesti useassa paikassa. Tämä tuo omat ongelmansa, kun kehittäjiä on useampia ja työlistalla olevat tehtävät viittaavaat samoihin tiedostoihin. Gitillä vastaavaa

ongelmaa ei ole, koska jokainen kehittäjä työstää omaa paikallista kopiotaan tiedostoista ja koodien yhdistely onnistuu synkronoinnin ja haarojen yhdistelyn avulla. Portaalin kehittämiseen oli provisioitu vain yksi kehittäjä, mutta Gitin tuoma notkeus versiohallintaan oli tulevaisuuden kannalta tarpeeksi hyvä syy valita se TFVC:n ylitse.

3.4.2 Visual Studio

Microsoft Visual Studio on sovelluskehitin (IDE), johon on kerätty paljon eri toiminnallisuuksia helpottamaan kehittäjän työtä. Se ei itsessään tue mitään kieltä, vaan asennuksen yhteydessä Visual Studioon voidaan määritellä liitännäisiä, jotka mahdollistavat tuen lähes minkä vaan kehittämiseen. Microsoftin tarjonnan lisäksi Visual Studioon voidaan asentaa kolmannen osapuolenkin liitännäisiä.

Visual Studion vaihtoehtona on olemassa myös Visual Studio Code. Se ei ole varsinaisesti kehitin, vaan pikemminkin editori. Visual Studio Code on täysin erilainen ohjelmisto, kuin Visual Studio, vaikka nimi antaisikin toisin ymmärtää. Visual Studio on rakennettu VSPackage-teknologian päälle, kun taas puolestaan Code hyödyntää ajossaan Electron-kehystä, eli JavaScriptiä. Olennainen ero IDE:n ja koodieditorin välillä on kuitenkin se, että IDE:n on tarkoitus sisältää valmiiksi työkalut kehittämiseen, kompilointiin, ajoon, debugaukseen ja rakentamiseen. Editori on nimensä mukaan vain koodin muokkaamista varten. Se on usein muovattampi ympäristönä ja siihen voi liittää toiminnallisuuksia, jotka tekevät siitä ominaisuuksiltaan saman, kuin kehitin. Valitsin Visual Studion, sillä sen asennuksen pystyi määritellä suoraan ASP.NET-kehyksen kehittämiseen, jolloin se sisälsi myös IIS-palvelinohjelmiston, minkä päällä koodi ajetaan. Visual Studio Coden olisi varmasti voinut saada vähintäänkin yhtä varustelluksi, mutta sen tutkiminen ja pysyttäminen olisi ollut ehkä liian työlästä tätä projektia varten.

3.4.3 Nuget

Riippuvuuksien hallintaan ja lataukseen on kehitetty liuta erilaisia teknologioita. Nuget Package Manager asentuu Visual Studion ohella ja se on luotu .NET-pohjaisten projektien riippuvuuksien paketinhallintaan. Nuget on Microsoftin palvelu, jossa eri tahot voivat

tarjoilla omia koodejaan kehittäjille. Se tukee versiointia, joten pakettien päivittyessä, kehittäjät voivat päivittää omaa tahtiaan riippuvuudet aina tuoreinpiin versioihin. Versiointi on olennaista, sillä usean kehittäjän työstäessä samaa koodia, riippuvuuksien koodien tulee olla myös samat. Eri versioiset riippuvuudet eivät välttämättä toimi samalla tavalla tai edes sisällä samoja rajapintoja.

Nugetin kaltaisia paketinhallintateknologioita on monia, jotka kaikki palvelevat suunnilleen samoja tarkoituksia. Ne saattavat erota kohdeympäristöistä tai -kielistä ja sisältävät eri paketteja. En lähtenyt vertailemaan paketinhallintaratkaisuja, sillä tähän mennessä linjatut kehykset löytyisivät Nugetin avulla. Useiden paketinhallintapalveluiden käyttäminen samassa projektissa ei ole kuitenkaan mitenkään harvinaista, sillä projektin osat saatetaan ajaa eri ympäristöissä ja täten vaatia eri palvelun.

Vaikka Nuget on ilmainen ja sen tarjoamat koodit ovat lähdekoodiltaan avoimia, ne ovat silti tekijänoikeuksien alaista materiaalia. Kehittäjän täytyy olla erittäin tarkka, käyttäessään ladattuja paketteja, että miten ne on lisensoitu.

3.4.4 MS SQL Server & SSMS

SQL, eli Structured Query Language, on standardisoitu kyselykieli, mitä käytetään relaatiotietokantojen hallintaan. Relaatiotietokannat ovat teknologia, jossa kaksiulotteisten tietotaulujen rivien tiedot voivat viitata toisiinsa, avainten avulla. Tämä mahdollistaa monimutkaisenkin datan tallentamisen indeksoitavaan muotoon. Relaatiokannat ovat tehokas ja järjestelmällinen tapa varastoida strukturoitua dataa. SQL-kannat pitävät sisällään tiedon jäsentämistä varten myös ohjaus- ja metatietoja.

Microsoft tarjoaa SQL-kantoja varten Microsoft SQL Server (MS SQL) -ohjelmiston, jota voidaan ajaa Windows-käyttöjärjestelmän päällä. Itse pystytettävän ja -ylläpidettävän palvelimen lisäksi Azuressa on useita SQL Server -ratkaisuja. Ne noudattavat samaa pilven palvelutasojärjestelmää, mitä aiemmin jo avattiin. Vaihtoehtoja oli ihan virtuaalikoneesta (Bring Your Own License) täysin ylläpidettyihin, globaalisti tasaviiveisiin, palveluihin, kuten

Azure Cosmos Database. Valitsin infrastruktuuriltaan kevyimmän ratkaisun, eli Azure SQL Server -instanssin, sillä se oli myös halvin ja vaati vähiten ylläpitoa.

NoSQL-teknologioiden vertailu SQL-ratkaisuihin ei projektin kannalta ole olennaista, sillä Umbraco vaatii toimiakseen SQL-tietokannan. Muitakin SQL-palvelinratkaisuja on olemassa, kuten MySQL, PostgreSQL ja MariaDB, joita voidaan pystyttää Azureen, mutta pidättäydyin Microsoftin teknologioissa, koska erityistä syytä muuhun ei ollut.

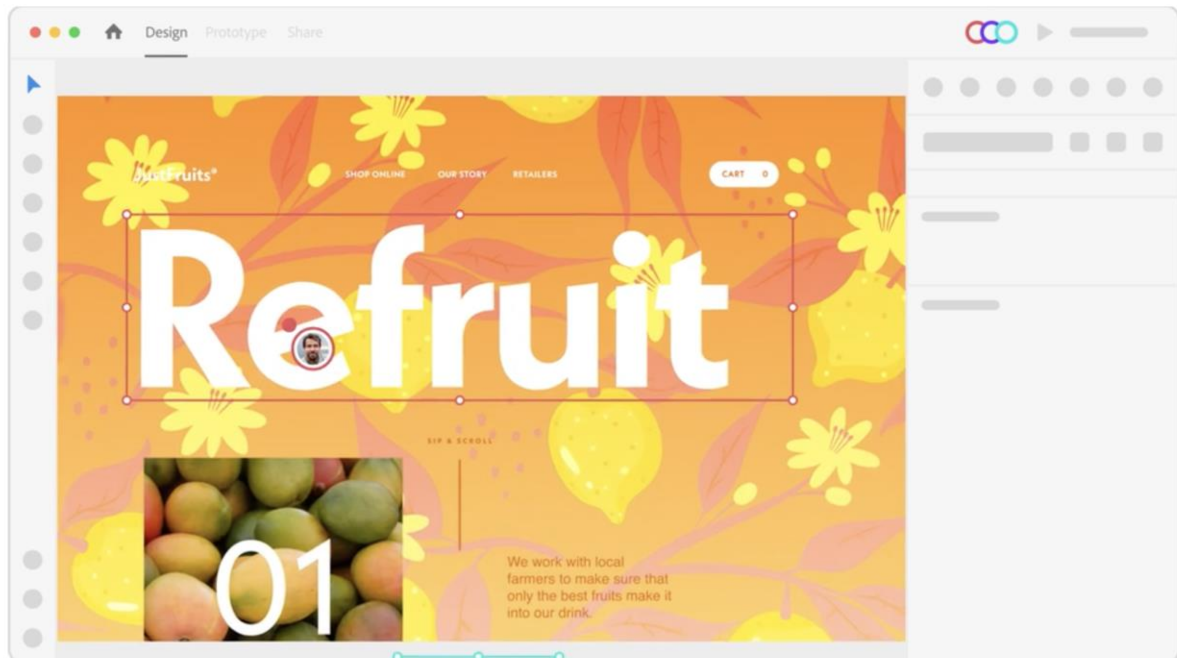
SQL-kantojen hallintaan Microsoftilla on SQL Server Management Studio (SSMS). Sen avulla käyttäjä voi yhdistää eri palvelimiin ja hallita useata ympäristöä samanaikasesti. SSMS sisältää ison määrän erilaisia datan visualisointi- ja automaatiotyökaluja, joilla monimutkaisiakin työvaiheita on helppo suorittaa. Projektia varten tämä työkalu ei ole välttämätön, sillä julkaistavan kannan ylläpidosta huolehtii Microsoft ja tietovaraston tason kanssa juttelu tapahtuu Umbracon toimesta. Otin sen silti kehittämisen ajaksi käyttöön, jotta näkisin paremmin, mitä Umbraco ja ASP.NET oikeastaan tekevät kannalle.

Azure tarjoaa myös pilvessä olevan kannan hallintaan muutamia työkaluja, mutta kehittämisen aikana kanta ja palvelin ovat omalla työasemallani, joten niistä ei ole minulle hyötyä.

3.4.5 Adobe XD

Adobe XD on Adoben kehittämä ohjelmisto, mikä on suunnattu käyttöliittymien ja -kokemuksien hahmotteluun. Se mahdollistaa prototyyppien nopean suunnittelun ja testauksen ilman riviäkään koodia tai ylipäätään mitään erityisempää teknistä osaamista. Käyttöliittymien navigoinnin rakentaminen on mutkatonta ja kaavionäkymät antavat selkeän kuvan suunnitelman sivustorakenteesta. Ohjelmiston käyttöönotto ei myöskään vaadi mitään monimutkaisia pystyttämismenettelyjä tai esimäärityksiä, joten se on helppo ottaa käyttöön projektiin kuin projektiin.

Kuva 6 Adobe XD käyttöliittymä, kuvituskuva



Periaatteessa käyttöliittymien suunnitteluun riittää ihan kynä ja paperikin, mutta projektin etenemistä seurataan välillä etäpalavereissa, joten diginatiivi ratkaisu on helpommin esiteltävissä ja muokattavissa. Ruudulla näkyvä käyttöliittymä on antaa myös heti realistisen kuvan lopputuloksenkin ulkoasusta.

Adoben ohjelmistot olivat entuudestaan itselleni niin tuttuja, joten en nähnyt mitään erityistä syytä edes harkita muita. Se olisi vain hidastanut projektin käynnistymistä, eikä suunnitteluohjelmisto ole kuitenkaan millään tavoin kytköksissä lopputulokseen.

4 Riihi DMA tilausportaali

Portaalin kehittäminen alkoi Azure DevOpsista. Minulle oli pystytetty projekti Riihisoft Oy:n projektipäällikön toimesta ja annettu oikeudet kehittäjän rooliin. Päätimme jättää tehtävälistan ja -taulujen käytön pois projektista, sillä se olisi ollut tässä vaiheessa vaan ylimääräistä byrokratiaa. Projekti on sen verran kevyellä kokoonpanolla, etten hyödynnä mitään erikoisempaa prosessia etenemisen seuraamiseen, kuin viikkopalaverit.

Koodivaraston luonnin jälkeen kloonasin Gitillä tyhjän varaston itselleni, jotta kaiken saisi alusta-alkaen heti talteen.

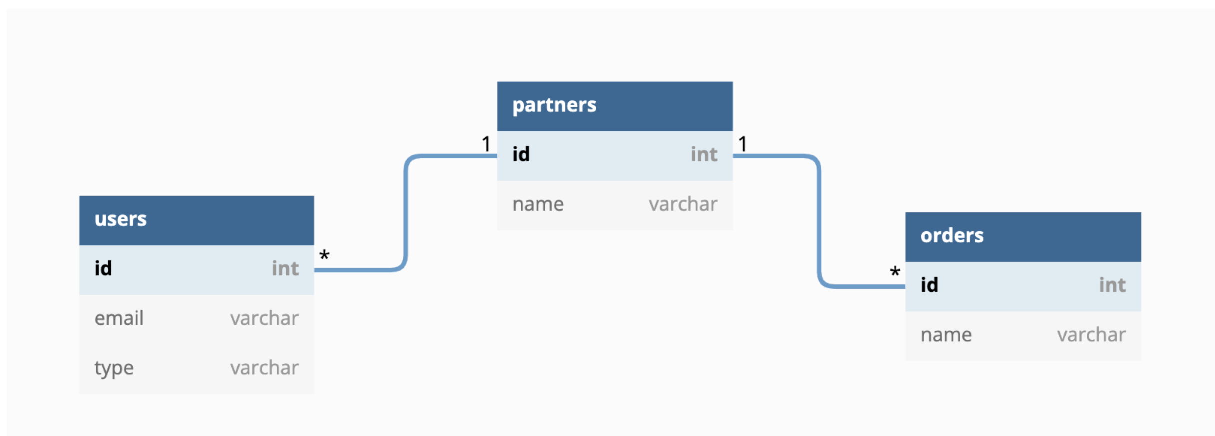
4.1 SQL Server

Aloitin työasemani käyttöönoton asentamalla siihen SQL Server -ohjelmiston. Näin saan paikallisen kantapalvelimen käyttööni, joten yhteys tietovarastoon kehittämisen aikana ei ole riippuvainen verkkoyhteydestä. SQL-palvelimen asennuksen lisäksi asensin SQL Server Management Studion, jolla yhdistin pystytettyyn kantapalvelimeen ja loin uuden kannan. Tuotantotilanteissa kantaan pääsyyn yleensä asetetaan sovelluskohtaisesti eri tunnuksia ja palomuuriasetuksia, jotta vain määrätyt yhteydet sallitaan. Paikallisessa ympäristössä en käsittele tuotantodataa, joten en katsonut tarpeelliseksi käyttää liikaa aikaa näiden asetusten säätämiseen.

Tämänhetkinen asiakasrakenne koostuu kumppanuuksista, jossa jokaisella kumppanilla (partner) on yrityksiä asiakkaina (tenant). Meidän kumppanikäyttäjien tulee siis nähdä ainoastaan heidän asiakasympärisöihinsä (tenantteihin) kohdistuvat tilaukset. Tietokannan taulut olisivat yksi-moneen relaatioilla tässä tapauksessa, eli kumppaneilla voi olla monta tilausta ja monta käyttäjää. Käyttäjää kohden olisi kuitenkin vain yksi kumppani. Työntekijän kuuluu nähdä kaikki kumppanit ja tilaukset, joten määrittelyt pätevät ainoastaan kumppanikäyttäjiin, vaikka nämä kaksi olisivatkin saman taulun riveissä.

Asiakasympäristöihin kohdistuvat tilaukset sisältävät tiedot valituista katalogin tuotteista ja asiakkaan määritykset niihin. Tilaukset voivat myös sisältää asiakkaan toiveita tuotteista, jotka eivät ole katalogissa, vaan ne räätälöitäisiin asiakkaalle tilausta käsiteltäessä. Katalogien tuotteiden relaatio tilauksiin ei ollut projektipäällikön mielestä kantatasolla olennainen, sillä tilaus sisälsi kaiken tarvittavan käsittelyä varten.

Kuva 7 Tietokantakaavio



Tilaus ja asiakasympäristö ovat oheisessa pelkistetyssä tietokantakaaviossa sama taulu, sillä loppuasiakkaan ympäristö luodaan tilauksen pohjalta.

4.2 ASP.NET ja Umbraco

Visual Studiossa on mahdollista perustaa projekteja valmiiden kehysten ja niiden yleisten käyttökohteiden pohjalta. Näiden alkumääritysten avulla ohjelmisto generoi projektin koodille rungon ja asentaa kehyksen vaatimat riippuvuudet. Pystytin projektin ASP.NET web-projektin asetuksilla, jonka jälkeen latasin Umbracon riippuvuudet Nugetin avulla.

4.2.1 ASP.NET ohjelmistokehyksenä

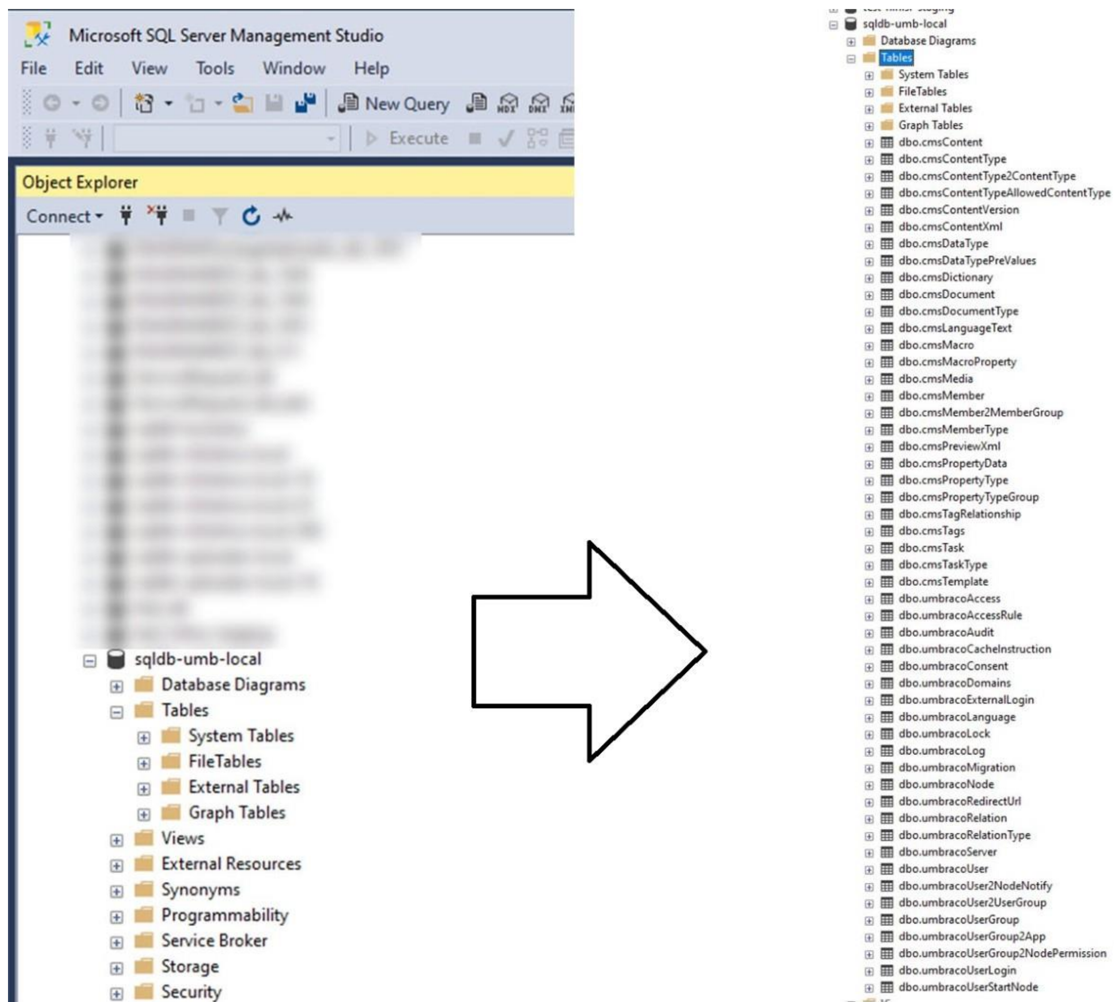
ASP.NET-projektit voidaan alustaa Visual Studiossa muutamalla eri tavalla. Umbracon kanssa kehittämiseen suositeltiin lähtemään liikkeelle MVC-projektin rungolla.

ASP.NET-projekteissa kehittäjällä käytössään valmiita kirjastoja, joilla MVC-komponenttien rakentelu tapahtuu luokkia laajentemalla. Kutsujen reitittämisestä kontrollereille vastaa kehyksen oma logiikka ja näkymien muodostamista varten on Razor-syntaksi, jonka avulla sivujen sisältö voidaan jäsentää omiksi alikomponenteikseen. Näkymät voivat olla myös muutakin, kuin sivuja, kuten vaikka XML- tai JSON-dataa. Mallit ovat perinteisiä .NET-luokkia ja hyödyntävät niiden ominaisuuksia.

4.2.2 Umbraco

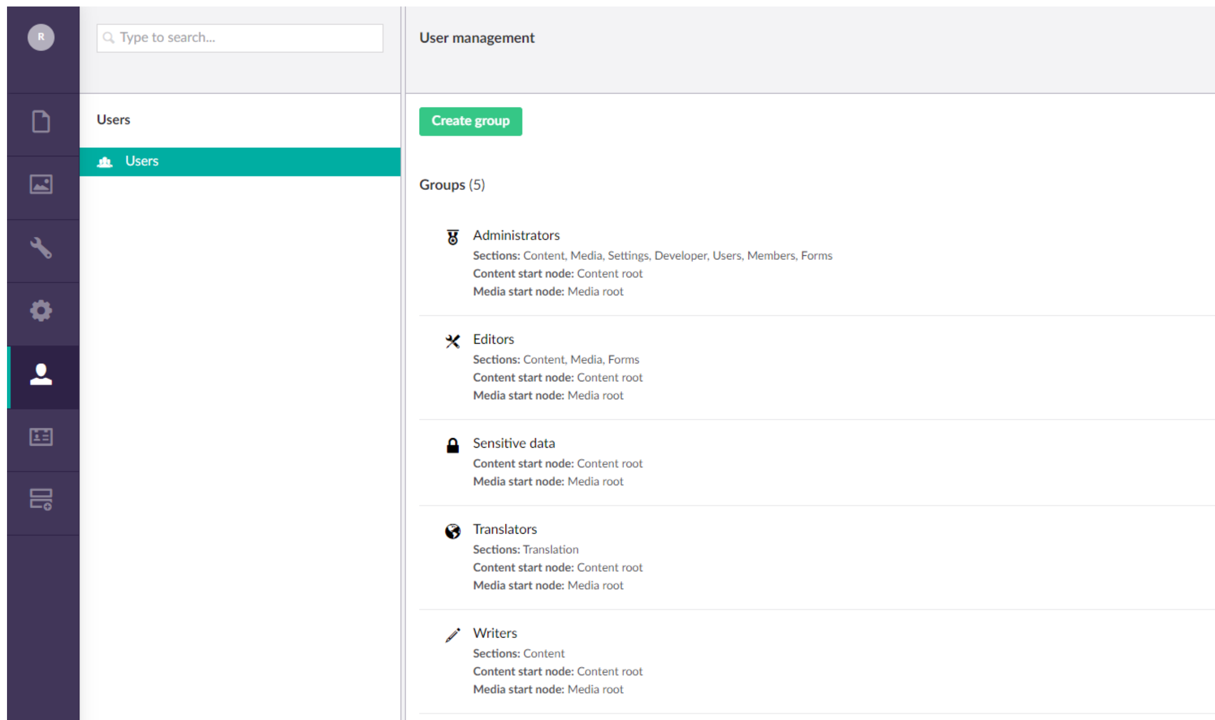
Umbracon ollessa sisällönhallintapalvelu, se vaatii ensimmäisellä ajollaan määrittelyjä asennusta varten. Määrittelyt tehdään selaimen kautta ja Umbraco käynnistää esiasennuksen. Seurasin SSMS:n avulla, mitä Umbraco käytännössä tekee kannalle. (Kuva 8)

Kuva 8 SSMS-näkymä Umbracon asennuksen ajalta



Umbraco luo kantaan omaa toiminnallisuuttaan varten tauluja ja relaatioita. Umbracon sisällönhallinnan puolelle kirjautudaan asennusvaiheessa määritellyllä käyttäjätunnuksella, jonka jälkeen sinne voidaan luoda tunnuksia muille käyttäjille.

Kuva 9 Umbracon käyttöliittymä, roolit



Sisällönhallintaan on iso määrä räätälöitäviä rooleja eri oikeuksilla, jotta ylläpidon käyttäjille ei tarvitse antaa, kuin pakolliset oikeudet työtehtäviensä suorittamiseen.

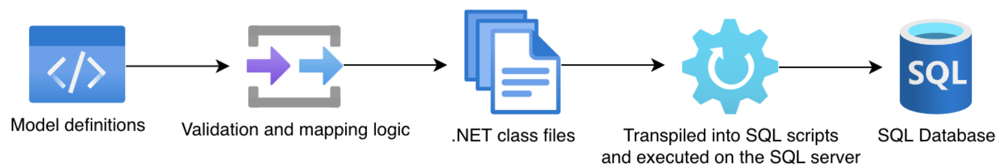
Hallintapuolen käyttäjiä (User) ei tule sekoittaa varsinaisen sovelluksen käyttäjiin (Member), sillä ne eivät ole yksi ja sama. Julkaistua sisältöä voidaan rajoittaa eri Member-käyttäjille tai käyttäjäryhmille oikeuksia jakamalla. Ainoastaan User-käyttäjät voivat kirjautua sisällönhallinnan portaaliin. Tulen hyödyntämään Member-käyttäjien roolittamista sovelluksessa, jotta voin jaotella työntekijät ja asiakkaat omiksi käyttäjätyypeikseen. Loogisesti, jätän tuotekatalogin päivittämisen sisällönhallinnan puolelle, joten ainoastaan User-käyttäjät pääsevät tähän toiminnallisuuteen. Tämä tietysti hankaloittaa vähän sovelluksen ylläpitoa, sillä jos tilausten työstäjän tarvitsee päivittää katalogia, hänellä tulee olla kaksi eri tunnusta sovellukseen. Oletuksena kuitenkin tässä tilanteessa on, että tilausten käsittely ja katalogin päivitys ovat eri henkilöiden vastuulla.

Sovelluksen toimintalogiikan tuottaman datan skeeman hallinta tapahtuu Umbracon backoffice-portaalista. Se luo dynaamisesti .NET mallit käyttäjän määritysten mukaan ja päivittää kannan taulut sopiviksi. Umbracon tietomalleja käytetään julkaistavan sisällön rakennuspalikoina. Jos julkaistavan sisällön on tarkoitus olla sivu, siihen liitetään myös

template-tiedosto, joka sisältää tietojen visualisointilogiikan HTML-muodossa. Katalogin mallia voidaan tällä tavoin periaatteessa laajentaa, sovelluksen julkaisunkin jälkeen, ajon aikana. Tämä ominaisuus sisältää paljon vastuuta ylläpidolta, sillä toimintalogiikka saattaa nojata mallien kenttiin, joten niiden muokkaaminen ajon aikana voi luoda ei-haluttuja sivuvaikutuksia sovelluksen toimintaan. Rajaan tämän ominaisuuden vain projektin kehitystiimille, joka on tietoinen sen riskeistä.

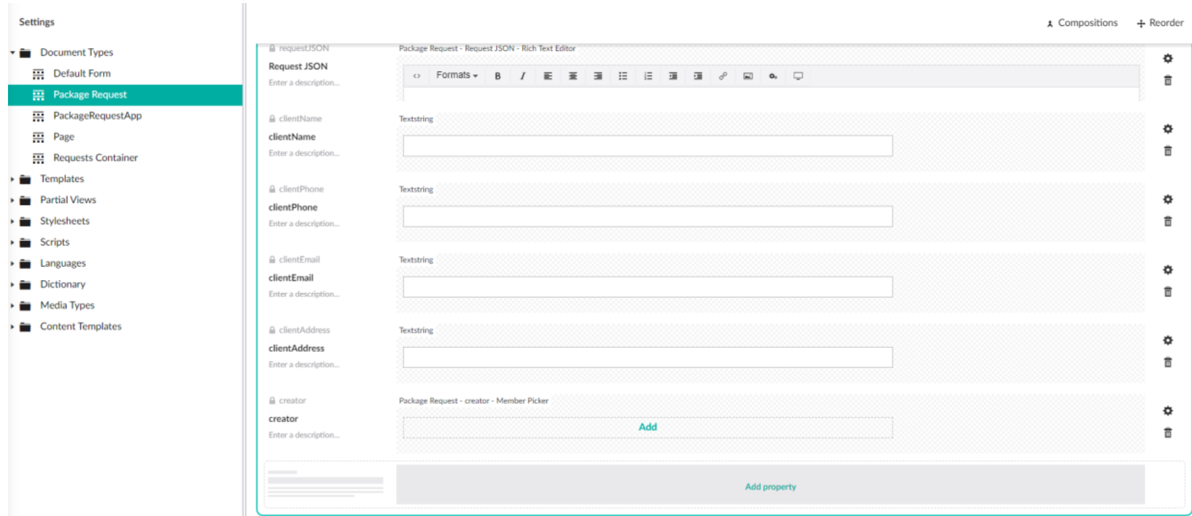
Ohessa yksinkertaistettu kuvion mallien käsittelylogiikasta Umbracossa. Kyseessä on käytännössä Code-first -lähestymistapa, jossa kanta päivittyy ohjelmallisesti luokkatiedostojen määritysten mukaan. Umbracon toteutus eroaa perinteisestä sillä, että kehittäjä ei käsin kirjoita .NET-luokkia, vaan luokat muodostuvat ajon aikana. (Kuva 10)

Kuva 10 Umbracon tietokantamallien käsittelylogiikka



Loin alustavat tietomallit kumppaneita, katalogia ja tilauksia varten. Voin nyt viitata niihin koodissa, vaikka ne ovatkin vielä kentiltään puutteelliset. Määrittelin myös roolit sovelluksen käyttäjille. (Kuva 11)

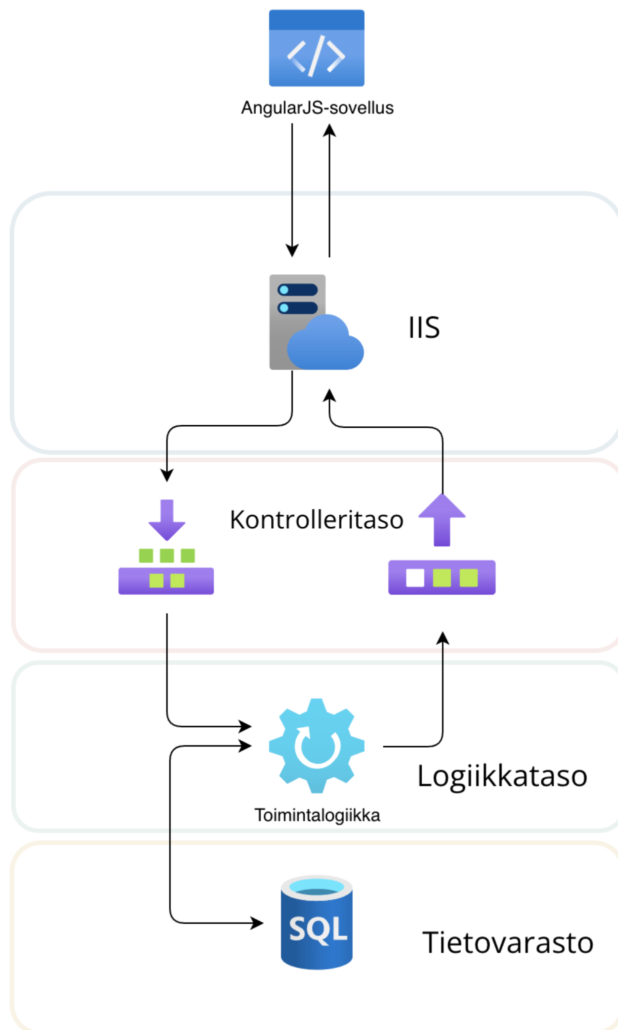
Kuva 11 Tilauksen tietokantamalli



4.2.3 Palvelinsovelluksen rakenne

Suunnittelin palvelinsovelluksen ja käyttäjärajapinnan väliseksi yhteydeksi http-protokollan, jotta myöhemmin toiset palvelut voisivat myös hyödyntää samoja rajapintoja. Tiedonkulku tulisi olemaan käytännössä http-kutsuja käyttäjärajapinnasta palvelimelle ja vastauksina palautettaisiin JSON-näkymä.

Kuva 12 Kutsujen reitittyminen palvelinsovelluksessa

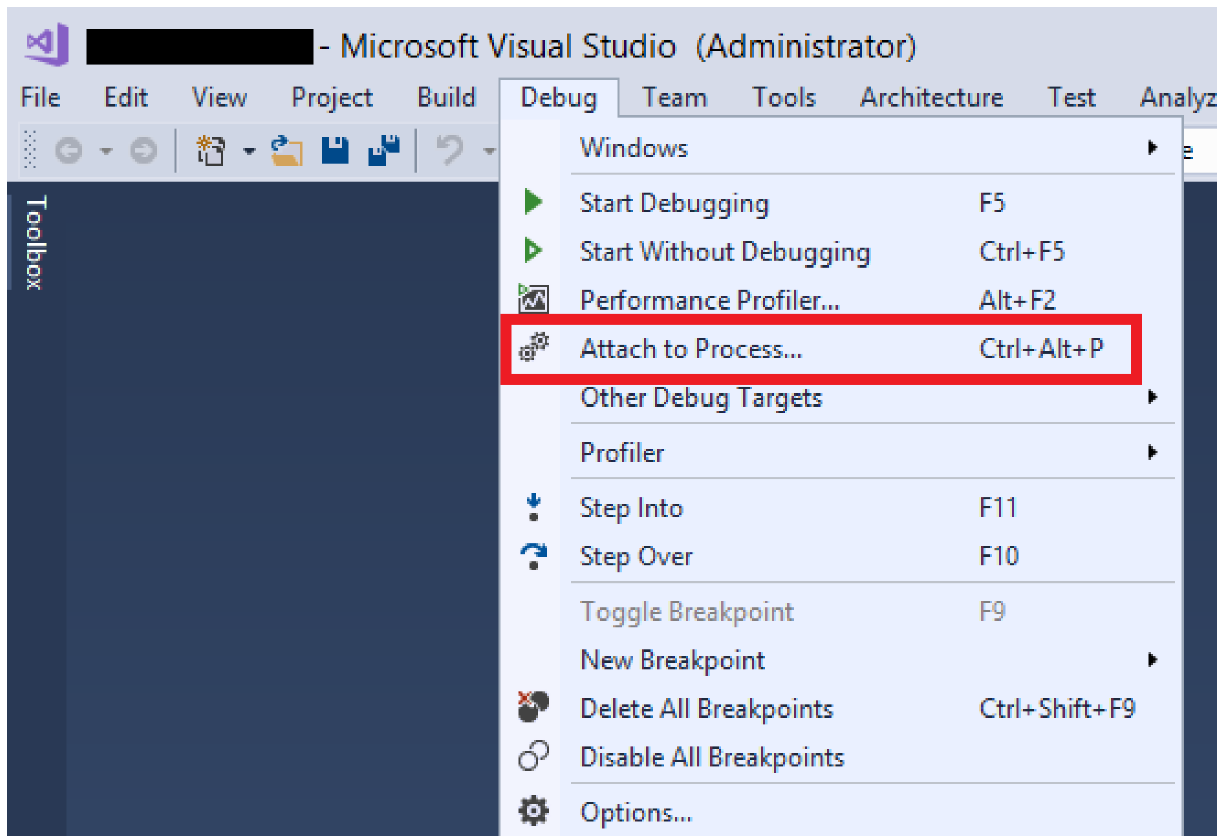


Kutsujen reitittyminen sovellukseen tapahtuu palvelinohjelmiston kautta, jonka jälkeen ASP.NET ohjaa sen käsittelijöilleen.

Visual Studion luoma ja Umbraco-asennuksen täydentämä projektirunko sisältää yhden projektin ja kansiorakenteen kehittämisen aloituksen tueksi. Tein valmiiksi kansiot kontrollereille ja malleille. Itse kehittäminen tapahtuu Visual Studion ja Umbracon backofficen välillä, joten asensin IIS-ominaisuuden työasemaani työnkulkua helpottamaan. IIS on palvelinohjelmisto, joka on suunniteltu web-sovellusten pyörittämiseen. Se mahdollistaa kehitysajan backoffice portaalin käytön, ilman että joutuisin käynnistämään projektia Visual Studiosta kaiken aikaa. Voin nyt rauhassa tehdä koodimuutoksia ja konfiguroida backoffice-portaalia samanaikaisesti. Debug onnistuu Visual Studion ”Attach to

process” -toiminnon avulla, jolloin kehitin yhdistää ajonsa IIS-prosessiin ja voin tarvittaessa rivi kerrallaan keskeyttää sovelluksen ajoa. (Kuva 13)

Kuva 13 Visual Studio, ajon liittäminen prosessiin



Julkaisin testisisältöä haettavaksi käsittelijälogiikkaa kehittäessä. Umbracossa on kaksi eri tapaa sisällön hakuun: LINQ ja Examine. Examine on Lucene-hakumoottorin päälle rakennettu kirjasto. Se pystyy indeksoimaan erittäin nopeasti isojakin datamääriä, mutta kirjaston käyttäminen vaatii aika paljon pohjatietoa Lucenen toiminnasta. LINQ on luettavampi ja toimii parhaiten suhteellisen pienillä datamäärillä. Vaikka LINQ on hitaampi, kuin Examine, se ei kuitenkaan ole missään tapauksessa ”hidas.” (Payne, 2017)

Päätin hyödyntää LINQ-kirjastoa katalogin hakuun ja kirjoitin pienen Examine-apumetodin tilausten indeksointiin, sillä niitä tulee olemaan huomattavasti enemmän, kuin katalogissa tuotteita. Tarvittava Examine-kysely on aika yksinkertainen, sillä sovelluksen ei tarvitse tarkistaa monimutkaisia relaatioita.

```
public class SearchHelper
{

    // We're using Examine + Lucene for efficiency and flexibility
    public IOrderedEnumerable<SearchResult> GetRequests(string memberGUID = "")
    {

        string _searcherName = "requestSearchSearcher";
        string _memberField = "creator";

        // If the user input is nothing that means that the search will return everything that the indexer can index
        string raw = (string.IsNullOrEmpty(memberGUID)) ? "*:*" : String.Format("+{0}:{1}", _memberField,
memberGUID);

        var searcher = ExamineManager.Instance.SearchProviderCollection[_searcherName];

        var criteria = searcher.CreateSearchCriteria().RawQuery(raw);

        var searchResults = searcher.Search(criteria).OrderByDescending(x => x.Fields["createdDate"]);

        return searchResults;
    }
}
```

Tilausten tekemiseen tarvitaan myös oma käsittelijäkomponentti. Sen toiminta on käytännössä kutsun validointi, parsiminen tilausmallin mukaiseksi ja talletus sisällönhallintajärjestelmään. Sisältö täytyy myös ”julkaista”, jotta se indeksoituu tilauksia hakiessa.

```
[HttpPost]
[MemberAuthorize]
[ValidateAntiForgeryToken]
public object SubmitRequest(PackageRequestViewModel packageRequest)
{

    int.TryParse(WebConfigurationManager.AppSettings["requestsParentID"], out int parent);

    if (ModelState.IsValid)
    {
        var userGuid = new UserGUIDHelper().GetUserGUID(Membership.GetUser());

        var newRequest = Services.ContentService.CreateContent(packageRequest.Client.ClientName, parent,
"packageRequest");
        newRequest.SetValue("clientName", packageRequest.Client.ClientName);
        newRequest.SetValue("clientPhone", packageRequest.Client.Phone);
        newRequest.SetValue("clientEmail", packageRequest.Client.Email);
        newRequest.SetValue("clientAddress", packageRequest.Client.Address);
        newRequest.SetValue("creator", userGuid);
```

```
            var json = JsonConvert.SerializeObject(packageRequest.Packages);

            newRequest.SetValue("requestJSON", json);

            Services.ContentService.SaveAndPublishWithStatus(newRequest);

            return Ok(new { message = "Success!" });
        }

        return BadRequest();
    }
```

Käsittelijämetodin määrittelyissä käytetään attribuutteja, mitkä ohjautuvat metodin suorittamista edeltävän logiikan ajoon. Esimerkiksi ”MemberAuthorize” -attribuutti tarkistaa ennen metodin ajoa, että kutsun lähettäjä on authentikoitunut palveluun.

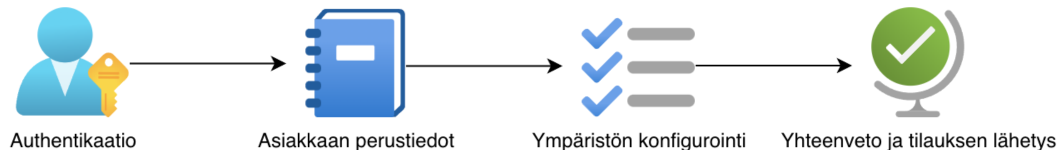
4.3 Käyttäjärajapinta

AngularJS-kehys ohjaa käyttämään ASP.NET:n tapaista MVC-mallia kehityksessä. Ainoa olennainen ero palvelinpuolen toimintaan on, että AngularJS ottaa myös vastuulleen sivujen reitittämisen, eli sovellusta tarjoileva palvelin ei ensimmäisen latauksen jälkeen enää vastaanota kutsuja selaimesta sovelluksen sisällä tapahtuviin sivunvaihtoihin. Käyttäjä ei välttämättä siis huomaisi, jos latauksen jälkeen, palvelinta ei enää olisi, ennen seuraavaa uudelleenlatausta. Tämän tyyppisiä sovelluksia kutsutaan nimellä Single Page App (SPA).

SPA:n hyödyt käyttäjälle ovat nopeat sivujen lataukset ja kehittäjän näkökulmasta, istunnon aikaisten tietojen käsittelyn helppous. Perinteisessä mallissa selaimen muisti tyhjenee sivustolla navigoidessa ja täten istunnon aikaiset tiedot tulee tallentaa selaimen välimuistiin, eli keksiin. SPA toimii tässä tilanteessa eri tavalla, sillä kaikki ajetaan samalla sivulla, joten muistia ei automaattisesti tyhjennetä sivulta toiselle mentäessä. SPA, konseptina, ei ole mitenkään erityisen hyvä hakuoptimoinnin kannalta, sillä sivut generoituvat ajon aikana, eikä niitä voida indeksoida samalla tapaa, kuin perinteisiä sivustoja. Tähän on olemassa ratkaisuja, kuten serverside-rendering, mutta se vaatii paljon enemmän suunnittelua ja työvaiheita, verrattuna perinteiseen tapaan luoda sivustoja. Toteutuksen sovellus on kuitenkin tarkoitettu jaettavaksi linkkinä Riihisoft Oy:n sivuilla ja ainoastaan kumppaniemme käyttöön, joten hakuoptimointi ei ole prioriteetti. Nopeus ja suorituskyky on olennaisempaa.

Lähdin toteuttamaan käyttäjärajapintaa ensin hahmottelemalla varsinasen tilausprosessin. Se toimisi periaatteessa samalla tavalla, kuin vanha malli, eli käyttäjä valitsee katalogista haluamansa tuotteet omilla määrityksillään ja sen jälkeen lähettää sen toteutettavaksi.

Kuva 14 Sovelluksen tilausprosessi



Rakensin käyttöliittymän alkeellisen prototyypin Adobe XD:llä. Vedoksien esittäminen etäpalavereissa tapahtui näytönjaolla ja pystyin näin näyttämään tilausprosessin eri vaiheet interaktiivisena kokonaisuutena.

Umbraco tarjoilee html-sisällön käsittelijänsä kautta ja hyödyntää Master.cshtml -tiedostoa asetteluun, joten rekisteröin vaadittavat JavaScript-kirjastot sen header-osioon.

Ohessa koodinäyte Master.cshtml -tiedostosta.

```
@*CSS*@

Html.RequiresCss("~/Content/bootstrap.css", 1);
Html.RequiresCss("~/css/general.min.css", 2);

@*APP CSS*@

Html.RequiresCss("~/PackageRequestApp/css/style.min.css", 10);

@*JavaScript & AngularJS*@

Html.RequiresJs("~/Scripts/jquery-3.0.0.min.js", 1);
Html.RequiresJs("~/Scripts/bootstrap.min.js", 2);
Html.RequiresJs("~/Scripts/angular.min.js", 3);
Html.RequiresJs("~/Scripts/angular-ui-router.min.js", 4);
Html.RequiresJs("~/Scripts/angular-animate.min.js", 5);
Html.RequiresJs("~/Scripts/angular-sanitize.min.js", 6);
```

```
@*APP*@

Html.RequiresJs("~/PackageRequestApp/app.js", 20);

@*CONTROLLERS*@

Html.RequiresJs("~/PackageRequestApp/controllers/routeCtrl.js", 21);
Html.RequiresJs("~/PackageRequestApp/controllers/loginCtrl.js", 22);
Html.RequiresJs("~/PackageRequestApp/controllers/homeCtrl.js", 23);
Html.RequiresJs("~/PackageRequestApp/controllers/formCtrl.js", 24);
Html.RequiresJs("~/PackageRequestApp/controllers/requestsCtrl.js", 25);
Html.RequiresJs("~/PackageRequestApp/controllers/userCtrl.js", 26);
Html.RequiresJs("~/PackageRequestApp/controllers/wikiCtrl.js", 27);

@*SERVICES*@

Html.RequiresJs("~/PackageRequestApp/services/loginService.js", 40);
Html.RequiresJs("~/PackageRequestApp/services/httpRequestService.js", 41);
Html.RequiresJs("~/PackageRequestApp/services/requestManagerService.js", 42);
Html.RequiresJs("~/PackageRequestApp/services/userService.js", 43);
Html.RequiresJs("~/PackageRequestApp/services/wikiService.js", 44);
Html.RequiresJs("~/PackageRequestApp/services/filters.js", 45);

@*DIRECTIVES*@

Html.RequiresJs("~/PackageRequestApp/directives/customOnChangeDirective.js", 60);
```

Rekisteröimällä vaaditut riippuvuudet Master.cshtml -tiedostoon, niitä ei tarvitse erikseen määritellä jokaiseen alisivuun.

AngularJS 1.7.2 -version sovelluksissa kaikki nivoutuu app.js -tiedostoon, missä sovelluksen reititys, käsittelijät ja palvelutaso rekisteröidään. AngularJS:n vaatimat kirjastot on määriteltävä html-tiedoston alkuun ”<script>”-tunnisteilla, jolloinka ne latautuvat ensin. Sen jälkeen AngularJS hakee sivulta ”ng-app”-attribuutilla merkityn html-elementin ja käskyttää selainta suorittamaan siihen viittaavan app.js-tiedoston.

Ohessa koodinäyte AngularJS-sovelluksen Umbraco-sivukomponentista. Sovelluksen riippuvuudet tulevat Master.cshtml -sivulta ja varsinainen scriptin suorittaminen tapahtuu ensimmäisen div-elementin sisällä.

```
@inherits Umbraco.Web.Mvc.UmbracoTemplatePage
@{
   Layout = "Master.cshtml";
}
```

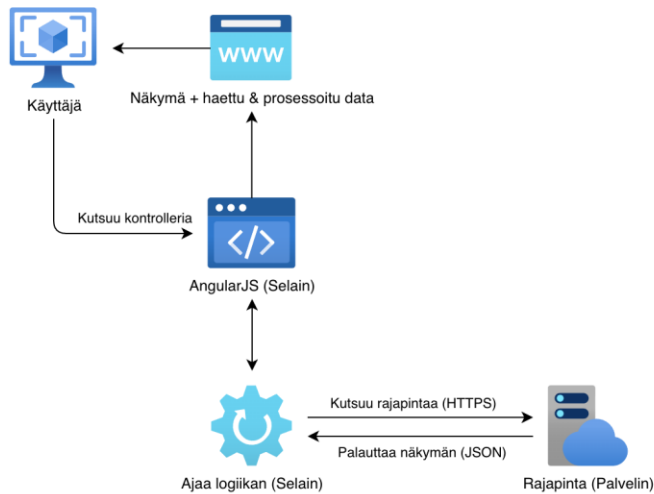
```html
<div ng-app="app" ng-controller="routeCtrl">
  <div class="app-views">
    <div class="view-{{direction}} view-screen" ui-view></div>
  </div>
</div>
```

Alla esimerkki sovelluksen sisäisen reitityksen rekisteröinnistä app.js -tiedostossa.

```javascript
$stateProvider
  .state('login', {
    url: '/login/?:returnUrl',
    templateUrl: pwd + 'templates/login.html',
    controller: 'loginCtrl'
  })
  .state('home', {
    url: '/',
    templateUrl: pwd + 'templates/home.html',
    controller: 'homeCtrl'
  })
  .state('form', {
    url: '/form',
    templateUrl: pwd + 'templates/form/index.html',
    controller: 'formCtrl'
  })
  .state('form.client', {
    url: '/client',
    templateUrl: pwd + 'templates/form/client.html'
  })
  .state('form.packages', {
    url: '/packages',
    templateUrl: pwd + 'templates/form/packages.html'
  })
  .state('form.review', {
    url: '/review',
    templateUrl: pwd + 'templates/form/review.html'
  });

$urlRouterProvider.otherwise("/");
```

Kuva 15 Angular-sovelluksen toimintalogiikka



AngularJS-sovelluksen käsittelijät tuovat näkymiin sisällön ja kutsuvat tarvittaessa palvelutasoa. Palvelutaso juttelee Umbracon rajapintojen kanssa, https-protokollaa hyödyntäen, Ajax-kutsuilla (kuva 15.) Tämä ratkaisu saattaa vaikuttaa tarpeettoman monimutkaiselta, sillä palvelin ja käyttäjärajapinta ovat samalla palvelimella, mutta eriyttäminen mahdollistaa tarvittaessa käyttäjärajapinnan sijoittamisen kokonaan toiselle palvelimelle. Myöhemmin on myös mahdollista vaihtaa koko käyttäjärajapinnan sovellus toiseen järjestelmään, esimerkiksi tilanteessa, missä AngularJS:n tuki lakkaisi.

Kaikki varsinainen validaatio tapahtuu palvelimen puolella, koska käyttäjän laitteella ajettu koodi ei ole ajon aikana enää palvelimen hallittavissa. Muussa tapauksessa osaava henkilö voisi peukaloida käyttäjän laitteella ajettavaa koodia ja täten ohittaa validoinnin kokonaan. Käyttöliittymän validointilogiikka onkin rakennettu käyttökokemusta varten, eikä tietoturvaa. Käyttäjälle on olennaista tietää, että onko heidän syöttämä data oikeassa muodossa ja jos ei, niin he voivat silloin korjata tilanteen ennen varsinaisen kutsun lähettämistä. Näin säästytään turhautumiselta ja käyttö on nopeampaa.

Alla olevien syöttö-elementtien attribuuteissa on määritelty tarkistuksia eri tyyppisille tiedoille. Syötetyt tiedot lähetetään palvelimelle ajax-kutsulla, joten mikään ei oikeastaan estä käyttäjää muuttamasta lähtevää kutsua sisältämään viallisia tietoja.

```
<div class="mt-lg-5">
    <div class="text-center">
        <h4>Enter client information</h4>
    </div>
    <div class="container col-lg-6 text-center">
    <hr />

    <form ng-submit="changeDir('left'); changeState('form.packages');">
        <div class="form-group">
            <label class="">Client name</label>
            <input class="form-control form-text-center" value="temp" ng-model="formData.Client.ClientName"
required />
        </div>
        <div class="form-group">
            <label class="">Client Email</label>
            <input type="email" class="form-control form-text-center" ng-model="formData.Client.Email"/>
        </div>
        <div class="form-group">
            <label class="">Client Phone</label>
            <input type="tel" class="form-control form-text-center" ng-model="formData.Client.Phone"/>
        </div>
        <div class="form-group">
            <label class="">Client Address</label>
            <input class="form-control form-text-center" ng-model="formData.Client.Address"/>
        </div>
        <hr />

        <div class="form-group">
            <input class="btn col btn-primary" type="submit" value="Select packages" />
        </div>
    </form>
    </div>

</div>
```

Palvelinsovelluksen mallimäärityksissä on tämä kuitenkin otettu huomioon

```
public class Client
{
    [Required]
    public string ClientName { get; set; }
    public string Address { get; set; }
    [PhoneAttribute]
    public string Phone { get; set; }
    [EmailAddress]
    public string Email { get; set; }

}
```

Mallin validaatio tapahtuu helposti ASP.Net -kehyksen sisäänrakennetulla attribuutilla.

```
if (ModelState.IsValid)
{
   ...
}
```

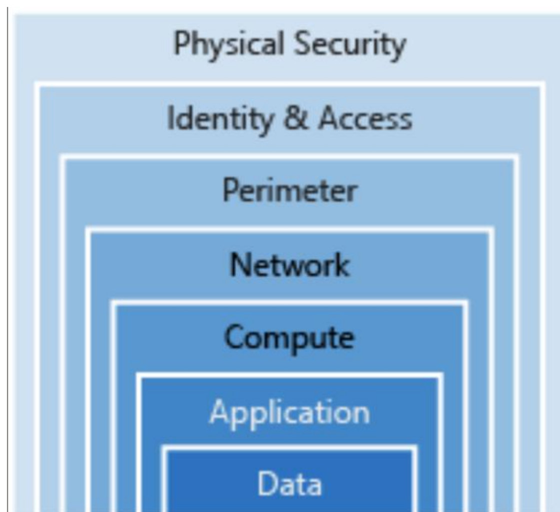
4.4 Sovelluksen suojaaminen

Tietoturvaa määritellessä täytyy ensiksi kartoittaa sovellusta tarjoavan osapuolen vastuualueet ja sovelluksen käsittelemien tietojen arkaluontoisuus. Kartoittamista tulee jatkaa niin pitkälle, kun omat valinnat vielä vaikuttavat turvan tasoon. Esimerkiksi yritys voi olla vastuussa sen resursseihin pääsystä, muttei välttämättä kirjautumispalvelun salauksen vahvuudesta. Tietysti yrityksen tulee osata valita aina turvallisin teknologia, jos sen valintaan on mahdollisuus ja jos ei, niin yrityksen tulee osata luottaa siitä vastaavaan tahoon. Juridisista syistä on erittäin olennaista, että tietomurron tapahtuessa, yritys on alusta alkaen tehnyt kaikkensa tietoturvan eteen, jottei murto johdu yrityksen omista laiminlyönneistä. Sovellus on niin turvallinen, kuin sen heikoin lenkki.

4.4.1 Tietoturva tasoina

Web-sovellusten vastuualueet voidaan jakaa eri kerroksiin. Näin on helppo nähdä, mistä Riihisoft Oy:n vastuu alkaa, portaalin suhteen.

Kuva 16 Tietoturva tasoina, (Microsoft Corporation, 2022, s. 217)



Kuvion tasot muistuttavat hyvin paljon, aiemmin esiteltyä, fyysisten palvelimien ja pilvipalveluiden vertailua, sillä ylläpito ja tietoturva kulkevat käsi kädessä. Ylläpitoon kuuluukin, huoltotöiden lisäksi, mahdollisten tietoturva-aukkojen paikkaus.

Voidaan olettaa, että luotto Microsoftin palveluihin on vähintäänkin riittävää, joten Riihisoft Oy:n vastuulle jää sovellustaso kokonaisuudessaan, sillä provisioin palvelimet PaaS-mallilla. Palvelimien vastuu on sinäänsä jaettu, sillä yrityksen on pidettävä huoli siitä, kuka saa ottaa etäyhteyden palvelimeen. Microsoft vastaa alemman tason ylläpidosta. Riihisoft Oy:n vastuu myös päättyy käyttäjän kirjautumistunnuksiin, sillä niiden tietoturvallisesta säilömisestä käyttäjä on itse vastuussa.

Palvelimelle pääsy on rajattu toimiston ip-osoitteisiin ja Azuren identiteettipalvelun kautta määritellyille käyttäjille. Palvelimen tiedot ovat myös kryptattu.

4.4.2 Sovelluskehitys ja tietoturva

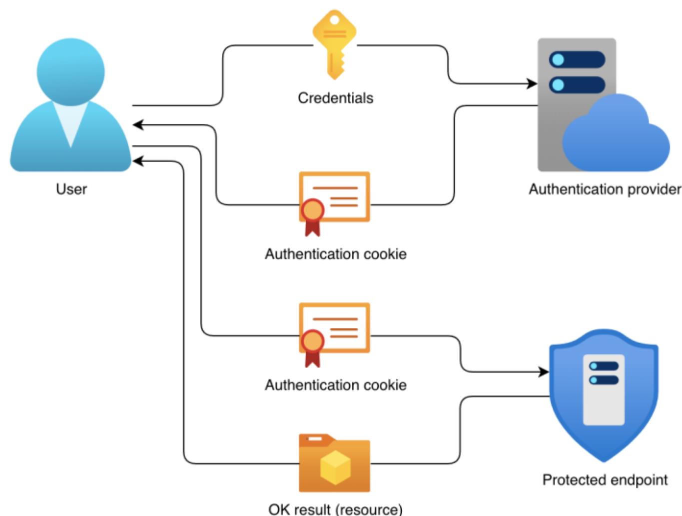
Sovellukseen itseensä liittyvään tietoturvaan voidaan vaikuttaa logiikkaa kehittäessä, eli mitä rajapinta antaa ulos käyttäjälleen. Käyttäjän syöttämien tietojen käsittelyyn on myös otettava kantaa ihan kooditasolla. Esimerkiksi mahdollisimman laajan logittamisen implementointi helpottaa ongelmatilanteiden selvittämistä, mutta liian arkaluontoisen datan

kirjaaminen lokitietoihin on tietoturvariski, sillä nyt tietoja on paikoissa, missä niitä ei välttämättä tarvittaisi. Palvelimen tietojen kryptaus ei mielestäni ole riittävä ratkaisu, koska käyttäjän syöttämiin tietoihin pääsy olisi silti helpompaa, kuin kantaan ja tietoihin pääsisi myös ne henkilöt, joiden ei tarvitsisi.

4.4.3 Authentikaatio

Authentikaatio käyttäjän ja sovelluksen välillä on perinteinen käyttänimeen ja salasanaan pohjautuva kirjautuminen. Kirjautumisen jälkeen käyttäjän selaimeen tallentuu keksi, mikä pitää session yllä. Umbraco sisältää käyttäjien hallintaan ja authentikointiin liittyvät valmiit kirjastot, jotka tallettavat tiedot kantaan turvallisesti, niin että salasanasta jää vain tiiviste. Kirjautumisen voisi myös antaa tehtäväksi ulkoiselle osapuolelle, kuten Azure Active Directorylle. Tässä vaiheessa sovelluksen sisäinen kirjautuminen riittää hyvin, sillä asiakkailla ei ole entuudestaan meidän järjestelmässä tunnuksia muualla.

Kuva 17 Authentikaatio



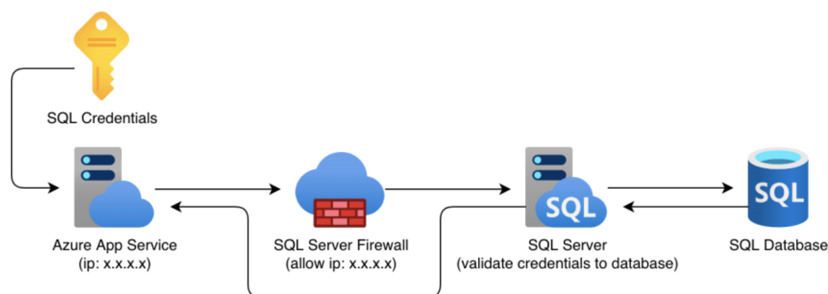
Kuvio esittää yksinkertaista authentikaatioprosessia, missä suojattu resurssi noudetaan kirjautumisen jälkeen saadulla keksillä. Kuvan authentikaatiosta vastaava auktoriteetti on merkitty erillisenä komponenttina, mikä tarkoittaa, että suojatun päätteen täytyy luottaa kyseiseen auktoriteettiin, jotta se voi hyväksyä tämän muodostaman keksin.

4.4.4 Tietokanta

Sovelluksen käyttäjien tuottaman datan suojaamiseen voidaan vaikuttaa useammassa eri tasossa. Tietysti käyttäjätietojen säilyttäminen itsessään on jo tietoturvariski, joten kannattaa miettiä tarkkaan, että mikä on oikeasti välttämätöntä dataa. Kannan rakenteen suunnitteluvaiheessa määriteltiin, että sovelluksen käyttäjien tulisi nähdä vain heille kuuluva data, joten käsittelijälogiikan tulisi palauttaa ainoastaan kirjautuneelle käyttäjälle oikeutettu data. Käyttäjärajapinnassa tämän maskaaminen olisi epäloogista, sillä suodatettava data olisi jo kuitenkin käyttäjän laitteen muistissa.

Tietovaraston eriyttäminen käyttäjärajapinnasta erillisellä palvelimella antaa myös ylimääräisen kerroksen suojausta, sillä käyttäjä ei tällöin keskustele suoraan kantapalvelimen kanssa, vaan välissä on abstraktiokerros. Tietokanta ei myöskään liikennöi nyt kaikkien julkisten ip-osoitteiden kanssa, vaan yhteys rajataan palomuurin avulla vain toimiston ja sovelluksen palvelimen ip-osoitteisiin.

Kuva 18 SQL-tietokantaan kirjautuminen

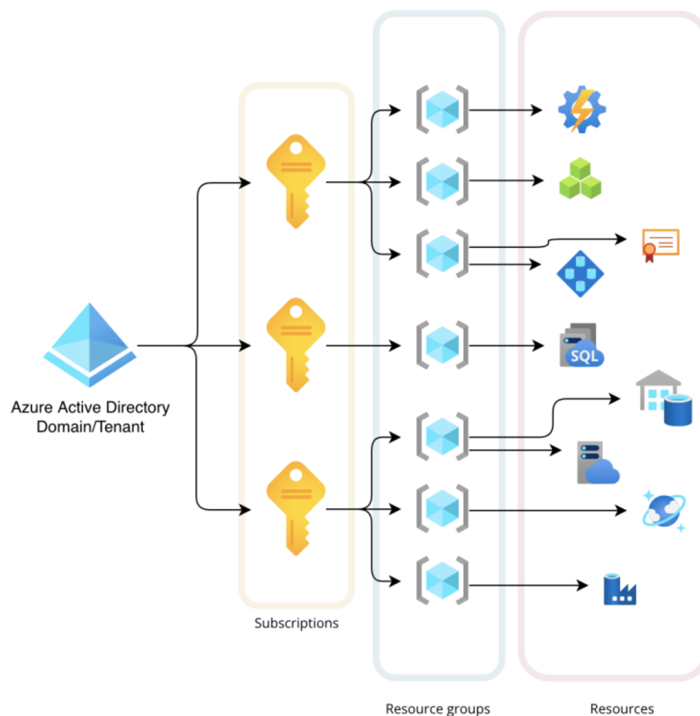


Palomuurimäärittelyjen jälkeen on myös salasanaan perustuva suojaus, jolla sovellus kirjautuu kantapalvelimelle. Salasanaan perustuva kirjautuminen ja palomuuri ovat Azuren tarjoaman SQL-palvelimen sisäänrakennettuja ominaisuuksia. Huomioitavaa tässä järjestelyssä on, että yhteyden tunnukset on oltava muunnettavissa selkotekstiin, sillä kantapalvelimeen kirjautuminen vaatii sen. Tunnukset tulee säilyttää vain siellä, missä se on välttämätöntä, eli tässä tapauksessa sovelluksen palvelimella.

4.5 Azure

Palveluiden pystyttäminen Azureen tapahtuu joko Azure Command Line -käyttöliittymän (CLI) tai web-portaalin graafisen käyttöliittymän kautta. Päätin lähteä portaalin avulla provisioimaan resursseja, sillä koin sen aloittelijaystävällisemmäksi lähestymistavaksi. CLI on ehdottomasti tehokkaampi tapa hallita resursseja, koska se tukee toimintojen automatisointia skriptien avulla.

Kuva 19 Azuren resurssihierarkia

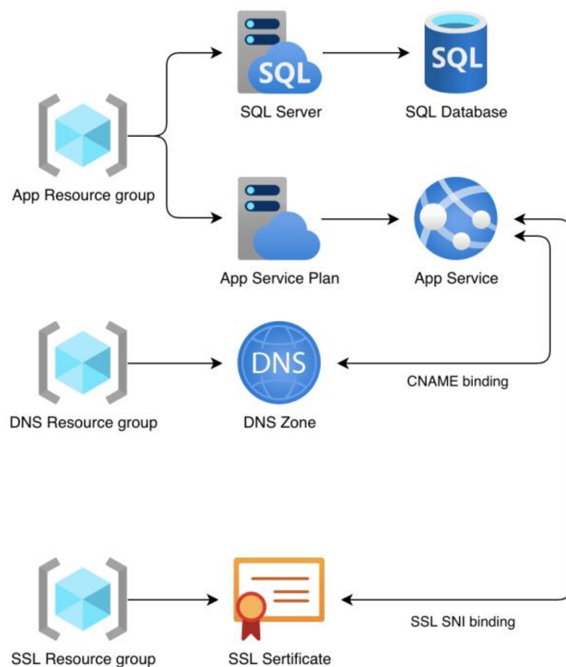


Azuren resurssirakenne on puu, jossa kaiken juurena on tenant, eli verkkotunnus. Tunnuksen alle pystytetään tilauksia, laskutusta varten. Resursseja ei voida sellaisenaan provisioida tilauksiin, vaan ne tulee jäsennellä resurssiryhmiin. Resurssiryhmän voi mieltää kansion tapaisena konttina. Se ei ole siis resurssi itsessään ja se ei sisällä mitään muuta, kuin metadataa itsestään. Niitä voidaan käyttää myös oikeuksien hallintaa varten, eli käyttäjille voidaan erikseen antaa oikeuksia eri resurssiryhmiin, tilauksen sijaan. Tällöin käyttäjiltä pysytään rajaamaan pääsy vain tiettyihin resursseihin tilauksen sisällä.

Riihisoft Oy:llä oli valmiiksi jo tilaus Azureen, joten minulle annettiin Contributor-oikeudet sinne. Contributor-rooli mahdollistaa skooppinsa laajuiset muokkausoikeudet kaikkiin resursseihin. Rooli on koko tilauksen kattava, sillä minun tarvitsee määritellä DNS- ja SSL-sertifikaattiasetuksia, mitkä ovat jo omissa resurssiryhmissään, joten pelkästään tilausportaalin resurssiryhmäkohtaiset oikeudet eivät olisi riittäneet.

Palvelun rakenne suunniteltiin koostuvan käyttäjärajapinnasta, palvelinsovelluksesta ja tietokannasta, joten tarvitsisin näitä varten laskenta- ja varastointiresursseja. Azure tarjoaa käyttöjärjestelmän kanssa varusteltuja virtuaalikoneita tätä varten, joissa on määriteltävissä kovalevyjä tallennusta varten. Virtuaalikone olisi yksinkertaisin ratkaisu, sillä palvelun saisi pystyyn tällöin vain yhdellä resurssilla. Voisin asentaa SQL- ja IIS-palvelimet sinne, jolloinka koko sovellus toimisi kätevästi samasta paikasta. Virtuaalikoneet ovat myös tavallaan käyttäjäystävällisiä, sillä Windows-virtuaalikoneet tarjoavat perinteisen etäyhteyden työpöytäjaolla, joten niiden käyttö muistuttaa perinteisen tietokoneen operointia. Virtuaalikoneiden heikkoudet ovatkin siinä, että käyttöjärjestelmän päälle täytyy asentaa vaadittavat sovellukset, määriteltävä tarpeelliset asetukset ja kaiken lisäksi niiden ylläpito jää tilaajan vastuulle.

Kuva 20 Sovelluksen Azure-arkkitehtuuri



Pyrimme pitämään kaiken mahdollisimman pienellä ylläpitotaakalla, jotta kehittäjäresurssit voitaisiin hyödyntää uuden kehittämiseen. Azuresta onneksi löytyy myös valmiiksi ylläpidettyjä instansseja SQL- ja web-palvelimista, jolloinka ainoastaan sovelluksen koodien ylläpito ja julkaisu olisi meidän vastuullamme.

4.6 Azure App Service

Azure App Service mahdollistaa web-sovellusten pystyttämisen ilman virtuaalikoneiden asennuksia. Sen voi myös integroida erilaisiin CI/CD-järjestelmiin, kuten Azure DevOpsiin, mikä mahdollistaa koodimuutosten julkaisun automatisoinnin. App Service tukee yleisesti käytettyjä ajoympäristöjä ja kieliä, kuten Java, Python, .NET ja Node.js.

App Service nivoutuu aina taustalla olevaan laskentakokoonpanoon, eli App Service Plan -resurssiin. App Service Planin voi mieltää eräänlaisena palvelinkoneena, sillä sille määritellään suoritin, muisti ja kovalevy, mutta siihen tai sen käyttöjärjestelmään ei ole oikeastaan mitään pääsyä. App Service Planiin voidaan pystyttää useampi App Service, jolloin sen resurssit jaetaan näiden välillä.

Pystytin halvimman, tuotantokäyttöön tarkoitetun, App Service Planin ja lisäsin siihen yhden App Service -instanssin. App Service Planeja voidaan aina skaalata tarpeen mukaan tehokkaammiksi, joten fiksuinta on aloittaa kevyimmällä mahdollisella kokoonpanolla, niin kulut pysyvät maltillisina. Latasin myös työasemalleni julkaisuprofiilin, jotta voin viedä koodini Visual Studiosta suoraan palvelimelle.

4.7 Azure SQL Server

App Service -resursseja ei voida määritellä SQL-palvelimiksi, vaan SQL-kantoja varten on olemassa Azure SQL Server -resurssi. Se on myös täysin ylläpidetty palvelu, eli kaikki palvelimen ohjelmistot ja niiden ylläpitö kuuluu Azuren vastuualueelle. Microsoft on myös onnistunut hinnoittelemaan palvelun niin edulliseksi, ettei virtuaalikonetta edes kannata harkita.

Taulukko 7 Hintavertailua (Microsoft Corporation, 2022)

Resurssi	Suorituskyky	Kapasiteetti	Palvelu	Lisenssi	Hinta (€/kk)
Virtual Machine (A1 v2 + Disk)	1 Core 2GB RAM	32GiB	PaaS	Vain käyttöjärjestelmä	42,19€
Azure SQL (DTU)	5 DTU	2GiB	SaaS/PaaS	Kyllä	4,41€

Valitsin vertailuun edullisimmat, tuotantokäyttöön tarkoitetut, vaihtoehdot kummastakin resurssityypistä.

DTU, eli Data Transaction Unit, on Microsoftin laskutusperiaate tietokannoille, mikä pohjautuu kiinteään kuukausihintaan, suhteutettuna valittuun kokoonpanoon. Toisin kuin App Service Plan, DTU-mallin laitekokoonpano on määritellään DTU-yksikköinä, eikä perinteisinä komponentteina, kuten suorittimina ja muistina.

Kapasiteettia vertaillessa täytyy ymmärtää, että virtuaalikoneelle tulee asentaa tarvittavat ohjelmistot, eli kovalevytila jaetaan koko instanssin kesken. Azure SQL -instanssin kapasiteetti on puhtaasti vain tietokannalle. Kaksi gigatavua riittää hyvin projektin kantavaatimuksiin.

Vaihtoehtoina olisi serverless- tai vCore-mallit. Serverless tarkoittaa, että taustalla ei ole enää tilaajan hallittavaa kantapalvelinta ja kannalle määritellään tehohaitari, jonka mukaan se skaalautuu kuorman muuttuessa. Käytännön näkökulmasta, vCore muistuttaa paljon DTU:ta, sillä sekin on varattuun kapasiteettiin perustuva laskutusmalli. Se eroaa DTU:sta siinä, että laitekokoonpanossa mainitaan varsinaiset fyysiset komponentit ja niiden määrittelyihin on enemmän vaihtoehtoja. vCore onkin näin omiaan tilanteissa, jossa fyysiset palvelimet on tarkoitus siirtää pilveen, sillä suorittimet ja muut osat voidaan määritellä vastaamaan fyysisten palvelimien kokoonpanoa. Valitsin DTU:n, sillä se oli kustannustehokkain vaihtoehto. Hinnoittelu DTU:lla alkaa noin viidestä eurosta kuussa, kun taas vCore-mallin halvin kokoonpano maksaa noin kolmesataa euroa kuussa.

Ulospäin Azure SQL Server näkyy ihan normaalina kantapalvelimena ja siihen voi ottaa yhteyden vaikka SSMS-työkalulla aivan kuten paikalliseenkin palvelimeen.

4.8 Portaalin julkaisu Azureen ja käyttöönotto

Azure App Service tarjoaa julkaisuprofiilin instassilleen, minkä latasin oman työasemani Visual Studioon. Tällä tavalla pystyin suoraan julkaisemaan sovelluksen Azureen, ilman ylimääräisiä FTP-ohjelmistoja. Koska projektia työsti vain yksi henkilö, emme kokeneet tarpeelliseksi vielä automatisoida julkaisuprosessia Azuren DevOps-työkaluilla.

Kuva 21 Julkaisuprofiilin lataus Azuren App Service -resurssin kautta

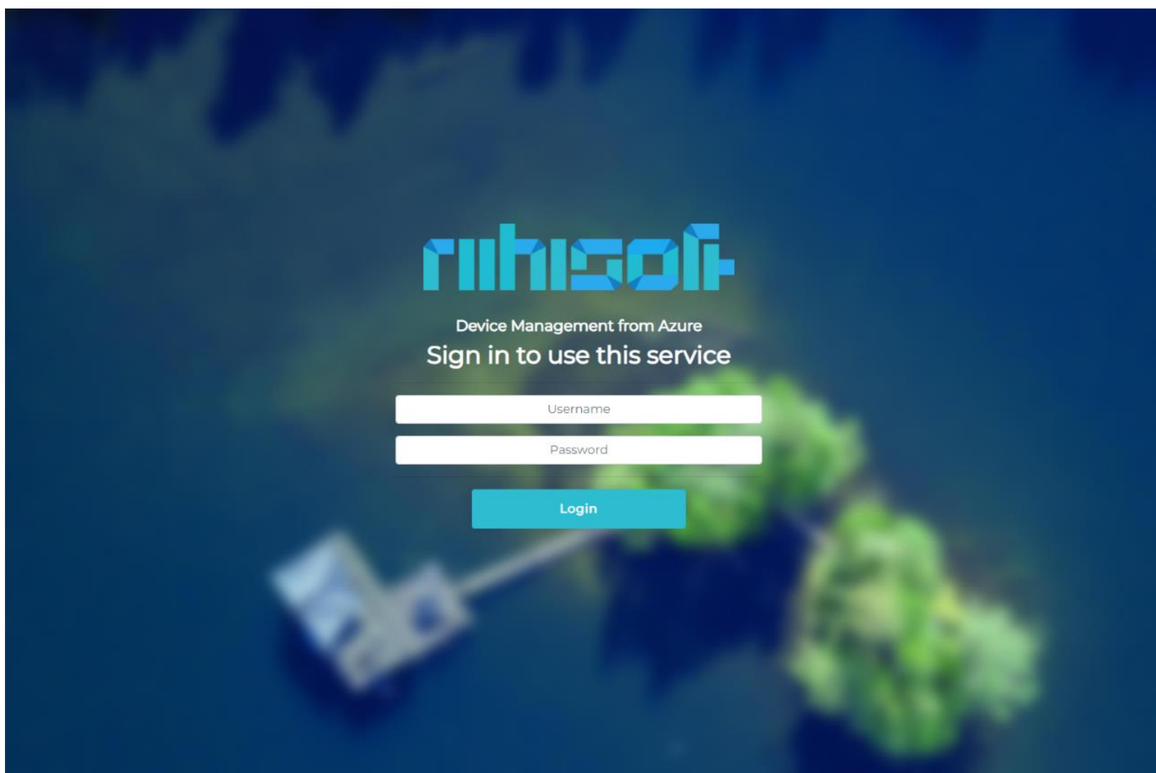
Browse Stop Swap Restart Delete Refresh Get publish profile Reset publish profile Share to mobile Send us your feedback

Asiakkaille tiedottaminen uuden portaalin käyttöönotosta tehtiin tunnusten luonnin yhteydessä. Ajoimme käyttäjiä palveluun porrastetusti, jotta pystyisimme seuraamaan käyttäjien tuottamaa kuormaa. Mahdollisten korjausten vienti tuotantoon tapahtuisi myös matalammalla kynnyksellä, kun käyttäjämäärä ei ole vielä suuri.

5 Projektin tulokset

Projektin tuloksena syntyi internetin kautta saavutettava, web-sovellus. Ensimmäiset tilaukset tulivat jo seuraavana päivänä julkaisusta.

Kuva 22 Sovelluksen kirjautumissivu



Kehitetyn tuotteen oli määrä nopeuttaa uusien asiakkuuksien kehittymistä maksaviksi asiakkaiksi ja sen tuli myös keventää yrityksen työntekijöiden manuaalista, toistuvaa työkuormaa, joten suoritimme muutaman testin siihen liittyen.

5.1 Asiakkaiden näkökulma

Ensimmäisenä ongelmana oli asiakkaiden hidas reagointi pdf-lomakkeisiin, mikä todennäköisesti johtui siitä, että lomakkeet liikkuivat sähköpostien liitteinä. Isojen yritysten päättävässä asemassa olevien henkilöiden sähköpostilaatikot vaikuttavat usein olevan täynnä lukemattomia sähköposteja, joten taistelemme huomiosta kaikkien muidenkin sähköpostien kanssa. On myös mahdollista, että postimme suodattuu roskapostiin.

Näiden lisäksi PDF-lomake tulisi tallentaa koneelle, täyttää ja lähettää meille takaisin. Tämä on erittäin vanhanaikainen ja kankea tapa tehdä tilauksia. Otin testin katalogin tuotemääräksi 48 pakettia, mistä tilaajat voivat valita haluamansa määrän. Mittasin, että katalogin saapumisesta sähköpostiini, sen täyttäminen ja palauttaminen vei 3 minuuttia ja 12 sekuntia, vaikka tiesin suunnilleen, mitä ”tilaisin.” Teetin saman kahdella kollegallani ja laskin meidän keskiarvoksi 2 minuuttia ja 45 sekuntia. Aika ei sinäässä vaikuta pitkältä, mutta jos työasemalla ei satu olemaan mitään pdf-editoria, niin tähän tulisi laskea sen hankkiminen. Koko prosessi oli suorittajien mielestä myös jokseenkin turhauttava ja epäintuitiivinen. Se ei heidän mielestään herättänyt vaikutelmaa mitenkään erityisen ammattitaitoisesta yrityksestä.

Portaalin käyttöönotto, valitettavasti, myös vaatii sähköpostin käytön, sillä rekisteröidyt tunnukset saapuvat sinne. Tämä on tietysti vain kertaluonteinen työvaihe käyttäjää kohden, joten en ota sitä tulosten vertailuun mukaan. Tarkoituksellinen käyttötapaus on kuitenkin usein yhtiökumppani, joka tilaa asiakkailleen ympäristöjä, sitä mukaan, kun asiakkuuksia syntyy.

Mittasin itsellläni sivun latautumisen jälkeisen ajan, kirjautumisesta, tilauksen lähettämiseen ja sain tulokseksi 1 minuutti ja 2 sekuntia. Teetin saman testiryhmällä ja keskiarvoksemme tuli 1 minuuttia ja 30 sekuntia. Tähän lukuun on myös laskettu asiakkaan tietojen lisäys, mitä vanhan järjestelmän ajassa ei ollut. Jätin sen vaiheen tarkoituksella pois, koska silloin tulisi ottaa myös huomioon sähköpostien liikkumisen nopeus ja työntekijän reaktioaika. Käytännössä uuden asiakkaan tiedot ovat jo meillä siinä vaiheessa, kun katalogi lähetetään. Vaikka ajallisesti tulos ei ollut mitenkään mullistava, oli käyttökokemus, testaajien mielestä,

paljon parempi aiempaan verrattuna. En tiedä, vaikuttaako tämä asiakkaan päässä reaktioaikaan, mutta voisin kuvitella, että uusien tilausten tekemisen helppous ei ainakaan nosta kynnystä niiden tekemiseen. Vanha prosessi vaati myös aina uuden sähköpostiketjun asiakasta kohden ja vanhojakin tuli säilöä, jos ympäristöön tehtiin lisätilauksia. PDF-katalogi täytyi myös lähettää jokaiseen ketjuun, jotta aina ajantasainen versio oli kumppanilla. Portaalin kautta uusien ympäristöjen tekeminen tapahtui aina samoilla tunnuksilla ja käytössä oli aina varmasti viimeisin katalogi.

5.2 Ylläpidon helpottaminen

Toinen ongelmakohta oli, katalogin kehittyessä, pdf-tiedostojen ylläpito. Umbracon backoffice tarjoaa nyt graafisen käyttöliittymän uusien tilattavien pakettien lisäämiseen sekä jo olemassaolevien muokkaamiseen. Aikaisemmin tiedostoa pystyi päivittämään vain yksi työntekijä kerrallaan, jotta se pysyi ajan tasalla. Tämä myös pakotti kaikki työntekijät aina lataamaan katalogi omalle työasemalleen, jotta aina viimeisin versio lähetettäisiin asiakkaalle. Prosessi tuli toistaa joka kerta, kun uusia asiakkaita liittyi palveluun.

Ajan säästö ei ole kovin helposti mitattavissa, sillä kyseessä on monivaiheinen prosessi; Sähköposteihin vastaaminen ja tiedostojen hallinta kuormittaa työntekijää jo itsessään, koska on monta muistettavaa asiaa ja tietty järjestys eri vaiheille. Työvaiheet ovat portaalin myötä vähentyneet ja vaativat asiakkuuksien lisäämiseen vähemmän työntekijän läsnäoloa.

Palvelun tuottamista kuukausikustannuksista voidaan kuitenkin laskea, milloin se on aidosti taloudellisesti hyödyllinen.

Taulukko 8 Palvelun tuottamat kustannukset (Microsoft Corporation, 2022)

Resurssi	Kustannukset (€/kk)
Azure App Service (B1, 1 Core)	49,31€/kk
Azure SQL Database (5 DTU)	4,41€/kk
SSL-sertifikaatti (Wildcard)	250€/vuosi, eli ≈ 20,83€/kk
Yhteensä	74,55€/kk

Oletetaan, että työntekijän aika maksaa yritykselle noin 25€/tunti. Taulukon perusteella voimme täten todeta, että kun palvelu on säästänyt työntekijöiden aikaa vähintään kolme tuntia kuukaudessa, se on maksanut itsensä takaisin.

5.3 Jatkokehitys

Ensimmäisinä kommentteina tilausten käsittelijöiltä tuli käyttäjien hallinnan taakka. Emme alkuun anna kumppaniemme lisätä uusia käyttäjiä, joten työntekijöiden tulee itse hoitaa se. Tällä hetkellä kumppanikäyttäjien määrä on aika maltillinen, mutta palvelun skaalautuessa, täytyy keksiä jokin ratkaisu käyttäjien hallintaan.

Tilauksia ei voida tällä hetkellä muokata, lähettämisen jälkeen. Käyttäjät joutuvat pahimmassa tapauksessa tekemään ympäristöä kohden useita tilauksia tai ottaa asiakaspalveluun yhteyttä, jos virheellisiä tilauksia syntyy. Johdonmukainen järjestelmä tilausten muokkaamiseen tehostaisi tilausten käsittelyä.

Ympäristötilaukset käsitellään edelleen manuaalisesti, eli täysin automaattinen järjestelmä tämä ratkaisu ei ole. Nyt tilaukset ovat kuitenkin helpommin ohjelmallisesti saavutettavassa muodossa, joten ne olisi helppo integroida automatisoituun tapahtumaketjuun, missä ympäristöt pystyttyisivät tilausten myötä.

Mahdollisia jatkokehityksen kohteita löytyy ominaisuuksien lisäksi myös itse kehitysprosesseista. DevOpsin julkaisutyökalujen käyttöönotto tehostaisi prosesseja. Yksikkötestien ja julkaisutapahtumaketjujen avulla uusien ominaisuuksien tuotantoonvienti nopeutuisi.

Palvelinkaluston päivittäminen tapahtuu kätevästi Azuren portaalista, kun tilanne sen vaatii. En näe resurssien lisäämiseen tällä hetkellä tarvetta, koska latausajat ovat nopeat ja monitorointitaso näyttää vain murto-osan verran käyttöä varatulle prosessointikapasiteetille.

5.4 Henkilökohtaiset tavoitteet

Opin paljon Azuresta ja moderneista ohjelmistokehitystekniikoista sekä -arkkitehtuurista. Oli mielenkiintoista päästä heti tuotantokäyttöön tulevan ohjelmiston kehitykseen mukaan, alusta loppuun. Sain hyvää opastusta vanhemmilta kehittäjiltä ja arvokasta kokemusta, jota voin hyödyntää tulevaisuuden asiakasprojekteissa.

Projektin päättymisen myötä sain vakituisen, toistaiseksi voimassaolevan, työsopimuksen Riihisoft Oy:ltä. Toimenkuvaani kuuluu tulevaisuudessa Riihi DMA -palveluiden kehittäminen ja Umbraco-konsultointi.

Onnistuin mielestäni aika hyvin. Pysyin toteuksen osalta lasketussa aikataulussa ja palvelu oli heti käytössä. Sovellus helpotti myös mitattavasti työntekoa, mikä oli yrityksen asettama päätavoite projektille.

Lähteet

Django Software Foundation. (2022). *FAQ*. Viitattu 16.4.2022. https://docs.djangoproject.com/en/4.0/faq/general/#django-appears-to-be-a-mvc-framework-but-you-call-the-controller-the-view-and-the-view-the-template-how-come-you-don-t-use-the-standard-names

Eurostat. (2021). *Cloud computing - statistics on the use by enterprises*. https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Cloud_computing_-_statistics_on_the_use_by_enterprises

Greif, S. (2021). *State of JS 2021*. https://2021.stateofjs.com/en-US/libraries/front-end-frameworks

Martin, R. (2018). *Clean Architecture: A Craftsman's Guide to Software Structure and Design (Robert C. Martin Series)*. Pearson.

Microsoft Corporation. (2021). *DP-200T01-A - Implementing an Azure Data Solution*. Microsoft Corporation.

Microsoft Corporation. (2022). *Azure Pricing Calculator*. Viitattu 16.4.2022 https://azure.microsoft.com/en-us/pricing/calculator/

Payne, T. (2017). *TESTING THE PERFORMANCE OF QUERYING UMBRACO*. https://skrift.io/issues/testing-the-performance-of-querying-umbraco

Q-Success. (2022a). *Historical yearly trends in the usage statistics of server-side programming languages for websites*. Viitattu 16.4.2022. https://w3techs.com/technologies/history_overview/programming_language/ms/y

Q-Success. (2022b). *Historical yearly trends in the usage statistics of client-side programming languages for websites*. Viitattu 16.4.2022. https://w3techs.com/technologies/history_overview/client_side_language/all/y

Q-Success. (2022c). Usage statistics of JavaScript libraries for websites. Viitattu 16.4.2022. https://w3techs.com/technologies/overview/javascript_library

Stack Overflow. (2020). *Survey*. https://insights.stackoverflow.com/survey/2020#technology

Stack Overflow. (2021). *Survey*. https://insights.stackoverflow.com/survey/2021#technology

TIOBE Software BV. (2022). *TIOBE Index for April 2022*. Viitattu 16.4.2022. https://www.tiobe.com/tiobe-index/

Liite 1: Riihi DMA -asiakaslomake

11.7.2018
v3.00

Riihi
Device Management from Azure

Asiakastiedot
käyttöönottoa varten

Jälleenmyyjä
(Loppuasiakas)

11.7.2018
v3.00

1 YLEISTÄ

Tällä lomakkeella kerätään perustiedot uutta Riihi DMA ympäristöä varten. Täytetty lomake liitteineen toimitetaan osoitteeseen [REDACTED]

2 ASENNETTAVAT PAKETIT

Paketti	Asennetaanko (X=asennetaan)	Info
7-Zip	x	
Adobe Air		
Adobe Flash Player for Firefox	x	
Adobe Flash Player for IE	x	
Adobe Reader (englanti)		Valitaan englanti tai suomi
Adobe Reader (suomi)	x	Valitaan englanti tai suomi
Adobe Shockwave Player	x	
Autodesk DWG TrueView		
BitLocker	x	Salasanatoive tekstiviestinä numeroon [REDACTED]
Chrome	x	
CiscoMeeting		
Citrix Receiver		
Firefox (englanti)		Valitaan englanti tai suomi
Firefox (suomi)	x	Valitaan englanti tai suomi
FortiClientVPN	x	
F-Secure	x	
Java	x	
KeePass	x	
LenovoSystemUpdate		
Lime CRM		
M-Files		
Microsoft Office 365 Business Premium	x	
Microsoft Office 365 ProPlus		
Microsoft Monitoring Agent		
Notepad++	x	
OpenVPN		
ownCloud	x	
PDFCreator	x	
Riihi Service Request	x	
Salesforce Chatter		
Silverlight	x	
Slack	x	
Team Viewer Host	x	Salasanatoive tekstiviestinä numeroon [REDACTED]
Visio 365		
VLC	x	

11.7.2018
v3.00

3 RIIHI SERVICE REQUEST (PALVELUPYYNTÖ)

Pyydetty tieto	Syötä tieto	Info
Sähköpostiosoite, johon palvelupyyntö lähetään		
Asiakaspalvelun puhelinnumero		
Käyttöliittymän vapaamuotoinen teksti	"Toimivien IT-palveluiden asiantuntija"	Esimerkiksi: "Meiltä parasta palvelua päivin öin!"
Käyttöliittymän www-linkit (1-3 kpl)		Kaksi linkeistä on kytketty logoihin ja yksi näkyy tekstinä. Saman linkin voi halutessaan laittaa kaikkiin kolmeen kohtaan. Esimerkiksi www.yritys.fi.

Käyttöliittymän logo (hyvälaatuinen & läpinäkyvä) lähetetään sähköpostin liitteenä osoitteeseen [REDACTED]

4 YHTEYSTIEDOT

Pyydetty tieto	Syötä tieto
Tekninen yhteyshenkilö (nimi, sähköposti, puhelinnumero)	
Hallinnon yhteyshenkilö (nimi, sähköposti, puhelinnumero)	
Laskutuksen yhteystiedot	