TAMPEREEN
AMMATTIKORKEAKOULU

MODERNI WEB-KEHITYS

Antti Virtanen

Opinnäytetyö
Huhtikuu 2017
Tietotekniikka
Ohjelmistotekniikka

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietotekniikka
Ohjelmistotekniikka

Virtanen Antti
Moderni web-kehitys

Opinnäytetyö 30 sivua, joista liitteitä 0 sivua
Huhtikuu 2017

Tämä opinnäytetyö on tehty projektiluontoisena työnä ja sille toimi tilaajana Suomen mainosmediat Oy.

Työn pääasiallinen tavoite oli automatisoida yrityksen manuaalisesti suorittamat prosessit. Työssä painona oli erityisesti mahdollisten uusien asiakkaiden seulonta ja tarjoaminen yritykselle. Yrityksessä asiakkaiden kartoittaminen on hoidettu manuaalisesti ja työ tilattiin tämän ongelman poistamiseksi.

Työn lopputuloksena onnistuttiin luomaan helppokäyttöinen telemarkkinointi sovellus, jolla yritys voi etsiä ja suodattaa mahdollisia asiakkaita. Sovellus käyttää useita avoimia rajapintoja hyödykseen, joista se myös on riippuvainen, esimerkkinä Patentti- ja Rekisterihallituksen tarjoamat rajapinnat. Osan tiedoista sovellus louhii erinäisten numerohakemistojen kautta. Sovelluksen lopputuotteena syntyy .xlsx –muotoinen tiedosto, joka voidaan viedä yrityksen soittojärjestelmään.

Asiasanat: Rajapinta, xlsx

ABSTRACT

Tampere University of Applied Sciences
ICT
Software engineering

Antti Virtanen
Modern web development

Bachelor's thesis 30 pages, appendices 0 pages
March 2017

This is a project based thesis and it was ordered by Suomen Mainosmediat Oy.

The objective was to automatize company's current manual processes. The main goal of the thesis was to implement a software which will be able to supply new potential customers to company. Currently customer filtering is handled by several employees and this work was ordered to eliminate that.

Outcome of the work was a user-friendly telemarketing software on which company can search and filter potential customers. Software uses several different open source application programming interfaces, for example Finnish Patent and Registration APIs. Some of the data that software offers is scraped from different directory assistances.

Key words: API, scraping

SISÄLLYS

1 JOHDANTO 7
1.1 Työn tavoite 7
1.2 Työn rakenne 7
2 OHJELMOINTIPROSESSI 8
2.1 Yleisesti 8
2.1.1 Tarve analyysi 8
2.1.2 Määrittely ja laatuvaatimus 8
2.1.3 Arkkitehtuuri 8
2.1.4 Toteutus 9
2.1.5 Testaus 9
2.1.6 Dokumentointi 9
2.1.7 Ylläpito 9
2.2 Työnkulku 10
2.2.1 Kehitysvaihe 10
2.2.2 Kehitysympäristö 10
2.2.3 Integroitu ohjelmointiympäristö(IDE) 11
2.2.4 Testaus 12
2.2.5 Tuotantovaihe 13
2.2.6 Ympäristömuuttujat 13
2.2.7 Resurssien pienennys ja pakkaus 14
2.2.8 Julkaiseminen 15
3 OHJELMOINTIKIELET 16
3.1 PHP 16
3.2 JavaScript 17
3.3 HTML 18
4 TEKNOLOGIAT 19
4.1 CSS & Sass 19
4.2 Laravel 19
4.3 Angular 21
5 RAJAPINNAT 23
5.1 Patentti- ja rekisterihallitus 23
6 SOVELLUS 25
6.1 Kirjautuminen 25
6.2 Näkymät 26
7 LOPPUSANAT 28
LÄHTEET 29

LIITTEET 30

LYHENTEET JA TERMIT

SPA	Single-Page Application, web-palvelu tyyppi jossa kontrolli pysyy koko ajan yhdellä sivulla. Sivun uudelleenlataukset poistettu tai minimoitu.
REST	Representational State Transfer, www-sovelluspalvelu millä mahdollistetaan systeemeiden ja tietokoneiden yhteentoimivuus internetin välityksellä.
Angular	Googlen kehittämä JavaScript-ohjelmistokehys
XAMPP	Alustariippumaton www-palvelin pino.
LAMP	Linuxille suunniteltu www-palvelin pino.
PHP	Hypertext Preprocessor, ohjelmointikieli.
Laravel	Avoimen lähdekodiin PHP-ohjelmistokehys
MySQL	Relaatiotietokantaohjelmisto
Git	Versionhallintaohjelmisto
Tietokanta migraatio	Tietokantojen versionhallintaratkaisu, mahdollistaa vaivattoman tietokanta mallien jaon ja käyttöönoton
Vagrant	Virtuaaliympäristöjen hallintaan ja rakentamiseen suunniteltu työkalu
Composer	Paketinhallintajärjestelmä PHP-ohjelmointikielelle
PSR-2	PHP Standard Recommendation, määrittelee esimerkiksi yleiset nimeämiskäytännöt ja koodin muotoilun
Jasmine	Kehys riippumaton yksikkötestaukseen suunniteltu työkalu JavaScript ohjelmille
Karma	Komentorivityökalu, mahdollistaa serverin käynnistämisen ja testien ajamisen.
PHPUnit	Testauskehys PHP-ohjelmointikielelle
API	Application Programming Interface, mahdollistaa järjestelmien ja ohjelmistokomponenttejen kommunikoinnin
UglifyJS	JavaScriptin kompressointi työkalu
Webpack	JavaScript moduulipaketoija

1 JOHDANTO

1.1 Työn tavoite

Työn tavoitteena oli luoda asiakkaalle, Suomen Mainosmediat Oy, täysin toimiva web-ympäristössä toimiva sovellus, jolla yritys voi hallita ja seuloa potentiaalisia asiakkaita. Tarkoituksena oli parantaa yrityksen liikevaihtoa ja optimoida resurssien kohdentamista. Käsittelemme tässä dokumentissa työn vaiheita ja yleisesti modernia web-kehitystä.

1.2 Työn rakenne

Aloitamme työn läpi käynnin toisesta kappaleesta, jossa käsittelemme yleisesti ohjelmointikehityksen prosesseja ja määrittelemme kehitykselle tyypilliset vaiheet. Tutustumme myös yleisellä tasolla itse työn vaiheisiin, erityisesti kehitys- ja tuotantovaiheeseen.

Kolmannessa ja neljännessä kappaleessa käsittelemme työssä ja web-kehityksessä yleisesti käytettyjä ohjelmointikieliä ja teknologioita.

Viides ja kuudes kappale käsittelee työssä hyödynnettyjä rajapintoja ja itse sovellusta.

2 OHJELMOINTIPROSESSI

2.1 Yleisesti

Ohjelmiston kehitysprosessi koostuu erinäisistä kehitysvaiheista, jonka lopputuloksena on valmis tuote. Ohjelmiston kehityksessä prosessi metodiikka on kasvattanut suosiotansa nykyaikaisessa kehityksessä. Yksi käytetyimmistä prosessimetodiikoista on Capability Maturity Model(CMM). CMM luomisen taustalla oli NASA:n ongelmat ohjelmistohankkeissa. CMM-malli koostuu erinäisistä kypsyystasoista joita prosessin on täytettävä (TechTarget). Yleisellä tasolla ohjelmiston kehitysprosessi voitaisiin jakaa seuraavaksi esitettyihin vaiheisiin.

2.1.1 Tarve analyysi

Vaiheen tarkoitus on kartoittaa tuotteen halutut ominaisuudet. Vaikkakin useat ohjelmistoprojektit toteutetaan asiakkaan tilaustyönä ja asiakas on pääosittain vastuussa halutuista ominaisuuksista, on tarve analyysin kartoitus silti välttämätön toimenpide, sillä asiakas ei aina tiedä mitä haluaa ja näin voidaan myös ehkäistä ristiriitaiset vaatimukset.

2.1.2 Määrittely ja laatuvaatimus

Määrittelyn tarkoitus on kuvailla täsmälleen ohjelmiston toimintaa. Määrittely voi sisältää suoria toiminnallisia vaatimuksia ja käyttötapauksia, jotka kuvailevat käyttäjätapauksia joita ohjelmiston odotetaan tarjoavan.

2.1.3 Arkkitehtuuri

Ohjelmiston arkkitehtuurilla kuvataan järjestelmää abstraktilla tasolla. Arkkitehtuurin suunnittelussa taustalla on järjestelmän tai tuotteen asettamat vaatimukset ja mahdolliset laajennukset.

2.1.4 Toteutus

Toteutusvaiheen tehtävä on toimeenpanna tarveanalyysin ja määrittelyn asettamat vaatimukset. Toteutusvaiheessa keskeistä on muun muassa kehitysympäristön asennus ja konfigurointi, kustomointi, järjestelmä integraatioiden toteuttaminen ja yksikkö ja integraatio testaus.

2.1.5 Testaus

Testaus vaiheella tarkoitetaan yleisesti järjestelmätestausta ja asiakkaan itse tekemiä manuaalisia testejä, että käyttöliittymätestausta.

2.1.6 Dokumentointi

Dokumentoinnin tehtävä on dokumentoida järjestelmän tai tuotteen sisäisestä suunnittelusta ja ominaisuuksista. Dokumentointi on erityisen tärkeää ylläpidolle ja mahdollisille laajennuksille.

2.1.7 Ylläpito

Tuotteen ylläpitoon kuuluu olennaisesti bugien ja virheiden korjaaminen. Olennaista on myös uusien toimintojen lisäys

2.2 Työnkulku

Työnkulku perustui pitkälti edellä mainittuihin vaiheisiin. Asiakkaalla oli selkeä visio tuotteen toiminnasta ja ominaisuuksista, joten näitä vastaavien tekniikoiden löytäminen oli suhteellisen helppoa. Käymme seuraavaksi läpi työn kehitys- ja tuotantovaiheita.

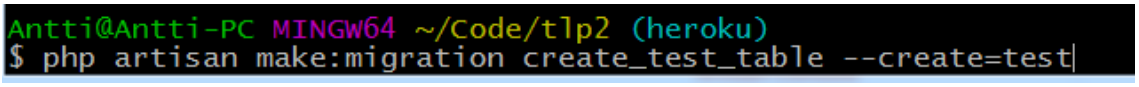
2.2.1 Kehitysvaihe

Kehityksen alussa valittiin spesifikaatioita vastaavat teknologiat. Koska palvelun haluttiin toimivan web-ympäristössä päätettiin asiakasohjelma toteuttaa SPA-pohjaisella ratkaisuilla. SPA-pohjaisilla ratkaisuilla saavutetaan huomattavasti käyttäjäystävällisempiä palveluita, sillä sivujen päivitykset ovat pyritty poistamaan tai vähentämään niitä huomattavasti. Yleisimpiä SPA-pohjaisia ratkaisuja on muun muassa Angular, Ember ja Backbone, tässä tapauksessa päädyimme Angulariin ja sen uudempaan versioon Angular 2:seen. Palvelinpuolen ratkaisuna toimi PHP:n päällä toimiva kehys, Laravel. Palvelinpuolen pääasiallinen tehtävä oli tarjota päätepisteitä asiakaspään toteutuksella, joten toteutusta voidaankin luonnehtia REST tyyppiseksi rajapinnaksi.

2.2.2 Kehitysympäristö

Kehitysympäristönä toimii web-kehityksessä tyyppilliseen tapaan kehittäjän paikallinen asema. Hyvin yleisesti käytettyjä ympäristöjä on muun muassa XAMPP tai LAMP pino. XAMPP pino koostuu Apache HTTP Serveristä, MySQL tietokannasta ja PHP tai Perl ohjelmointikielestä. Muita ratkaisuja on virtuaalinen kehitysympäristö, jossa itse serveri ja projekti pyörivät virtuaaliympäristössä. Virtuaaliympäristön etuna on erityisesti se, että kehittäjän oma ympäristö ei roskaannu ylimääräisistä kannoista ja riippuvuuksista. Jokainen projekti pyörii omassa ympäristössänsä, joten esimerkiksi tietokantojen nimien yhteentörmäyksiltä vältytään. Toinen mainittava etu on ympäristön konfiguraatioiden jakaminen kehitystiimin välillä, jolloin kaikki tiimin jäsenet työskentelevät varmasti samoilla konfiguraatioilla ja vältytään ”kyllä se toimi PHP 5.1:sellä” -tyyppisiltä tilanteilta. Itse työ toteutettiinkin Laravel Homestead ympäristössä, joka on Vagrant pohjainen virtuaaliympäristö. Mitä suuremmaksi kehitystiimi kasvaa, sitä tärkeämpi kehitysympäristöjen yhtenäinen konfiguraatio on. Muita työkalu, joilla tätä voidaan edistää on esimerkiksi tietokanta migraatiot. Tietokanta migraatioita voidaan pitää tietokantojan versionhallintana(ks. Git). Migraatioiden ideana on, että tiimi pystyy

muuttamaan ja jakamaan projektin tietokanta skeemoja ja näin suorittamaan niitä yksinkertaisten komentojen avulla. Kuvassa yksi ja kaksi on havainnollistettu Laravel migraatioiden ajo Artisan nimisen komentoliittymän kautta.



KUVA 1. Taulun luominen migraatio työkalulla

KUVA 2. Migraatio tiedostojen ajaminen

2.2.3 Integroitu ohjelmointiympäristö(IDE)

Integroidun ohjelmointiympäristön tarkoituksena on helpottaa kehittäjän työtä. Triviaalina esimerkkinä yksinkertaisen C-ohjelman luominen. Käytettäessä kehitintä, kehitin tarjoaa valmiin tekstieditointi ympäristön, kielikohtaisilla korostuksilla. Ohjelma voidaan ajaa suoraan kehittiminen kautta yhdellä painalluksella. Toistettaessa sama ilman kehitintä, tarvitsee kehittäjä useita eri työkaluja. Kulku olisi seuraanvanlainen: tiedoston luonti tekstieditorilla, kääntäjän eksplisiittinen kutsuminen komentorivin kautta ja viimeisenä juuri luodun exe-tiedoston ajo komentorivin välityksellä. Kehittimet tarjoavat myös huomattavasti optimoidummat debug-työkalut.

Työssä kehittimenä toimi JetBrainssin PhpStorm. Syynä kyseisen kehittiminen valintaan oli muun muassa Laravel ja TypeScript tuki ja näin ollen se mahdollisti sekä asiakaspään että serveripään toteuttamisen samalla kehittimellä. PhpStormissa on myös sisäinen versionhallinta integraatio, komentorivityökalut, Vagrant ja Composer tuki ja monia muita ominaisuuksia. Työskennellessä suuremmassa tiimissä, itse kehitin ei ole niin suuressa roolissa. Periaatteessa kukin kehittäjä voi työskennellä mieluisallansa kehittimellä, kunhan koodin muotoilu on yhtenäinen. Esimerkiksi PHP:ssa käytetään yleisesti PSR-2 muotoilua, tämä siksi että versionhallinnassa nähdään muutokset selkeästi ja koodi pysyy yhtenäisenä.

2.2.4 Testaus

Testauksen tarkoituksena on suunnitelmallinen virheiden etsintä ohjelmasta tai sen jostakin osasta. Ohjelmistotestaus voidaan jakaa karkeasti kahteen eri osaan, manuaaliseen ja automatisoituun testaukseen.

Manuaalinen testaus, nimensä mukaisesti suoritetaan ilman minkään näköisiä automatisoituja työkaluja. Testaaja suunnittelee, suorittaa ja raportoi testitapaukset ohjelman eri osioille. Manuaalinen testaus on aikaa ja resursseja kuluttavaa, mutta silti suurin osa testauksista suoritetaan manuaalisesti.

Automatisoiduissa testeissä käytetään apuna siihen tarkoitukseen suunniteltuja työkaluja. Automatisoidut testit myös poistavat rajoitukset joita manuaalinen testaus luo. Esimerkkinä tapaus, jossa sivustoa pyritään kuormittamaan esimerksi 100 000 pyynnöllä, on huomattavan hankala toteuttaa ilman automatisoituja työkaluja.

Yleisimmät testaustasot ovat yksikkö- ja integraatiotestaus. Yksikkötestauksessa kehittäjä testaa luotua moduulia tai jopa pienempää yksikköä kuten yksinäistä funktiota, todetakseen sen olevan virhevapaa. Onnistuneen yksikkötestauksen ja teknisen määrittelyn mukaisen suunnittelun tuloksena voidaan todeta, että ohjelman osa on vaatimuksien mukainen ja virhevapaa.

Integraatiotestauksen tavoitteena on testata ohjelman eri komponenttien ja rajapintojen välistä toimintaa.

Työssä suoritettu testaus jäi melko marginaaliseksi tiukan aikataulun vuoksi. Asiakaspäähän kirjoitettiin muutama testi hyödyntäen sekä Jasmine -kehystä, että Karma -kehystä. Palvelinpuolen testaus suoritettiin PhpUnitilla.

2.2.5 Tuotantovaihe

Käymme tässä läpi projektirakenteen ja ympäristölliset muutokset siirryttäessä kehitysvaiheesta tuotantovaiheeseen.

2.2.6 Ympäristömuuttujat

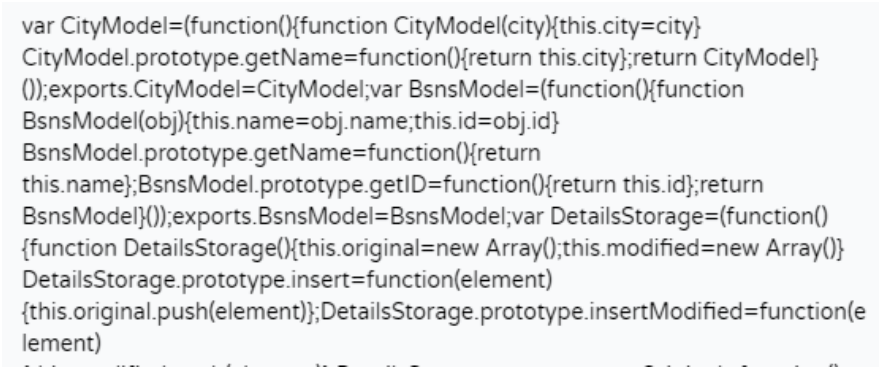
Tyypillisesti ympäristömuuttajat sijaitsevat erillisessä tiedostossa kuten esimerkiksi .env -nimisessä tiedostossa. Ympäristömuuttujien tarkoituksena on arkaluontoisen tiedon eristäminen suoraan lähdekoodista, esimerkkinä vaikka sisäisen API-avaimen määritys. Toinen tehtävä on yleisten konfiguraatioasetuksien määrittäminen. Tämänkaltaisia asetuksia on muun muassa: debug-tilan, aikavyöhykkeiden, kehitystilan sekä tietokanta-asetusten määrittäminen. Siirryttäessä kehitysvaiheesta tuotantovaiheeseen, yleisimmät muutokset ympäristömuuttujissa koskevatkin juuri debug-tilaa ja sen hetkistä kehitystilaa. Myös erillisten .env (kuva 3) tiedostojen käyttäminen on yleistä, joka tarkoittaa että sekä kehitysvaiheella että tuotantovaiheella on omat tiedostonsa, kuten .env ja .env.local. Työssä käytetty Laravel hyödyntää kolmannen osapuolen kirjastoa PHP dotenv:ia, joka mahdollistaa yhden konfiguraatiotiedoston käytön useiden sijasta.

```
APP_ENV=local
APP_DEBUG=true
APP_KEY=base64:XXX
APP_URL=XXX
APP_LOG_LEVEL=debug

DB_CONNECTION=mysql
DB_HOST=XXX
DB_PORT=3306
DB_DATABASE=XXX
DB_USERNAME=XXX
DB_PASSWORD=XXX

CACHE_DRIVER=file
SESSION_DRIVER=file
QUEUE_DRIVER=sync

REDIS_HOST=127.0.0.1
REDIS_PASSWORD=null
REDIS_PORT=6379

MAIL_DRIVER=smtp
MAIL_HOST=mailtrap.io
MAIL_PORT=2525
MAIL_USERNAME=null
MAIL_PASSWORD=null
MAIL_ENCRYPTION=null
```

KUVA 3. .env -tiedosto

2.2.7 Resurssien pienennys ja pakkaus

Palvelulle on erityisen tärkeää, että se on optimoitu mahdollisimman hyvin ja turha vasteaika on poistettu. Resurssien pienentäminen ja pakkaaminen pyrkivät edistämään juuri tätä. Pienentämisellä tarkoittaa prosessia jossa turha ja ylimääräinen data pyritään poistamaan tiedostosta, muuttamatta tiedoston toiminnallista rakennetta. Ylimääräiseksi dataksi nähdään muun muassa kommentit, rivinvaihdot, välilyönnit ja ylimääräiset välimerkit. Minimoimisella pyritään siis pienentämään tiedoston kokoa, jotta sivuston lataaminen nopeutuisi. Käytetyimpiä työkaluja minimoimisiin on muun muassa JSMin ja työssä käytetty UglifyJS. Kuvassa neljä JavaScript koodin pätkä minimoituna.



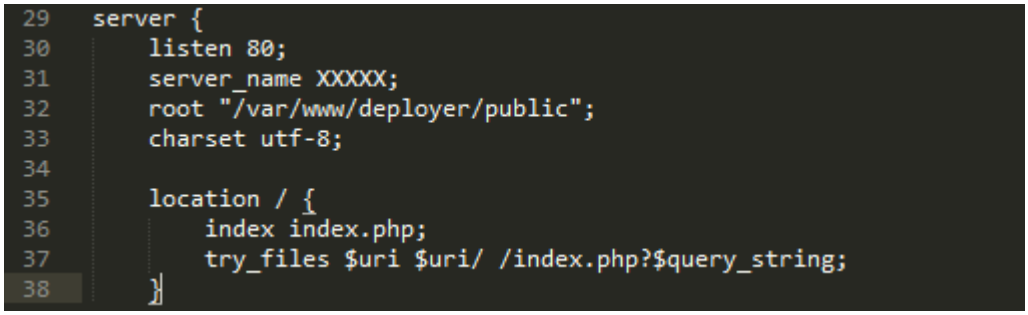
KUVA 4. Pienennetty JavaScript -tiedosto

Toinen toimenpide on resurssien pakkaus. Tyypillisesti projekti saattaa sisältää satoja JavaScript tiedostoja ja toiset samanlaiset tyylitiedostoja. Tämä tarkoittaa serverin kuormituksen kannalta sitä, että sivua ladattaessa jokainen tiedosto lähetetään erikseen ja sen hakemiseen tarvitaan oma kutsunsa. Toisin sanoen, tilanne jossa etusivu on esimerkiksi riippuvainen kymmenestä JavaScript –tiedostosta, käyttäjän avatessa sivu, sivusto tekee kymmenen erillistä HTTP-kutsua palvelimelle. Optimaalisempi tilanne olisi tilanne, jossa skripti tiedostot on pakattu yhdeksi tiedoksi, esimerkiksi bundle.js nimiseksi tiedostoksi ja samainen toimenpide tehty tyylitiedostoille. Nyt käyttäjän ladatessa sivu, sivun ainut riippuvuus on yksi tiedosto, joten kymmenen pyynnön sijaan tehdäänkin vain yksi.

Tähän tarkoitukseen on suunniteltu erilaisia työkaluja. Esimerkkinä työssä käytetty Webpack.

2.2.8 Julkaiseminen

Projektin julkaiseminen on yksinkertaistettuna tilanne, jossa projektiympäristö vaihtuu kehittäjän paikallisesta asemasta julkiseksi asemaksi. Siirtymän helpottamiseksi on luotu useita kolmannen osapuolen kirjastoja kuten PHP:ssa yleisesti käytetty Deployer. Deployer voidaan asentaa joko Curlin tai Composerin välitykselle. Asennuksen jälkeen suoritetaan alustus. Alustus luo uuden tiedoston nimeltä deploy.php. Tiedosto sisältää valmiita serverikonfiguraatioita, mutta se vaatii myös lisä konfiguroimista. Pakolliset muutokset ovat projektin säilytyspaikan, serveriasetusten, kuten osoitteen, sijainnin ja etäyhteysasetusten määrittäminen (Deployer). Viimeisenä vaiheena on riippuen käytetäänkö Apache vai nginx pohjaista serveriä, .htaccess tai nginx.conf (kuva 5) tiedoston konfigurointi. Konfigurointi pitää sisällään minimissään portin, domainin, ja projektin sijainnin määrittämisen. Ohessa kuva esimerkki konfiguroinnista.



KUVA 5. nginx.conf

3 OHJELMOINTIKIELET

Tässä osiossa käymme läpi työssä käytetyt ohjelmointikielet.

3.1 PHP

PHP on palvelinpuolen komentosarjakieli, joka on erityisesti suunniteltu toimimaan web-palvelinkehityksessä (Luke Welling). Tyypillinen käyttötapaus on tilanne, jossa PHP koodia on upotettuna HTML sivun sekaan. Upotettu koodi (kuva 6) suoritetaan joka kerta kun sivulla vieraillaan. Erona asiakaspuolen kieliin on, että PHP-koodi tulkitaan palvelimen päässä, joten käyttäjällä ei ole mahdollisuutta nähdä tehtyä toteutusta edes kehittäjätyökalujen kautta.

```
<section>
    <div class="row">
        <div class="col-xs-12">
            <h1> Numerot 1-100 </h1>
            <?php for( $i=0; $i<=100; $i++) { ?>
                    <p> <?= $i ?> </p>
            <?php  } ?>
        </div>
    </div>
</section>
```

KUVA 6. PHP-koodia upotettuna HTML sivuun

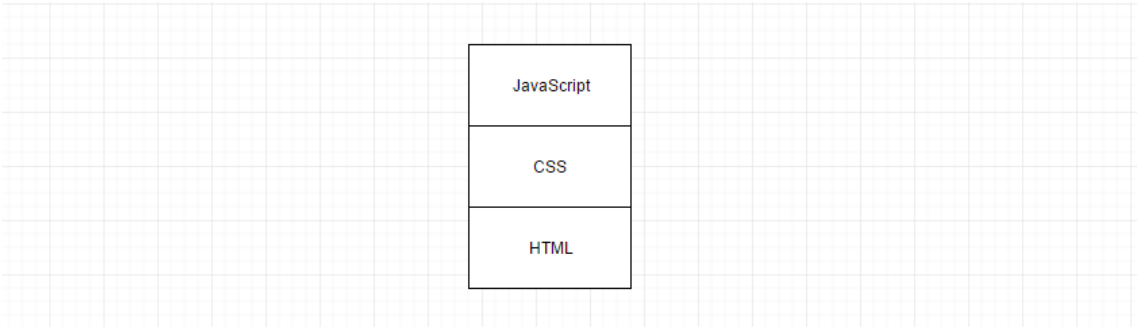
PHP tukee monen muun kielen tavoin oliomaisuutta. Tämä tarkoittaa sitä että kehittäjällä on mahdollisuus luoda olioita, jotka koostuvat muuttujista ja funktioista. Oheisessa kuvassa yksinkertaisen olion rakentaminen.

```
class MyClass {
  private $value;

  function __construct(value) {
    $this->value = value;
  }

  public function getValue() {
    return $this->value;
  }

  public function setValue(value) {
     $this->value = value;
  }
}

$myClass = new MyClass(100);
```

KUVA 7. Luokka ja olion luominen

3.2 JavaScript

JavaScript on ohjelmointikieli joka mahdollistaa dynaamisten ominaisuuksien luomisen web-sivulle. Nykyaikaiset web-sivustot voidaan karkeasti jaotella kahteen tyyppiin, dynaamisiin ja staattisiin sivuihin. Staattiset sivut ovat rakennettu puhtaasti HTML:n ja CSS:n avulla joten sivuston ja käyttäjän interaktio on hyvin vähäistä. Dynaamisissa sivuissa on hyödynnetty yhtä tai useampaa skriptaus kieltä, kuten esimerkiksi JavaScriptia. Dynaamiseksi ominaisuudeksi voidaan nähdä esimerkiksi niinkin yksinkertainen tapaus, kun muuttuvan ajan näyttö tai yksinkertainen 2D tai 3D animaatio (Mozilla Developer Network).



KUVA 8. Tyypillisen www-sivuston rakenne

JavaScript on myös oliopohjainen kieli, mutta se ei tue suoranaisesti luokkien rakentamista. JavaScriptissä luokat rakennetaan erillisten rakentaja funktioiden kautta ja perintä hoidetaan prototyyppiketjun kautta. Metodien luontiin on useita vaihtoehtoja, joista yksi on edellä mainittu prototyyppiin pohjautuva. Kuvassa yhdeksän esimerkki JavaScript luokan ja olion luonnista muutamalla eri tavalla.

```
// Luokkatyylillä, aito private muuttuja
function MyClass(value) {
  var value = value;

  this.getValue = function() {
    return value;
  }

  this.setValue = function(val) {
    value = val;
    return this;
  }
}

var myClass = new MyClass(100);

console.log(myClass.getValue());    // -> 100
console.log(myClass.setValue(42).getValue()); // -> 42
```

```
// Prototyypit ja 'private' muuttuja
function MyClass(value) {
  this._value = value;
}

MyClass.prototype.setValue = function(value) {
  this._value = value;
  return this;
}

MyClass.prototype.getValue = function(value) {
  return this._value;
}

var myClass = new MyClass(100);

console.log(myClass.getValue());    // -> 100
console.log(myClass.setValue(42).getValue()); // -> 42
```

```
// Perinteinen toteutus
var myClass = {
  value: 100,
  setValue: function(val) {
    this.value = val;
    return this;
  },
  getValue: function() {
    return this.value;
  }
}

console.log(myClass.getValue())   // -> 100
console.log(myClass.setValue(42).getValue())  // -> 42
```

KUVA 9. Ylhäällä olion luominen rakentaja funktion kautta, alhaalla perinteinen tapa

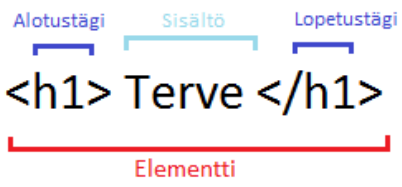
3.3 HTML

HTML, HyperText Markup Language, on deklaratiivinen ohjelmointikieli. HTML:n tehtävä on puhtaasti sivuston näkymän kuvailu, sen vastuuseen ei kuulu sivuston renderöinti tai minkäänlaisen käyttäjäinteraktion tarjoaminen. HTML käyttää XML:n kaltaista merkkausta (kuva 10) kuvien, tekstien ja muiden sisällön rakentamiseen.

```
    <div class="row">
        <div class="col-xs-12">
            <customer-table></customer-table>
        </div>
    </div>
```

KUVA 10. HTML merkkausta

Tyypillistä HTML elementeille on seuraavat piirteet. Aloitustägi, joka sisältää aina elementin nimen. Aloitustägi merkitään kulmasulkeiden sisään, tarkoituksena osoittaa elementin alkukohta. Lopetustägi, aloitustägin kaltaisesti merkkaus on identtinen, lukuunottamatta kauttaviivaa elementin nimen edessä. Tägien välissä sijaitsee itse sisältö, joka voi vaihdella tekstistä muihin elementteihin. Itse elementti (kuva 11) muodostuu näistä kolmesta eri tekijästä (Mozilla Developer Network).



KUVA 11. HTML-elementti

4 TEKNOLOGIAT

Käymme tässä osiossa lyhyesti läpi työn keskeisimmät teknologiat.

4.1 CSS & Sass

CSS, Cascading Style Sheets, on tyylikieli jonka tehtävänä on kuvailla HTML dokumentin visuaalinen ulkoasu. Perinteisten tyylitiedostojen ongelmana on kankeus ja itsensä toistavuus. Tämän vuoksi CSS:lle on kehitetty esikääntäjiä, joiden tavoitteina on poistaa em. rajoitteet. Esikääntäjät, kuten Sass, ovat käytännössä skriptauskieliä, jotka tarjoavat uusia ominaisuuksia tyylitiedostojen rakentamiseen. Esikääntäjien ajamiseen tarvitaan erillinen kääntäjä. Kääntäjän tehtävä on kääntää esikäsitelty tiedosto takaisin perinteiseksi tyylitiedostoksi, sillä selain ymmärtää ainoastaan .css -päätteisiä tyylitiedostoja. Kuvassa 12 vertailussa syntaksit. Huomattavaa on, että vasemmanpuoleinen Sass mahdollistaa muuttujien käyttämisen ja tyylien niputtamisen.

```
$bg-color: #f3f3f4;
app {
  #wrapper {
    display:block;
    background-color: $bg-color;
    margin: 0 0 0 5em;
    height: 100%;
  }
  @media(min-width:767px) {
     #wrapper {
        margin: 0 0 0 17em;
     }
  }
}
```

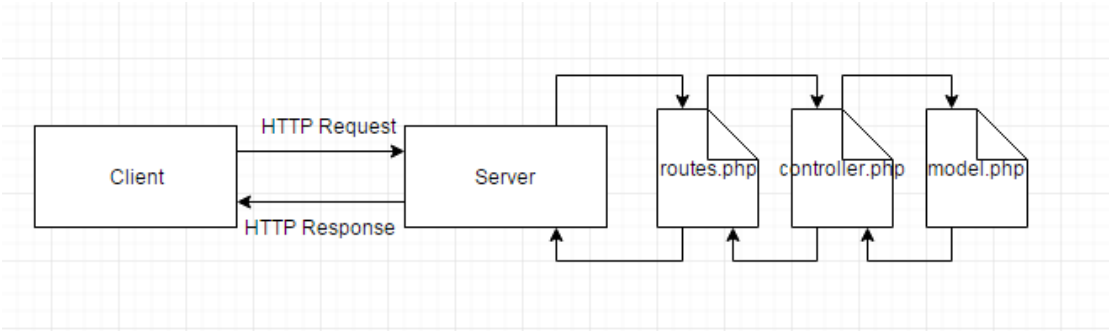
```
app #wrapper {
    display:block;
    background-color: #f3f3f4;
    margin: 0 0 0 5em;
    height: 100%;
}

@media(min-width:767px) {
    app #wrapper {
       margin: 0 0 0 17em;
    }
}
```

KUVA 12. Vasemmalla scss syntaksi ja oikealla perinteinen css

4.2 Laravel

Laravel on Taylor Otwellin kehittämä avoimen lähdekoodin PHP kehys (Laravel). Laravel noudattaa monen muun kehyksen tavoin MVC suunnittelumallia. MVC:n ideana on erotella tietokanta-, bisneslogiikka- ja esitystaso toisistansa. Tämänkaltaisella erottelulla saavutetaan siistimpää ja ymmärrettävämpää koodia. Jotta kehyksen toiminta ja MVC-malli olisi helpompi sisäistää, käymme seuraavaksi läpi koko tapahtumaketjun.

Tilanteena tapaus, jossa käyttäjä lähettää HTTP-pyynnön palvelun yhdestä sivusta (kuva 13).



KUVA 13. HTTP-pyynnön kulku

Käyttäjän avatessa sivu, selain lähettää serverille HTTP-pyynnön. Serverillä kaikki mahdolliset pyynnöt ovat kerätty yhteen tiedostoon nimeltä routes.php. Reititystiedostossa pyyntö välitetään oikealle reitille. Tässä tapauksessa pyynnön tyyppi on GET ja osoitteena "/clients", joten kuvan 14 mukainen reitti on siitä vastuussa.

```
23   Route::get('/clients', 'API\ClientController@index');
```

KUVA 14. Yksittäinen reitti routes.php -tiedostosta

Reititystiedostosta pyyntö välitetään eteenpäin reitin mukaiselle käsittelijällä. Kuten aikaisemmin mainitsimme, palvelinpuolen toteutus on puhtaasti REST tyyppinen, joten käsittelijän palauttama data on JSON tyyppistä. Käsittelijän tehtävänä ei ole tarjota minkäänlaista logiikkaa vaan itse logiikka on toteutettu malliin, jota käsittelijä (kuva 15) hyödyntää.

```
27          public function index($floor = null, $ceil = null) {
28              $companies = Company::with('contactDetails')
29                                  ->with('businesslines')
30                                  ->has('contactDetails');
31                                  ->offset($floor)
32                                  ->limit($ceil)
33                                  ->get();
34              $count = $companies->count();
35              $result = array(
36                  'active' => $count,
37                  'companies' => $companies
38              );
39
40              return response()->json($result, 200);
41          }
```

KUVA 15. Käsittelijä funktio controller.php -tiedostosta

Tässä tapauksessa käsittelijä hyödyntää kahta mallia nimeltä Company ja BusinessLines.

Käsittelijä kutsuu mallien kautta näille toteutettuja metodeita. Mallit (kuva 16) eivät ole tavallisia olioita vaan itseasiassa olio-relaatio-mallinnuksia tietokannasta. Toisin sanoen olion avulla on mahdollista suorittaa tietokanta toimenpiteitä.

```
8   class Company extends Model
9   {
10
11      use SoftDeletes;
12
13      protected $table = 'companies';
14      protected $fillable = ['name', 'businessId', 'liquidations'];
15
16      public function visibilities() {
17          return $this->belongsToMany('App\Models\Visibility')->withTimeStamps();
18      }
19
20      public function contactDetails()
21      {
22          return $this->hasOne('App\Models\ContactDetails', 'id', 'contactDetails');
23      }
24
```

KUVA 16. Kuvakaappaus mallista

Malli palauttaa datan takaisin käsittelijälle, joka parsii siitä JSON-muotoista ja välittää sen takaisin reititystiedostolle joka lopulta palauttaa vastauksen takaisin käyttäjälle.

4.3 Angular

Angular on TypeScript pohjainen avoimen lähdekoodin kehys (Angular.io). Vaikkakin Angularilla on avoimen lähdekoodin lisenssi, pääkehittäjänä toimii Googlen oma Angular tiimi. TypeScript tuen ansiosta Angular mahdollistaa puhtaan olio-ohjelmointi pohjaisen kehityksen (kuva 17), syntaksi muistuttaa etäisesti Java-koodia.

```
6   export class LoginModel implements BaseModel {
7
8       constructor(public email?: string, public password?: string) { }
9
10      public stringify() {
11          let email = this.email;
12          let password = this.password;
13          return JSON.stringify({ email, password });
14      }
15  }
```

KUVA 17. Esimerkki TypeScriptistä

Angularissa on keskeisenä osana komponentti pohjainen ajattelutapa. Yksinkertaistettuna komponentti on HTML pohjainen elementti. Komponentit ovat kapseloituja elementtejä, joilla on oma oletus ilmentymänsä. Kapseloimisille tarkoitetaan tässä tapauksessa sitä,

että komponentit sisältävät oman HTML merkkauksen ja oletustyylit, jotka eivät vuoda komponentin näkyvyysalueelta. Kuten tavallisille elementeillä, komponenteille voidaan määrittää tapahtumakuuntelijoita ja käsittelijöitä. Komponenteilla pyritään luomaan uudelleenkäytettävämpää ja selkeämpää koodia.

5 RAJAPINNAT

Rajapinnat mahdollistavat ohjelmien eri osien keskinäisen kommunikoinnin. Tyypillinen esimerkki on asiakasohjelman ja web-palvelimen välinen kommunikointi, joka demonstroitiin jo kappaleessa 4.2. Edellä mainitun sisäisen rajapinnan lisäksi työssä käytettiin ulkoisia rajapintoja ja käsittelemme tässä niistä merkittävimmässä roolissa olevaa.

5.1 Patentti- ja rekisterihallitus

Patentti- ja rekisterihallitus(PRH) tarjoaa avointa dataa, joka on käytettävissä HTTP-protokollan välityksellä, tiedonsiirtomuotona toimii tyypilliseen tapaan JSON. PRH tarjoaa kahta palvelua, kaupparekisterin kuulutustieto- ja yritystietopalvelua, työssä hyödynnettiin jälkimmäistä. YTJ-tieto sisältää muun muassa seuraavat tiedot: yritys- ja yhteisötunnukset, toiminimen, aputoiminimet, kotipaikan, kunnan, lähiosoitteet, toimialan, yhteistiedot ja kaupparekisterin. Itse rajapinnan käyttö koostuu kahdesta osasta, itse päähausta ja yksityiskohtaisesta yrityshausta. Päähakua on mahdollista konfiguroida rekisteröintipäivämäärän, yrityksen nimen ja muiden parametrien mukaan. Päähaku palauttaa maksimissaan 1000 yritystä kerralla, palautuneessa datasta oleellisinta on generoitu yritysosoite, joka mahdollistaa yrityksen yksityskohtaisemman haun. Tämän lisäksi päähaku palauttaa seuraavan datatuloksen osoitteen eli viitateen kuvaan 18 *nextResultsUri* sisältää osoitteen jossa on seuraavat 1000 yritystä.


```
{
  "type": "fi.prh.opendata.bis",
  "version": "1",
  "totalResults": 4196,
  "resultsFrom": 0,
  "previousResultsUri": null,
  "nextResultsUri": "http://avoindata.prh.fi/opendata/bis/v1?totalResults=true&maxResults=1000&resultsFrom=1000&companyRegistratio
nFrom=1980-01-01&companyRegistrationTo=1981-01-01",
  "exceptionNoticeUri": null,
  "results": [
    {
      "businessId": "0370698-1",
      "name": "Tandklinik A & G Nielsen Ab",
      "registrationDate": "1980-12-15",
      "companyForm": "OY",
      "detailsUri": "http://avoindata.prh.fi/opendata/bis/v1/0370698-1"
    },
    {
      "businessId": "0370723-9",
      "name": "Rakennusliike Asuntoinsinöörit Oy",
      "registrationDate": "1980-12-15",
      "companyForm": "OY",
      "detailsUri": "http://avoindata.prh.fi/opendata/bis/v1/0370723-9"
    },
```


KUVA 18. Rajapinnan palauttama data

Työssä rajapintaa käytettiin seuraavasti. Yrityksien haku alkoi vuodesta 1980, yrityksiä haettiin vuosi kerrallaan, joten esimerkkinä 1980-1981, 1981-1982 ja niin edelleen. Koska hausta palautui aina seuraavan datatuloksen osoite, haku pystyttiin hoitamaan rekursiolla. Mikäli seuraavan datatuloksen osoitteena palautui *null* parsittiin uusi osoite. Parsinta tapahtui nollaamalla *resultsFrom=* ja lisäämällä *companyRegistrationFrom & companyRegistrationTo* vuosia yhdellä (kuva 19).


```
  "totalResults": 4196,
  "resultsFrom": 4000,
  "previousResultsUri": "http://avoindata.prh.fi/opendata/bis/v1?totalResults=true&maxResults=1000&resultsFrom=3000&companyRegistr
ationFrom=1980-01-01&companyRegistrationTo=1981-01-01",
  "nextResultsUri": null,
```

```
http://avoindata.prh.fi/bis/v1?totalResults=true&maxResults=1000&resultsFrom=0&companyRegistrationFrom=1981-01-01&companyRegistrationTo=1982-01-01
```


KUVA 19. Kuvassa parsittu uusi osoite

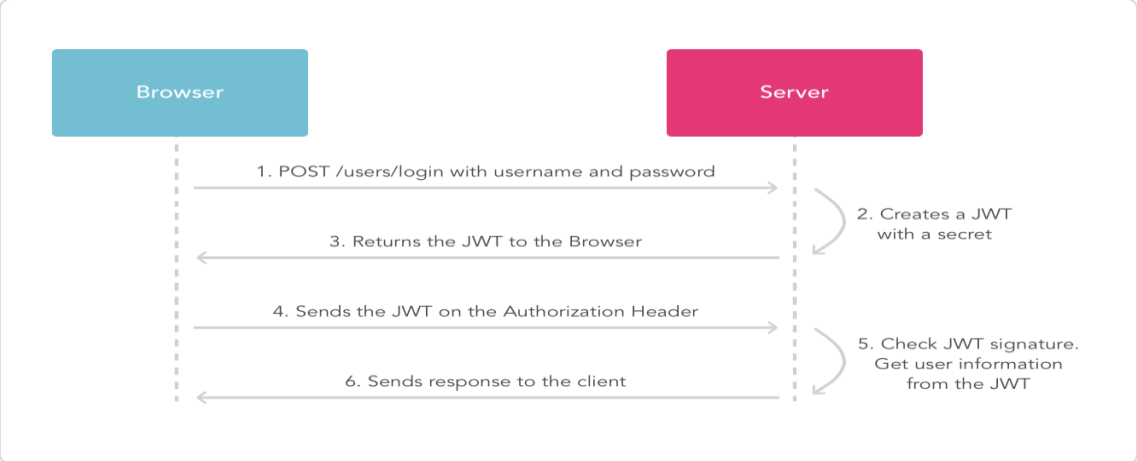
Omat haasteensa toi rajapinnan asettamat rajoitteet ja datan suurehko määrä. Kaiken kaikkiaan yrityksiä vuodesta 1980 vuoteen 2017 on rekisteröity noin 464 tuhatta. Hitautta onnistuttiin poistamaan osittain käyttämällä rekursion lisäksi useampaa säiettä(rinnakkaisuutta).

6 SOVELLUS

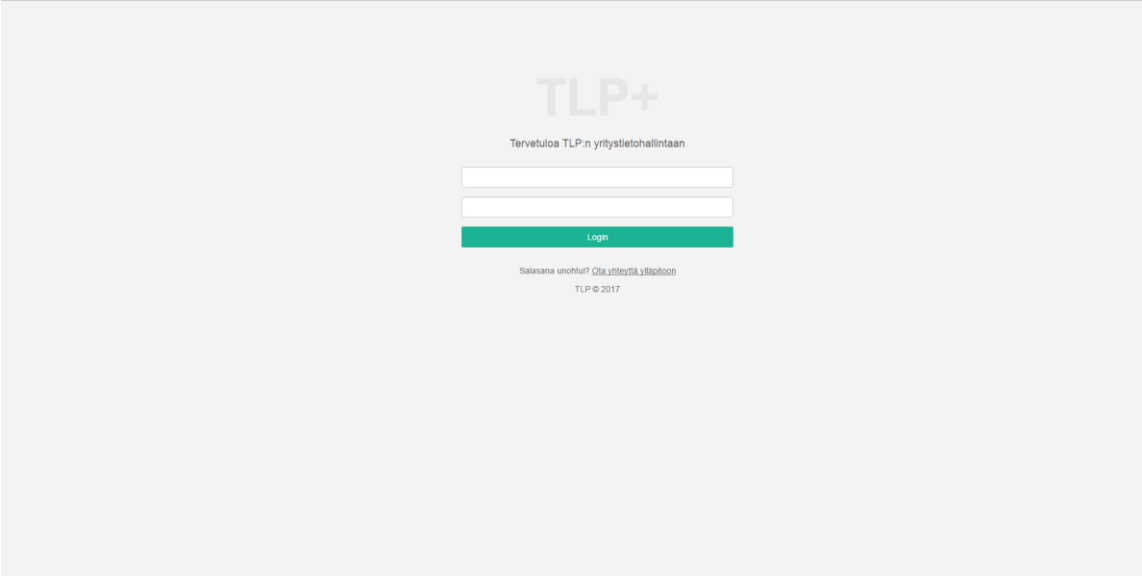
Sovellus osassa käsittelemme sovelluksen toimintaa, suunnittelua ja rakennetta sekä käyttäjän että ohjelmoijan perspektiivistä.

6.1 Kirjautuminen

Suunniteltaessa autentikointia päädytään tyypillisesti joko sessio- tai tokenpohjaiseen ratkaisuun. Merkittävimpänä erona näiden välillä on että sessiopohjainen ratkaisu on tilallinen kun taas tokenpohjainen on tilaton. Tilattomassa (eng. Stateless) menetelmässä käyttäjän tietoja tai minkäänlaista avainta ei tallenneta palvelimen muistiin, joten suurimmissa järjestelmissä tästä saadaan huomattava hyöty. Työssä käytetty tilaton ratkaisu oli nimeltänsä JWT. JWT:ssä käyttäjä autentikoituu palvelimelle, palvelin tarkistaa tiedot ja palauttaa JSON pohjaisen tokenin takaisin (kuva 19). Tokeni tallennetaan tyypillisesti selaimen paikalliseen muistiin, mutta keksien käyttäminen on myös mahdollista. Koska menetelmä on tilaton, täytyy tokeni lähettää jokaisen kutsun yhteydessä (jwt.io).



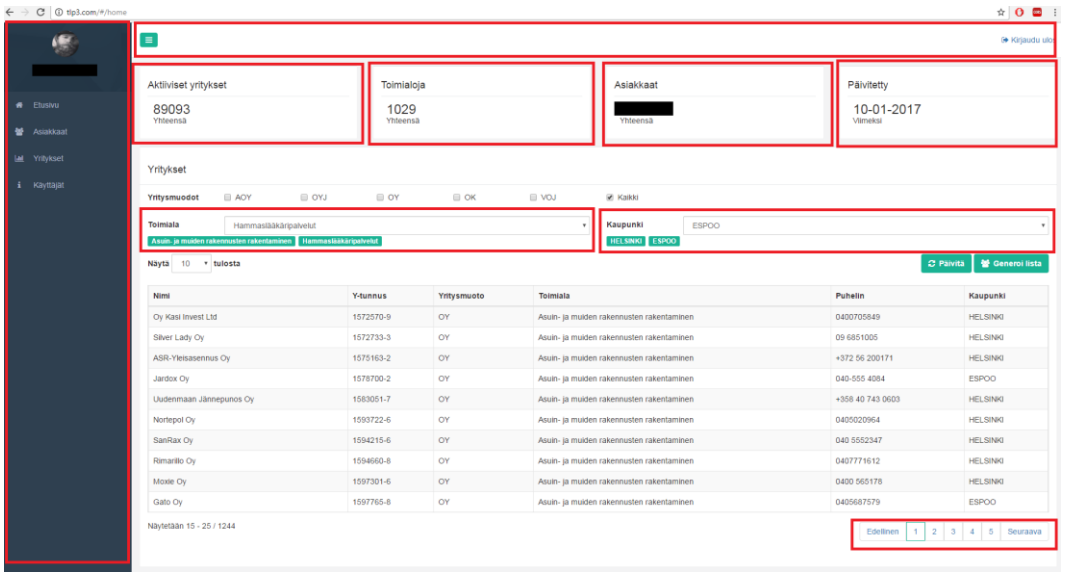
KUVA 19. JWT:n toiminta (jwt.io)



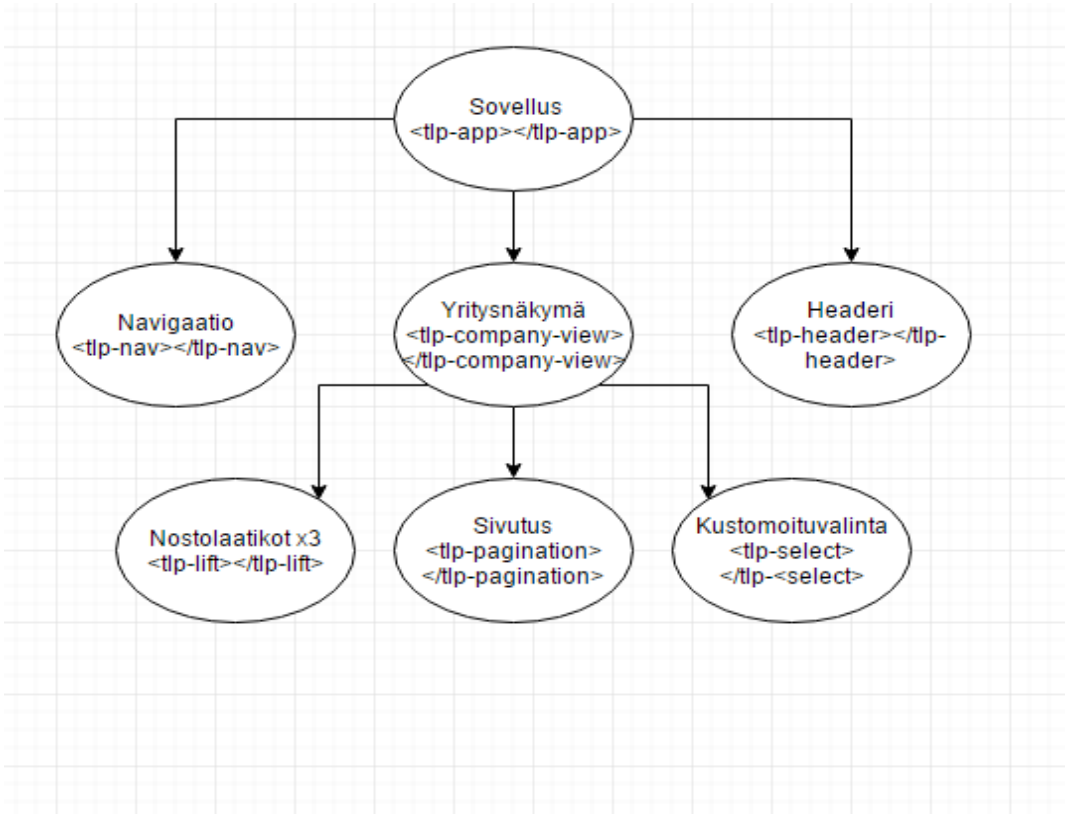
KUVA 20. Sovelluksen kirjautuminen

6.2 Näkymät

Kaikki näkymät perustuvat komponenteista. Komponenttirakenne on hierarkinen (kuva 22) ja se voitaisiin ajatella binääripuun kaltaisena rakenteena, missä solmujen määrä voi vaihdella 0-n:n välillä. Solmuilla voi siis olla lapsisolmuja ja kaikista korkeimmalla hierarkiassa on juurisolmu. Ohessa kuva palvelun yhdestä näkymästä, yksittäiset komponentit ovat rajattu punaisella värillä. Kuten kuvasta 21 voi huomata komponenteiksi on määritelty osia, joita suurella todennäköisyydellä hyödynnetään uudestaan. Komponenttien välinen interaktio hoidettiin joko perinteisesti sitomalla syöte komponentin attribuutteihin tai mikäli kyseessä oli jokin muu tapaus kun lapsi-vanhempi tilanne käytettiin apuna RxJS:n tarjoamia kuuntelijoita.



KUVA 21. Sovelluksen etusivu, komponentit merkitty punaisilla laatikoilla



KUVA 22. Sovelluksen etusivun komponentti hierarkia

7 LOPPUSANAT

Työn lopputuloksena syntyi web-ympäristössä toimiva telemarkkinointia edistävä sovellus. Vaikkakin sovellus tehtiin tilaustyönä asiakkaalle, olisi täysin mahdollista hyödyntää sovellusta myös muissa telemarkkinointi pohjaisissa yrityksissä.

Työn kulku oli odotetun kaltainen ja suurimmilta ongelmilta vältyttiin. Kaikki asiakkaan määrittämät ominaisuudet onnistuttiin toteuttamaan, mutta työ jätti runsaasti tilaa myös jatkokehitykselle. Esimerkiksi sovellusta käyttää ja hallitsee pääasiallisesti myynnistä vastaavat henkilöt, jotka koostavat ja tulostavat soittolistat, mikäli sovellukseen haluttaisiin antaa pääsy myös perus myyjille täytyisi käyttäjähallintaa ja käyttöoikeuksia vielä parantaa.

Kaiken kaikkiaan olen tyytyväinen saavutettuun lopputulokseen. Yleisesti työ antoi hyvän kokonaiskuvan nykyaikaisesta full stack-kehityksestä.

LÄHTEET

TechTarget 2017: Capability Maturity Model. Luettu 1.2.2017
http://searchsoftwarequality.techtarget.com/definition/Capability-Maturity-Model

Deployer 2017: Installation. Luettu 5.2.17
https://deployer.org/docs/installation/

JWT.io 2017: Introduction to JSON Web Tokens. Luettu 10.2.17
https://jwt.io/introduction/

Mozilla Developer Network 2017: HTML. Luettu 15.2.17
https://developer.mozilla.org/en-US/docs/Web/HTML

Mozilla Developer Network 2017: JavaScript. Luettu 15.2.17
https://developer.mozilla.org/en-US/docs/Web/JavaScript

Laravel 2017: Documentation. Luettu 1.3.2017
https://laravel.com/docs/5.4

Angular.io 2017: Angular Docs. Luettu 2.3.2017
https://angular.io/docs/ts/latest/

Luke Welling 2011: PHP and MySQL Web development (4th edition). Luettu 10.3.2017

LIITTEET