Tomi Lammi

MVVM-MALLI JA JAVASCRIPT. CASE: YHTEYSTIETOJEN HALLINTA

Tietojenkäsittelyn koulutusohjelma
2014

samk
Satakunnan ammattikorkeakoulu
Satakunta University of Applied Sciences

MVVM-malli ja JavaScript. Case: Yhteystietojen hallinta

Lammi, Tomi
Satakunnan ammattikorkeakoulu
Tietojenkäsittelyn koulutusohjelma
Tammikuu 2014
Ohjaaja: Nieminen, Hans
Sivumäärä: 33
Liitteitä:

Asiasanat: mvvm, knockout, javascript, html, T-Base

Opinnäytetyön aiheena oli Knockout.js JavaScript-kirjaston esitteleminen T-Base Oy:llä toteutettavan yhteystietohallintajärjestelmän avulla. Projekti on toteutettu web-sovelluksena käyttäen Sharepoint-tekniikkaa palvelinpuolella ja MVVM-mallia asiakaspuolen JavaScript-koodille. Opinnäytetyössä esitelty osuus on vain pieni osa T-Base Oy:n yhteystietojen hallintajärjestelmää.

Opinnäytetyön teoriaosuudessa pohditaan web-ohjelmoinnin nykytilaa, esitellään MVVM-malli, sekä kerrotaan miten malli otetaan käyttöön Knockout-kirjaston avulla. Lisäksi osuudessa verrataan Knockoutilla ja perinteisellä JavaScriptillä toteutettua komentorivikoodia.

Toteutusosuudessa esitellään Knockout-kirjaston perustoimintaperiaatteita toimeksiannon kautta rakentamalla moderni ja responsiivinen käyttöliittymä. Toimeksianto koostuu uusien yhteystietojen ja yrityksien lisäämisestä sekä siihen liittyvästä web-käyttöliittymästä.

MVVM pattern and JavaScript. Case: Contact management

Lammi, Tomi
Satakunnan ammattikorkeakoulu, Satakunta University of Applied Sciences
Degree Programme in Information and Communication Technology
January 2014
Supervisor: Nieminen, Hans
Number of pages: 33
Appendices:

Keywords: mvvm, knockout, javascript, html, T-Base

The purpose of this thesis was showcasing Knockout JavaScript library in a contact management software developed in T-Base Oy. Project is implemented as a web application using Sharepoint for server management and introducing MVVM design pattern for the clientside JavaScript. The presented part of the software in this thesis is only a small part of the final contact management system developed in T-Base Oy.

In the theoretic part of this thesis, I'll contemplate the current situation of web-based software development. I'll also introduce MVVM pattern and explain how it can be utilized in JavaScript programming.

In the concrete part of the thesis, I'll showcase how Knockout is used in developing a modern and responsive user interface. The application showcased in this thesis consists of adding new companies with contacts and how to build the JavaScript logic using Knockout's MVVM pattern for the user interface.

Sisällysluettelo

1 JOHDANTO....5
2 YRITYS JA TOIMEKSIANNON KUVAUS....5
3 MVVM-MALLI....6
3.1 MVVM-mallin historia....6
3.2 Datasidonta....7
4 JAVASCRIPT....8
4.1 JavaScriptin ominaisuuksia ja rajoitteita....8
4.2 JQuery....9
5 KNOCKOUT-KIRJASTO....10
5.1 Knockout hyödyt ja haitat....12
5.2 Knockout asennus....12
6 TOTEUTUKSEN KUVAUS....13
6.1 Observable-muuttujat ja näkymämallit....13
6.1.1 Observable-muuttujat....14
6.1.2 Johdettu Observable-muuttuja....15
6.1.3 ViewModel-olion luominen....16
6.1.4 Foreach-sidonta....17
6.1.5 Click-sidonta....18
6.2 Sapluunat....20
6.2.1 Sapluunoiden käyttäminen....21
6.2.2 Yhteystietojen näyttäminen....21
6.2.3 Tapahtumasidonnat....23
6.3 Elementtikohtainen näkymämalli....25
6.4 Alasvetovalikko....29
6.5 Jatkokehitys....30
6.6 Yhteenveto....30

1 JOHDANTO

Käyttöliittymän ohjelmointi on monimutkainen prosessi ohjelmiston kehityksessä. Käyttöliittymä on kokonaisuus, jossa ulkoasu, käytettävyys, kielet, suorituskyky ja käyttäjän vaatimukset nivoutuvat yhteen. Näiden tarpeiden tyydyttäminen saattaa olla monimutkaista käyttöliittymää rakentaessa hankalaa.

Modernin responsiivisen käyttöliittymän rakentaminen puhtaalla JavaScriptillä on työlästä ja usein myös hidasta. Tämän vuoksi JavaScriptin päälle on rakennettu useita kirjastoja HTML-elementtien ja toiminnallisuuden hallintaan, kuten esimerkiksi jQuery ja jQueryUI. Näkymien responsiivisuus ja monimutkaisen toiminnallisuuden rakentaminen ei koske vain web-ohjelmointia, vaan ongelma esiintyy myös perinteisessä näyttöohjelmoinnissa (esimerkiksi WinForms ja WPF).

MVVM on sovellusarkkitehtuurimalli, jolla käyttöliittymän ohjelmointia pyritään helpottamaan ja yksinkertaistamaan. Tässä opinnäytetyössä kuvataan web-sovelluksen toteuttamista Knockout-kirjastolla, joka tuo MVVM-mallin JavaScriptiin. Opinnäytetyön tarkoituksena on esitellä Knockout-kirjaston oleellisimmat osat sovelluksen avulla.

2 YRITYS JA TOIMEKSIANNON KUVAUS

T-Base Oy on Kotkassa toimiva ohjelmisto- ja konsultointiyritys. Yrityksellä on sivutoimipiste Harjavallassa, jossa tällä hetkellä työskentelen. Yritys käyttää pääosin Microsoft-lähtöisiä tekniikoita mukaanlukien Sharepoint, ASP.NET MVC ja WPF.

T-Base Oy:n toimeksianto on osa laajempaa web-pohjaista tietojenhallintajärjestelmää, joka toteutetaan Sharepoint- ja MVVM-tekniikoilla. Sovelluksen asiakaspuoli toteutetaan Knockout-kirjastoa käyttämällä palvelinkuorman minimoimiseksi. Tässä opinnäytetyössä käsiteltävä osa-alue on yhteystietojen hallinnan totetutus.

3 MVVM-MALLI

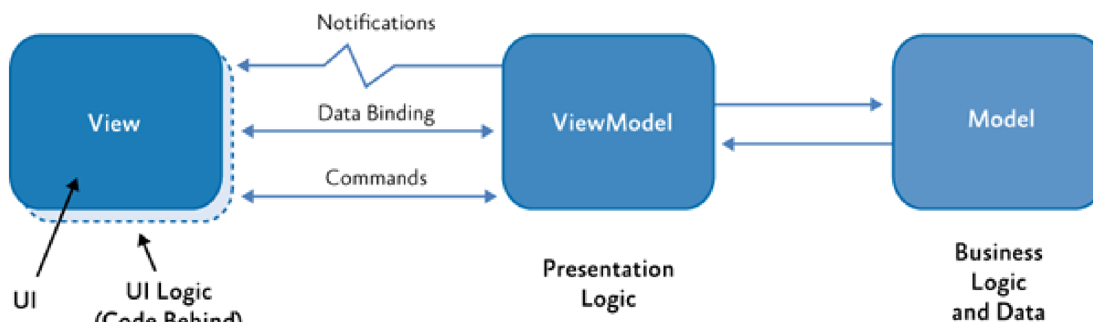
MVVM on Microsoftin kehittämä arkkitehtuurimalli käyttöliittymän toteutukselle. MVVM-mallin perusajatuksena on erottaa graafisen käyttöliittymän ohjelmointi ohjelman business-logiikasta jaottelemalla ohjelmointiprosessi kolmeen osa-alueeseen: Model (malli), View (näkymä) sekä ViewModel (näkymämalli). Näkymämalli toimii kääntäjänä mallin ja näkymän välillä, purkaen mallista saatavan datan esitettävään muotoon näkymälle ja välittämään näkymästä tulevat komennot mallille.

3.1 MVVM-mallin historia

Käyttöliittymäohjelmoinnin helpottamiseksi on luotu useita arkkitehtuuri- ja suunnittelumalleja. MVP (model, view, presenter) on variaatio MVC-arkkitehtuurimallista, joka on ollut laajasti käytössä jo vuosikymmeniä. MVP-mallissa näkymä on mallin visuaalinen representaatio, jonka Presenter rakentaa. Presenter vastaa myös näkymästä tulevista komennoista. (Smith 2009)

Vuonna 2004 Martin Fowler julkaisi artikkelin uudenlaisesta arkkitehtuurimallista nimeltä Presentation Model (PM) (Fowler 2004). PM muistuttaa läheisesti MVC-mallin ajattelua, jossa näkymä ja sitä ohjaava osapuoli erotetaan toisistaan. Uutena ajatuksena mallissa oli, että PM, näkymää ohjaava osapuoli, on itse näkymän abstrakti kuvaus. Näin ollen näkymä on PM:n visualisoitu versio, ei datan, kuten MVC- ja MVP-malleissa. PM pitää näkymän jatkuvasti ajan tasalla, jotta näkymä vastaa presentation modelia. (Smith 2009)

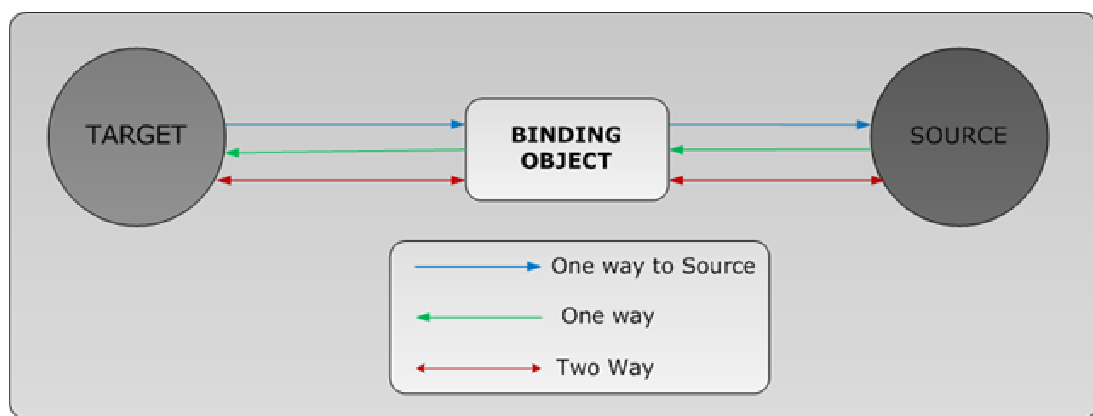
Vuonna 2005 John Gossman paljastaa Model-View-ViewModel (MVVM) -mallin blogikirjoituksessaan (Gossman 2005) (Kuva 1). MVVM muistuttaa läheisesti Fowlerin esittämää Presentation Modelia. Erona on, että Presentation Modelissa näkymän päivittämisen logiikka on rakennettu PM-luokkaan, mutta MVVM-mallissa päivittämisen hoitaa käytössä oleva tekniikka, kyseisessä artikkelissa Microsoftin WPF tai Silverlight. (Smith 2009)



Kuva 1. Kuvaus MVVM-mallista.
(http://msdn.microsoft.com/en-us/library/gg405484%28v=pandp.40%29.aspx)

3.2 Datasidonta

MVVM-mallin näkymä koostuu useista komponenteista, yksinkertaisimmillaan painikkeista, tekstistä, tekstikentistä ja grafiikasta. Toiminnallisuuden monimutkaistuessa ja kenttien riippuvuuksia luodessa näiden kenttien pitäminen ajan tasalla hankaloituu ja tarvittavan ohjelmakoodin määrä moninkertaistuu yhä monimutkaisempien yhteyksien ylläpitämiseksi. Käsite datasidonta (eng. data binding) tarkoittaa kenttien sitomista suoraan näkymämallin ominaisuuksiin. Yksinkertaisuudessaan tämä tarkoittaa sitä, että kun näkymämallin ominaisuutta muutetaan, vastaava kenttä muuttuu näkymässä automaattisesti. Datasidonta toimii myös toiseen suuntaan tai molempiin yhtäaikaisesti (Kuva 2). Kun käyttöliittymän kenttä päivitetään, näkymämallin ominaisuus muuttuu automaattisesti. Näin näkymä ja näkymämalli vastaavat jatkuvasti toisiaan.
(Gossman 2005)



Kuva 2. Kuvaus datasidonta-prosessista.
(http://www.codeproject.com/Articles/22798/Introduction-to-WPF-Data-Binding)

4 JAVASCRIPT

JavaScript on Netscapen kehittämä kevyt komentosarjakieli. JavaScript on laajasti käytössä erityisesti web-ohjelmoinnissa. JavaScript on dynaamisesti tyypitetty kieli eli sen muuttujia ei tyypitetä, vaan sen tyyppi perustuu sille annettuun arvoon. JavaScriptillä ohjelmoidaan web-sivujen responsiivisuus manipuloiden selaimen DOM-objekteja (Document Object Model). DOM-objektien manipuloinnissa toiminnallisuus sidotaan usein HTML-elementtien tapahtumiin (eng. event) (Kuva 3). (Mozilla 2013)

```
var painike = document.getElementById("Painike");
    painike.onclick = function() {
        alert("Terve maailma!");
    }
```

Kuva 3. Tapahtumapohjainen toiminnallisuus JavaScriptissä. DOM-elementtiä id-attribuutilla "Painike" napsauttamalla tervehditään käyttäjää alert-ikkunalla.

4.1 JavaScriptin ominaisuuksia ja rajoitteita

JavaScriptin prototyyppipohjaisessa oliomallissa ei ole olemassa luokkia toisin kuin useissa ohjelmointikielissä. JavaScriptin perusoliot ovat object ja function. Nämä prototyypit ovat erittäin dynaamisia, eikä niiden rakennetta rajoiteta tai määritetä erikseen. Myös muuttujat ovat dynaamisia eikä niitä tyypitetä. Tämä tekee MVVM-mallin mukaisesta ohjelmoinnista hankalaa, sillä mallissa ulkoasu ja toiminnallisuus perustuvat näkymämalli-olion ominaisuuksiin.

Koska web-sivun elementtejä manipuloidaan natiivisti JavaScriptissä tapahtumien avulla, datasidonta-toiminnallisuutta ei ole olemassa. Esimerkiksi WPF- ja WinForms-ohjelmoinnissa kenttien sitominen luokkien ominaisuuksiin on mahdollista (Smith 2009).

Callback-funktiot ovat olennainen osa JavaScript-ohjelmointia. JavaScriptin funktiot ovat itse asiassa jo itsenään funktio-tyyppisiä objekteja. Callback-funktio on muuttuja (yleensä funktiolle annettava parametri), johon on sijoitettu funktio ja se suoritetaan usein tapahtumankäsittelijän sisällä (Kuva 4). (Vollmer 2011)

```
var matti = function() {
    return "Matti Meikäläinen";
}

Terve(matti);

function Terve(henkilo){
    alert("Terve " + henkilo());
}
```

Kuva 4. Callback-funktion toiminta.

JavaScript suoritetaan selaimen ajoympäristön (eng. runtime) toimesta. Tämä ajoympäristö on erilainen eri selaimissa: esimerkiksi Chromessa se on nimeltään V8 ja Internet Explorerissa Chakra. Tästä erilaisuudesta johtuen selaimissa saattaa esiintyä eroavaisuuksia toiminnallisuuden suhteen ja web-sivu voi toimia eri tavalla, näyttää erilaiselta tai pahimmassa tapauksessa ei toimia ollenkaan riippuen siitä, mitä selainta käyttäjä käyttää.

4.2 JQuery

JQuery on runsaasti käytetty JavaScript-kirjasto. Sen tarkoituksena on helpottaa ja nopeuttaa DOM-elementtien käsittelyä (Kuva 5). Vaikka jQuery korjaa osiltaan JavaScriptin puutteita, se pohjautuu silti edelleen samanlaiseen ajattelutapaan kuin JavaScriptikin.

```
$("#Painike").click(function() {
    alert("Terve maailma!");
});
```

Kuva 5. Tapahtumapohjainen toiminnallisuus jQueryllä.

Myöskään jQueryssä ei ole datasidonta-toiminnallisuutta, mutta HTML:n käsittely on huomattavasti helpompaa, joten tällaisen toiminnallisuuden rakentaminen on nopeampaa kuin natiivilla JavaScriptillä. Lisäksi jQueryä vaivaa samat puutteet kuin JavaScriptiäkin, sillä sen ydin kuitenkin on kirjoitettu samalla komentorivikielellä.

Tekstin määrittäminen HTML-elementteihin tapahtuu jQueryssä seuraavasti. Oletetaan, että sovelluksessa on HTML-sivu, jossa näytetään henkilön tiedot tekstikentteinä, joihin asetetaan arvot jQueryllä (Kuva 6).

HTML:

```
<h2>Henkilön tiedot:</h2>
<p id="etunimi"/>
<p id="sukunimi"/>
<p id="postiosoite"/>
<p id="postinumero"/>
<p id="paikkakunta"/>
```

jQuery:

```
$(document).ready(function() {
        var henkilo = {
                etunimi: "Teppo",
                sukunimi: "Testimies",
                postiosoite: "Testikatu 1",
                postinumero: "123456",
                postitoimipaikka: "Testikaupunki"
        };

  $("#etunimi").text(henkilo.etunimi);
  $("#sukunimi").text(henkilo.sukunimi);
  $("#postiosoite").text(henkilo.postiosoite);
  $("#postinumero").text(henkilo.postinumero);
  $("#postitoimipaikka").text(henkilo.postitoimipaikka );
});
```

Kuva 6. HTML-elementteihin kirjoittaminen jQueryllä.

JQueryllä tehdyssä toteutuksessa täytetään henkilo-olion tiedot yksi kerrallaan HTML-elementteihin niiden id-attribuutin perusteella. Jokaiselle ominaisuudelle kirjoitetaan oma rivinsä. Jos henkilo-objektin jokin ominaisuus muuttuu, sama rivi joudutaan ajamaan uudelleen. Taulukoita käytettäessä toiminnallisuus muuttuu yhä monimutkaisemmaksi ja käyttöliittymän ohjelmakoodin kirjoittaminen käy nopeasti erittäin työlääksi.

5 KNOCKOUT-KIRJASTO

Knockout on JavaScriptillä rakennettu kirjasto MVVM-ohjelmointiin. Kirjaston on kehittänyt Microsoftilla työskentelevä Steve Sanderson. Knockout on Sandersonin henkilökohtainen, avoimen lähdekoodin koodikirjasto, jonka tarkoitus on tuoda

MVVM-malli web-ohjelmointiin. Kirjasto erittelee mallin mukaisesti näkymän, näkymään liittyvän toiminnallisuuden ja datamallin toisistaan. Kirjasto perustuu vahvasti JavaScriptin prototyyppeihin sekä JSON-objekteihin (Papa 2012). Knockout tukee myös datasidontaa, mikä tekee JavaScriptin ohjelmoinnista huomattavasti nopeampaa (Kuva 8).

HTML:

```
<h2>Henkilön tiedot:</h2>
<p data-bind="text: etunimi"/>
<p data-bind="text: sukunimi"/>
<p data-bind="text: postiosoite"/>
<p data-bind="text: postinumero"/>
<p data-bind="text: postitoimipaikka"/>
```

JavaScript:

```
$(document).ready(function() {

        var henkilo = {
                etunimi: "Teppo",
                sukunimi: "Testimies",
                postiosoite: "Testikatu 1",
                postinumero: "123456",
                postitoimipaikka: "Testikaupunki"
        };

        ko.applyBindings(henkilo);
    });
```

Kuva 8. HTML ja JavaScript henkilötietojen näyttämisestä Knockoutilla.

5.1 Knockout hyödyt ja haitat

Vertaillaan edellä esitettyä koodia (Kuva 7 ja Kuva 8). JQueryä käyttämällä jokaiselle HTML-elementille annettiin id-arvo, jonka avulla sille määritetään teksti henkilo-objektin ominaisuuksista yksitellen. Knockoutilla kentän arvo määritetään suoraan HTML-koodissa data-bind-attribuutilla. Lopuksi kutsutaan metodia applyBindings nimiavaruudesta ko (Knockout), joka kertoo mikä tämänhetkinen datakonteksti on, tässä tapauksessa henkilo-muuttuja. Tarvittavan koodin määrä on huomattavasti pienempi.

MVVM-mallia käyttämällä JavaScriptissä HTML-sivujen ja elementtien koodit pysyvät täysin irrallaan JavaScriptistä. Lisäksi itse JavaScript on vielä eroteltuna toiminnalliseen logiikkaan, näkymämalliin eli näkymän abstraktiin malliin. Näin JavaScript on huomattavasti ylläpidettävämpää ja kaikki ohjelmakoodit noudattavat samaa rakennetta. (Gossman 2006)

Yksinkertaisissa web-applikaatioissa Knockout saattaa kuitenkin olla liian monimutkainen yksinkertaisen toiminnallisuuden saavuttamiseksi. Lisäksi uuden kirjaston lisääminen monimutkaistaa debuggausta. Varsinkin datasidontaa on vaikeaa debugata mikäli JavaScript ei toimikaan. Usein datasidontavirheissä selain ei edes ilmoita ongelmasta, joten virheen etsiminen saattaa olla työlästä. (Gossman 2006)

5.2 Knockout asennus

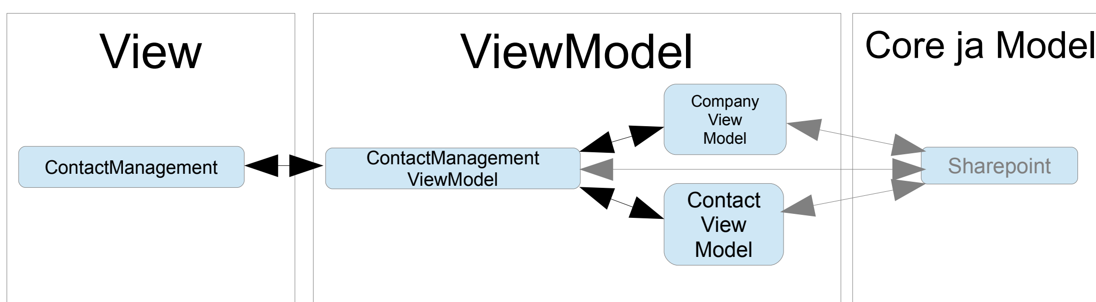
Knockout-kirjaston voi ladata osoitteesta http://knockoutjs.com/downloads/index.html. Kirjasto koostuu yhdestä js-tiedostosta, joka sijoitetaan yleensä projektikansion määrättyyn JavaScript-alikansioon.

Kirjasto otetaan käyttöön HTML-sivulla normaalina script-referenssinä, jossa src viittaa kirjaston sijaintiin:

```
<script type='text/javascript' src='knockout-3.0.0.js'></script>
```

6 TOTEUTUKSEN KUVAUS

Alla on esitetty kuvaus yhteystietojenhallinnan MVVM-mallin toteutuksesta (Kuva 9). Asiakaspuolen toteututus on jaettu kahteen osaan: näkymään ja näkymämalleihin. Näkymä on yhteystietojen selausnäkymä, jossa käyttäjä voi hakea yrityksiä ja yhteystietoja Sharepoint-tietokannasta. Näkymämalli on näkymän abstrakti kuvaus, joka toimii rajapintana Sharepointin objektien (modelien) ja näkymän välillä. Core koostuu Sharepointin Client Object Modelista ja sille rakennetuista JavaScript helper-funktioista, vakioista, sekä palvelinpuolen C#-koodista. Mallit ovat Sharepointin tietokannassa sijaitsevia objekteja, jotka parsitaan Knockoutin näkymämalleiksi. Core on kuvassa piirretty harmaalla, koska se ei ole osa tätä toimeksiantoa.



Kuva 9. Yhteystietojenhallinta, MVVM-mallin kuvaus.

6.1 Observable-muuttujat ja näkymämallit

Toteutuksen näkymämallit ovat CompanyViewModel eli asiakasyritys sekä ContactViewModel eli yhteystieto. Lisäksi kokonaisuuden käsittelylle luodaan oma näkymämalli nimeltä ContactManagementViewModel. Yrityksillä on omia yhteystietojaan sekä yhteyshenkilöitä. Koska JavaScript on prototyyppipohjainen ja luokaton, näkymämallit toteutetaan JavaScript funktioina. Näkymämallifunktio sisältää useita muuttujia, jotka nimitetään observable-muuttujiksi. Näistä muuttujista tulee olion ominaisuuksia.

6.1.1 Observable-muuttujat

Knockoutin yksi pääajatuksista on käyttöliittymän automaattinen päivittäminen näkymämallin mukaan. Tämä saavutetaan nimittämällä JavaScript-muuttujat Knockoutin observable-muuttujiksi (Kuva 10).

```
/*
 * Parametric Constructor of Contact ViewModel
 * @param {JSON} data
 * @returns Contact ViewModel
 */
function ContactViewModel(data) {
    /*
     * Meaning of "this" may change in Javascript
     * so we store it into self-variable
     */
    var self = this;

    self.FirstName = ko.observable(data.FirstName);
    self.LastName = ko.observable(data.LastName);
    self.Address = ko.observable(data.Address);
    self.PostalCode = ko.observable(data.PostalCode);
    self.City = ko.observable(data.City);
    self.Email = ko.observable(data.Email);
    self.Phone = ko.observable(data.Phone);
}
```

Kuva 10. ContactViewModel.

JavaScriptin ominaisuuksiin kuuluu, että sanan this merkitys muuttuu sen sijainnista riippuen. Näkymämallissa muuttujat tallennetaan aina itse näkymämallifunktioon. Tämän vuoksi on määriteltävä muuttuja self, johon muuttujat asetetaan. Konstruktorin data-parametri on JSON-objekti, johon sijoitetaan näkymämallin asetusarvot. Observable-funktio ko nimiavaruudessa luo observable-muuttujan, jolloin Knockout pystyy automaattisesti päivittämään näkymää kun näkymämalli muuttuu. Observable-funktion parametri on muuttujan asetusarvo.

Knockoutin sisäistä toimintaa tarkastellessa huomaa, että kirjasto käyttää paljon hyväkseen JavaScriptin callback-funktion periaatteita. Jokainen näkymämallille

annettava arvo on itseasiassa funktio. Tämä on tärkeää näkymämallin arvoja tulostettaessa ja testattaessa.

Observable-tyyppiset taulukot määritetään eri tavalla. Niille on olemassa oma funktio: observableArray, jota tarvitaan yrityksen näkymämallissa (Kuva 11).

```
/*
 * Parametric Constructor of Company ViewModel
 * @param {JSON} data
 * @returns Company ViewModel
 */
function CompanyViewModel(data) {
    var self = this;

    self.Contacts = ko.observableArray(data.Contacts);
    self.Name = ko.observable(data.Name);
    self.Address = ko.observable(data.Address);
    self.PostalCode = ko.observable(data.PostalCode);
    self.City = ko.observable(data.City);
}
```

Kuva 11. CompanyViewModel ja observable-taulukko (eng. observable array).

Observable-taulukko toimii samalla periaatteella kuin observablekin. Funktion parametri on observable-taulukon asetusarvo eli JavaScriptin Array-tyyppinen muuttuja.

6.1.2 Johdettu Observable-muuttuja

Observablen ja observableArrayn lisäksi Knockoutissa on olemassa vielä yksi usein käytetty observable-muuttuja: computed observable eli johdettu observable-muuttuja (Kuva 12). Johdettu observable-muuttuja on funktio, jonka avulla luodaan observable-muuttuja, joka riippuu useammasta muusta muuttujasta. ContactViewModel-näkymämalliin tarvitaan tällainen muuttuja kuvaamaan yhteyshenkilön koko nimeä.

```
// Create computed observable for FullName
self.FullName = ko.computed(function() {
    return self.LastName() + " " + self.FirstName();
});
```

Kuva 12. Johdettu observable-muuttuja.

Huomion arvoinen asia on, että suku- ja etunimen arvoja pyydetään kutsumalla funktiota. Tämä johtuu siitä, että ko.observable on funktio ja se sijoitetaan LastName-nimiseen ominaisuuteen, jolloin LastName-funktio palauttaa sille asetetun arvon.

6.1.3 ViewModel-olion luominen

Sovelluksessa näkymämallin data-argumentti on Sharepointista parsittu JSON-objekti. Demonstrointitarkoituksessa uusi yhteystieto voidaan myös luoda vain asiakaspuolelle (Kuva 13).

```
/*
 * Creates a test contact for debugging purposes
 * @returns {ContactViewModel}
 */
function CreateTestContact() {
    var testContact = {
        FirstName: "Teppo",
        LastName: "Testimies",
        Address: "Testikatu 1",
        PostalCode: "123456",
        City: "Testikaupunki"
    };


    var contact = new ContactViewModel(testContact);

    return contact;
}
```

Kuva 13. Testiyhteystiedon luominen.

Näkymää varten tarvitaan vielä yksi näkymämalli: ContactManagementViewModel (Kuva 14). ContactManagementViewModel toimii rajapintana näkymän ja

Sharepointin välillä sekä vastaanottaa käyttäjän toimintoja. Aluksi näkymään listataan sovelluksessa olevat yritykset.

```
/*
 * Constructor of ContactManagementViewModel
 * Handles all Contact and Company related actions in UI
 */
function ContactManagementViewModel() {
    var self = this;
    self.Companies = ko.observableArray(GetCompanies());

}
```

Kuva 14. ContactManagementViewModel

GetCompanies-funktio pyytää corelta kaikki sovelluksessa olevat yritykset Sharepointin listasta ja parsii ne CompanyViewModel-objekteiksi. Lopulta funktio palauttaa JavaScript-taulukon, joka koostuu CompanyViewModel-objekteista.

6.1.4 Foreach-sidonta

Foreach-sidonta kopioi HTML-alueen jokaiselle näkymämallin taulukon elementille. Tässä tapauksessa sovellus renderöi jokaisen yrityksen nimen Companies observable-taulukosta (Kuva 15).

```
<div data-bind='foreach: Companies'>
    <div class='item company action'>
        <div class="companyHeader" >
            <div class='mangoBlock' ></div>
            <a class='itemContent' data-bind='text: Name'></a>
        </div>
    </div>
</div>
```

Kuva 15. Foreach-sidonta

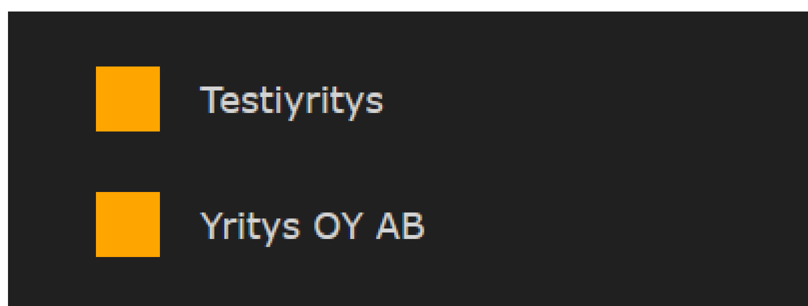
Datasidokset eivät toimi itsenäisesti. Knockoutille on vielä määriteltävä mikä näkymämalli näkymälle ladataan. Tässä tapauksessa ContactManagementViewModel. Tätä lataamista varten luodaan uusi funktio, jossa

ladataan näkymämalli ja tehdään muut näkymään liittyvät asetustoimenpiteet (Kuva 16).

```
/*
 * Load ContactManagementViewModel to Knockout and initialize view
 *
 */
var _viewModel;
function LoadContactManagementViewModel() {
    _viewModel = new ContactManagementViewModel();
    ko.applyBindings(_viewModel);
}
```

Kuva 16. Näkymämallin lataaminen Knockouttiin

Nykyinen näkymämalli tallennetaan _viewModel-muuttujaan, jolloin sitä voidaan käyttää yksinkertaisesti kaikkialla JavaScriptissä. Tämän jälkeen kutsutaan funktiota ko.applyBindings, joka lataa näkymämallin Knockoutin datakontekstiksi. Käyttöliittymässä sovelluksessa olevat yritykset listautuvat automaattisesti foreach-sidonnan avulla (Kuva 17).



Kuva 17. Yritysten listaaminen näkymämallista

6.1.5 Click-sidonta

Käyttäjän napsauttaessa yrityksen nimeä käyttöliittymässä sen alle listataan yrityksessä olevat yhteyshenkilöt. Tätä varten on olemassa click-sidonta, joka kutsuu määriteltyä näkymämallin funktiota DOM-elementtiä napsauttaessa (Kuva 18).

HTML-koodissa yrityksen yhteyshenkilöt piilotetaan oman elementtinsä sisälle, jotta niiden näkyvyyttä voidaan käsitellä click-tapahtumankäsittelijässä.

```
<div data-bind='foreach: Companies'>
    <div class='item company action'>
        <div class="companyHeader" data-bind="click: $root.SelectCompany">
            <div class='mangoBlock' ></div>
            <a class='itemContent' data-bind='text: Name'></a>
        </div>
        <div class='contactList' style='display:none;'>
            <div data-bind='foreach: Contacts'>
                <div class="subItem contact action">
                    <div class='contactHeader'>
                        <img src='images/contact.png'/>
                        <a class='itemContent' data-bind='text: FullName'></a>
                    </div>
                </div>
            </div>
        </div>
    </div>
</div>
```

Kuva 18. Päivitetty HTML yhteyshenkilöitä varten.

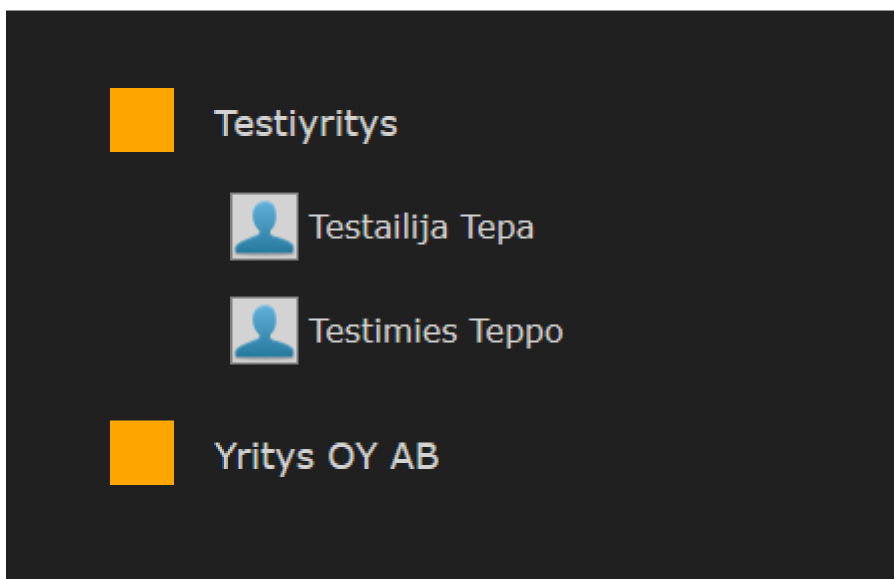
Div-elementti luokalla companyHeader toimii napsautettavana elementtinä. Click-sidonnassa näkymämalliin viitataan sanalla $root, joka kertoo, että funktio SelectCompany sijaitsee juurinäkymämallissa. Tässä tapauksessa se viittaa ContactManagementViewModel-näkymämalliin. Ilman $root-avainsanaa SelectCompany viittaisi foreach-silmukan sisällä olevaan CompanyViewModel-objektiin. Määrittelyn mukaisesti käyttäjän ja käyttöliittymän väliset toiminnallisuudet sijoitetaan ContactManagementViewModel-näkymämalliin (Kuva 19).

```
/*
 * Constructor of ContactManagementViewModel
 * Handles all Contact and Company related actions in UI
 */
function ContactManagementViewModel() {
    var self = this;
    self.Companies = ko.observableArray(GetCompanies());

    self.SelectCompany = function(item, event) {
        // Crawl back to parent company item and toggle the list visibility
        $(event.target).parents(".company").find(".contactList").slideToggle(200);
    };
}
```

Kuva 19. Yrityksen valinta.

Click-tapahtumankäsittelijässä on kaksi parametria; item ja event. Item on käyttäjän napsauttama näkymämalli ja event on JavaScriptin natiivi tapahtumaobjekti.

Napsautettu elementti saadaan event-objektin target-nimisestä ominaisuudesta. JQueryn avulla DOM-puusta etsitään lähin company-luokan omaava elementti, jonka allaoleva lista näytetään tai piilotetaan (Kuva 20).



Kuva 20. Yhteyshenkilöiden listaus ja piilotus.

6.2 Sapluunat

Monimutkaisempien toiminnallisuuksien ja rakenteiden ohjelmointi tekee HTML-koodista yhä vaikeammin luettavaa. Knockout-kirjasto sisältää ratkaisun tähän ongelmaan template-sidonten avulla. Tarkoituksena on, että jokaiselle kokonaisuudelle määritellään sapluuna (eng. template), jota käytetään näkymämallia esitettäessä (Kuva 21). Sapluunoita käytetään usein foreach-silmukoiden kanssa.

```
<script type='text/html' id='company-template'>
    <div class='item company action'>
        <div class="companyHeader" data-bind="click: $root.SelectCompany">
            <div class='mangoBlock'></div>
            <a class='itemContent' data-bind='text: Name'></a>
        </div>
        <div class='contactList' style='display:none;'>
            <div data-bind='template:{name:"contact-template", foreach: Contacts}'>
            </div>
        </div>
    </div>
</script>

<script type='text/html' id='contact-template'>
    <div class='subItem contact action'>
        <div class='contactHeader'>
            <img src='images/contact.png'/>
            <a class='itemContent'  data-bind='text: FullName'></a>
        </div>
    </div>
</script>
```

Kuva 21. Sapluunat yrityksille ja yhteystiedoille.

6.2.1 Sapluunoiden käyttäminen

Sapluunat voidaan sijoittaa omiin tiedostoihinsa tai niille määrätyille paikoille HTML-koodissa. Sapluunat kirjoitetaan script-tagin sisälle ja niille annetaan id-attribuutti. Edelläesitetyssä yrityssapluunassa yhteystietosapluunaa käytetään yrityssapluunan sisällä. Näin HTML-koodin määrä kutistuu näkymässä huomattavasti. Sapluuna otetaan käyttöön datasidonnan avulla. Elementin data-bind-attribuuttiin kirjoitetaan avainsana template, jolla viitataan käytettävän sapluunan id-attribuuttiin (Kuva 22).

```
<div data-bind='template: {name:"company-template", foreach: Companies}'>
</div>
```

Kuva 22. Sapluunan käyttäminen.

6.2.2 Yhteystietojen näyttäminen

Määrittelyn mukaisesti yhteyshenkilöä napsautettaessa tulee avata kyseisen henkilön yhteystiedot näkyviin käyttöliittymään (Kuva 23). Toimintaperiaate on sama kuin yrityksissäkin. Lisäksi SelectContact-tapahtumankäsitteljiä lisätään ContactManagementViewModel-näkymämalliin (Kuva 24).

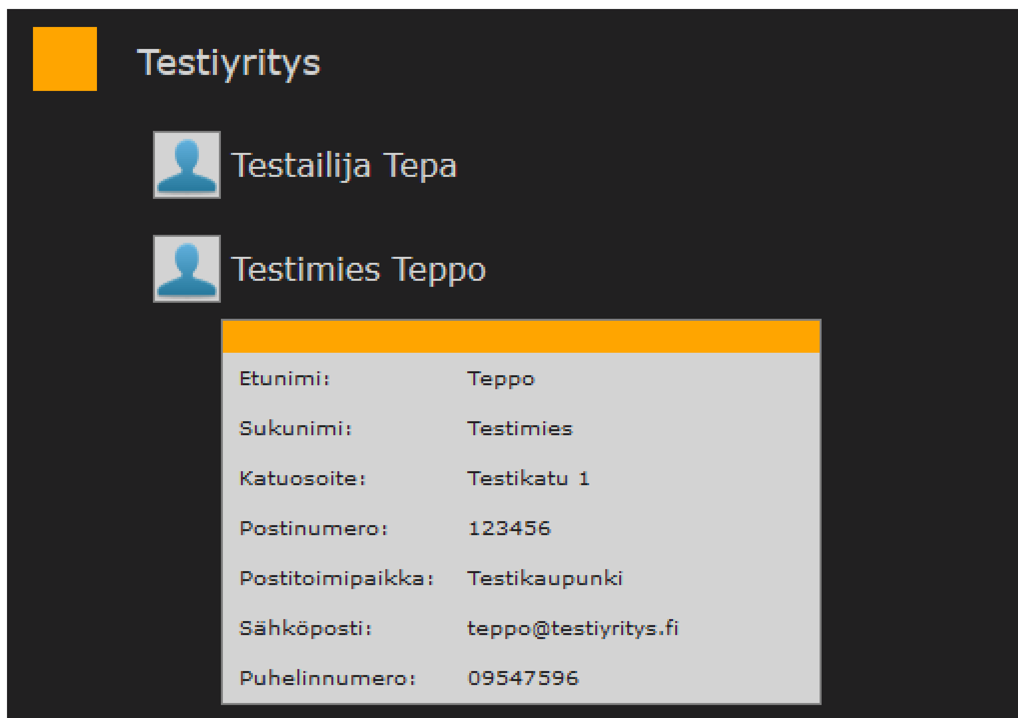
```
<script type='text/html' id='contact-template'>
    <div class='subItem contact action'>
        <div class='contactHeader' data-bind="click: $root.SelectContact">
            <img src='images/contact.png'/>
            <a class='itemContent'  data-bind='text: FullName'></a>
        </div>
        <div class='card'style='display:none;'>
            <table>
                <thead>
                    <tr><td colspan="2"></td></tr>
                </thead>
                <tbody>
                    <tr><td class='label'>Etunimi:</td><td data-bind="text: FirstName" /></tr>
                    <tr><td class='label'>Sukunimi:</td><td data-bind="text: LastName" /></tr>
                    <tr><td class='label'>Katuosoite:</td><td data-bind="text: Address" /></tr>
                    <tr><td class='label'>Postinumero:</td><td data-bind="text: PostalCode" /></tr>
                    <tr><td class='label'>Postitoimipaikka:</td><td data-bind="text: City" /></tr>
                    <tr><td class='label'>Sähköposti:</td><td data-bind="text: Email" /></tr>
                    <tr><td class='label'>Puhelinnumero:</td><td data-bind="text: Phone" /></tr>
                </tbody>
            </table>
        </div>
    </div>
</script>
```

Kuva 23. Päivitetty yhteystietosapluuna.

```
/*
 * Constructor of ContactManagementViewModel
 * Handles all Contact and Company related actions in UI
 */
function ContactManagementViewModel() {
    var self = this;
    self.Companies = ko.observableArray(GetCompanies());

    self.SelectCompany = function(item, event) {
        // Crawl back to parent company item and toggle the list visibility
        $(event.target).parents(".company").find(".contactList").slideToggle(200);

        // Close all underlying Contact cards
        $(event.target).parents(".company").find(".card").slideUp(200);
    };

    self.SelectContact = function(item, event) {
        // Crawl back to parent contact item and toggle the card visibility
        $(event.target).parents(".contact").find(".card").slideToggle(200);
    };
}
```

Kuva 24. SelectContact-tapahtumankäsittelijä.

SelectContact-tapahtumankäsittelijässä DOM-puusta etsitään card-luokan elementti, joka näytetään käyttäjälle (Kuva 25).



Kuva 25. Käyttöliittymä.

6.2.3 Tapahtumasidonnat

Tapahtumasidonta mahdollistaa minkä tahansa tapahtumakäsittelijän kutsumisen. Edellä esitetyssä sovelluksessa on tarvittu vain Click-tapahtumasidontaa, mutta sidonnan voi tehdä mihin tahansa JavaScriptin tapahtumaan. Yhteystietojen haku toteutetaan jQueryUI-kirjaston autocomplete-elementillä. Tarvitaan tapahtumasidonta, jossa valinta autocomplete-elementistä laukaisee tapahtumankäsittelijän. Valikkoelementin napsautus näyttää elementin alle piilotetun sisällön (Kuva 26).

```
self.ToggleAction = function(item,event){
    $(event.target).parents(".action").find(".actionContents").slideToggle(200);
}
<div class='contentLeft'>
    <div class="action item">
        <div id="searchHeader" data-bind="click: ToggleAction">
            <div class='mangoBlock' ></div>
            <a class="menuItemContent">Hae yhteystietoja</a>
        </div>
        <div class="actionContents" style='display:none;'>
            <div>
                <input type="text" id="searchContact" name="searchContact"
                       data-bind="event: { autocompleteselect: $root.SearchContact}">
            </div>
        </div>
    </div>
</div>
```

Kuva 26. Yhteystietojen haku ja sen näyttäminen.

Autocomplete-elementti luodaan LoadContactManagementViewModel-funktiossa (Kuva 27).

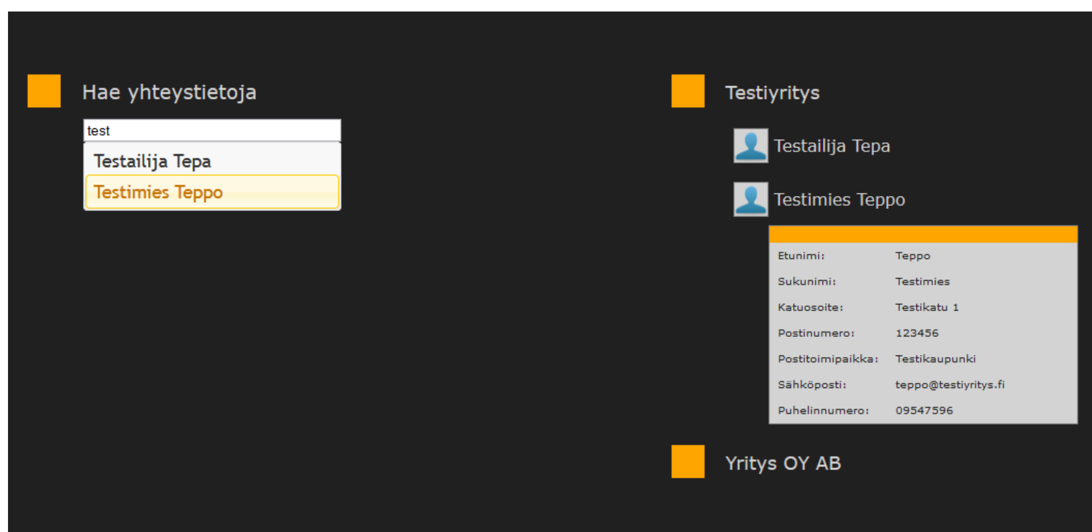
```
/*
 * Load ContactManagementViewModel to Knockout and initialize view
 *
 */
function LoadContactManagementViewModel() {
    _viewModel = new ContactManagementViewModel();
    ko.applyBindings(_viewModel);

    $( "#searchContact" ).autocomplete({
        source: _viewModel.Contacts()
    });
}
```

Kuva 27. Autocomplete-elementin luominen.

Input-elementtiin id:llä "searchContact" sidotaan tapahtumankäsittelijä SearchContact, joka sijaitsee ContactManagementViewModel-näkymämallissa. Käyttäjän valitessa yhteyshenkilö autocomplete-elementistä, tapahtuma autocompleteselect laukaistaan, joka taas on sidottu SearchContact-tapahtumankäsittelijään (Kuva 28 ja Kuva 29). Lisäksi autocomplete-elementtiä varten tarvitaan uusi taulukko kaikkia yhteyshenkilöitä varten. Tämä toteutetaan Knockoutin observable-taulukkona.

```
self.Contacts = ko.observableArray([]);
// initialize all contacts
$.each(self.Companies(), function(i,e) {
     $.each(e.Contacts(), function(i,e) {
            self.Contacts.push(e.FullName());
     });
});

// Search for given contact and expand it's card
self.SearchContact = function(item,event,ui) {
    // Close everything
    $(".contactList").slideUp(200);
    $(".card").slideUp(200);

    // Expand parents and found element
    var search = ui.item.value;
    var $element = $("a:contains("+search+")");

    // Set value to input
    $("#searchContact").val(search);

    $element.parents().slideDown(200);
    $element.parents(".contact").find(".card").slideDown(200);
};
```

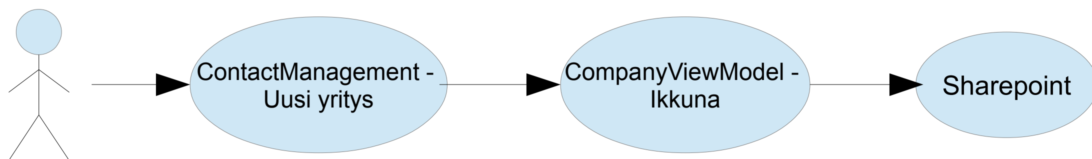
Kuva 28. Yhteyshenkilöhaku.



Kuva 29. Käyttöliittymä.

6.3 Elementtikohtainen näkymämalli

Knockout-kirjasto tukee myös näkymämallien sitomista tiettyyn elementtiin. Sovelluksessa ContactManagementViewModel sidotaan koko näkymään. Kuitenkin joissain tapauksissa halutaan sitoa näkymämalli vain tiettyyn elementtiin näkymän sisällä. Tämä mahdollistaa useamman näkymämallin sitomisen samaan aikaan. Elementtikohtaista näkymämallia tarvitaan sovelluksessa uusien yrityksien lisäämiseen (Kuva 30).



Kuva 30. Uusi yritys -prosessi ja niiden näkymät.

Käyttöliittymään toteutetaan Uusi yritys -linkki, joka avaa jQueryUI-kirjastolla toteutetun ikkunan. Ikkunaan sidotaan uusi tyhjä CompanyViewModel, joka ikkunan tallenna-painiketta painamalla siirretään Sharepointtiin.

JavaScript-koodissa ContactManagementViewModel-näkymälle lisätään uusi tapahtumankäsittelijä, jonka tehtävänä on hakea ikkunan HTML (Kuva 31) Ajaxilla ja avata ikkuna. Ikkunan avaamisen suorittaa OpenDialog-funktio, jonka parametreiksi annetaan ikkunan HTML-koodi, ikkunan otsikko, ikkunaan sidottava näkymämalli, sekä callback-funktio, joka suoritetaan ikkunan tallenna-nappia painettaessa (Kuva 32).

```
<div id="dialog" style="display:none;">
    <table>

        <tr><td>Nimi:</td> <td><input type="text" data-bind="value: Name"></td></tr>
        <tr><td>Osoite:</td><td><input type="text" data-bind="value: Address"></td></tr>
        <tr><td>Postinumero:</td><td><input type="text" data-bind="value: PostalCode"></td></tr>
        <tr><td>Kaupunki:</td><td><input type="text" data-bind="value: City"></td></tr>
    </table>
</div>
```

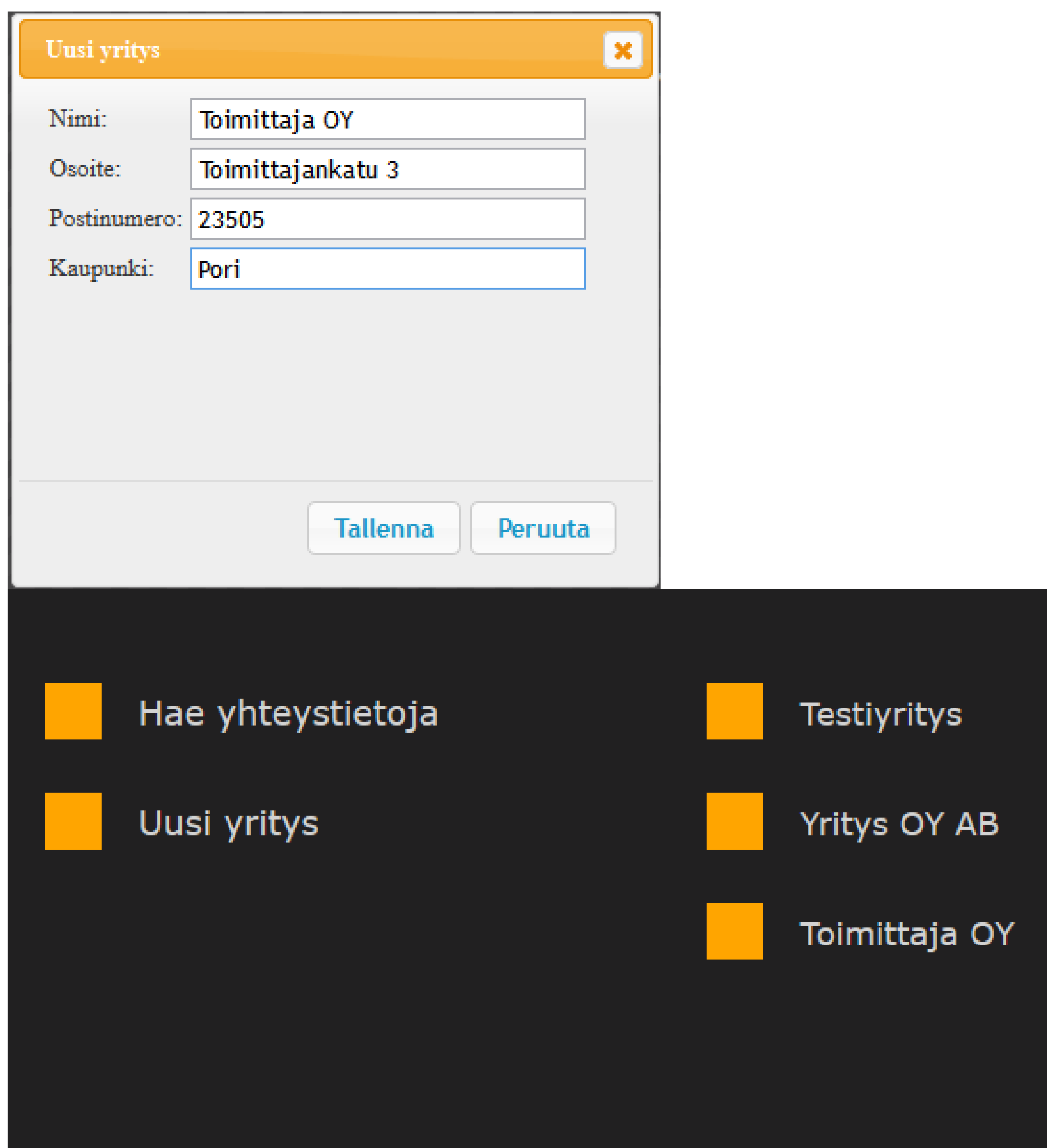
Kuva 31. Yritysikkunan HTML.

```
self.NewCompany = function() {
    // Request the html
     $.ajax({
        url: "views/companydialog.html",
        success: function(html) {
            // Opendialog
            OpenDialog( html,
                    "Uusi yritys",
                    new CompanyViewModel(),
                    function(vm) {
                        // Add on dialogcallback to Sharepoint
                        AddCompany(vm);
                    });
        },
        dataType: "html"
    });
};
 /*
  * Open new dialog
  * @param {type} dialoghtml html of dialog to display
  * @param {type} viewModel viewModel to bind
  * @param {type} callback on dialog save event
  * @returns {undefined}
  */
 function OpenDialog(dialoghtml, title, viewModel, callback){
     $(dialoghtml).dialog({
         height:400,
         width:330,
         title: title,
         modal: true,
         close: function(){
             // Clear bindings for the dialog
             ko.cleanNode(document.getElementById("dialog"));
         },
         buttons: {
             "Tallenna": function() {
                 // Callback
                 if (typeof(callback) !== "undefined") {
                     callback(viewModel);
                 }
                 // Close dialog
                 $( this ).dialog( "close" );
             },
             "Peruuta": function() {
                  // Close dialog
                 $( this ).dialog( "close" );
             }
         }
     });

     // Apply viewModel to dialog
     ko.applyBindings(viewModel,  document.getElementById("dialog"));
 }
```

Kuva 32. Uuden yrityksen JavaScript.

NewCompany -tapahtumankäsittelijää kutsuttaessa ikkunan näkymämalliksi annetaan uusi tyhjä CompanyViewModel, joka ko.applyBindings-funktiolla sidotaan elementtiin "dialog". Näin CompanyViewModel sidotaan vain ikkuna-elementtiin. Tallenna-nappia painettaessa kutsutaan OpenDialog-funktiolle annettua callback-funktiota, suljetaan ikkuna ja tyhjennetään ikkunan sidonta kutsumalla ko.cleanNode-funktiota.

Koko prosessin tuloksena kutsutaan AddCompany-funktiota, joka parsii JSON-objektin Sharepointin listaobjektiksi sekä lisää sen Sharepoint-listaan. Knockout-kirjastoa käyttämällä ikkunan input-elementtejä ei tarvitse parsia erikseen JSON-objektiksi, vaan Knockoutin datasidonta hoitaa tämän automaattisesti (Kuva 33).



Kuva 33. Käyttöliittymä.

6.4 Alasvetovalikko

Olennaisiin HTML-elementteihin kuuluu Select-elementti eli alasvetovalikko. Uutta yhteystietoa lisättäessä käyttäjän on valittava yritys, johon lisättävä yhteyshenkilö kuuluu. Tähän tarkoitukseen Knockoutissa on options-sidonta, jonka avulla alasvetovalikon vaihtoehdot renderöidään. Ikkunan avaamiseen käytetään samaa OpenDialog-funktiota eri parametreilla. ContactViewModel-funktioon tarvitaan myös uusi observable-taulukko, jossa kaikki alasvetovalikossa olevat yritykset sijaitsevat sekä Company-muuttuja, joka viittaa yhteyshenkilön yritykseen (Kuva 34).

```
self.Company = ko.observable(data.Company);

// Copy available companies from main viewmodel for editing
if (typeof(_viewModel) !== "undefined") {
    self.AvailableCompanies = ko.observableArray(_viewModel.Companies());
}
```

```
<div id="dialog" style="display:none;">
    <table>

        <tr><td>Etunimi:</td> <td><input type="text" data-bind="value: FirstName"></td></tr>
        <tr><td>Sukunimi:</td> <td><input type="text" data-bind="value: LastName"></td></tr>
        <tr><td>Osoite:</td><td><input type="text" data-bind="value: Address"></td></tr>
        <tr><td>Postinumero:</td><td><input type="text" data-bind="value: PostalCode"></td></tr>
        <tr><td>Kaupunki:</td><td><input type="text" data-bind="value: City"></td></tr>
        <tr><td>Sähköposti:</td><td><input type="text" data-bind="value: Email"></td></tr>
        <tr><td>Puhelinnumero:</td><td><input type="text" data-bind="value: Phone"></td></tr>
        <tr><td>Yritys:</td><td><select data-bind="options: AvailableCompanies,
                                    optionsText: 'Name',
                                    value: Company,
                                    optionsCaption: '-Valitse yritys-'"></td></tr>
    </table>
</div>
```

Kuva 34. Options-sidonta select-elementtin.

Select-elementin data-bind-attribuuttiin annetaan seuraavat parametrit:

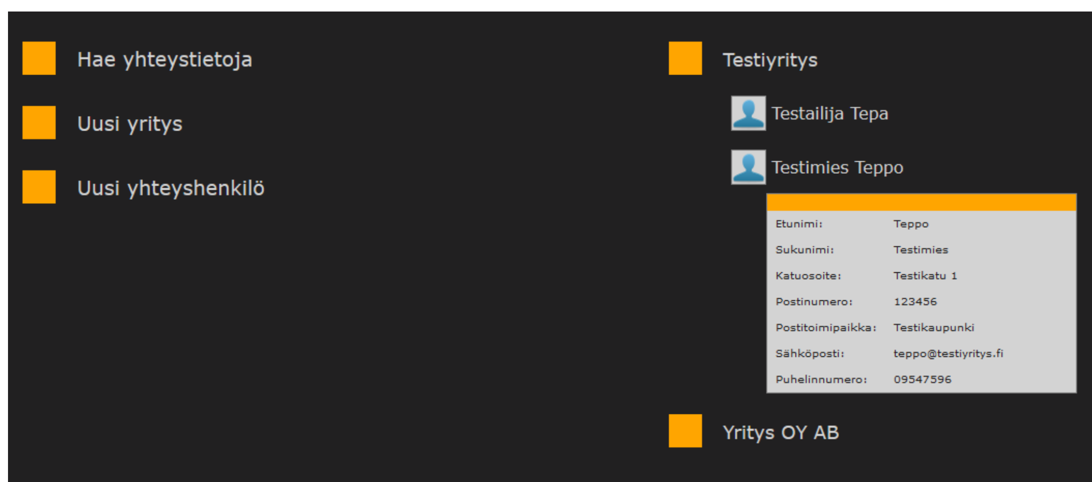
- Options: Näkymämallissa sijaitseva observable-taulukko.
- OptionsText: Observable-taulukon elementin ominaisuuden nimi, joka alasvetovalikossa näytetään. Tässä tapauksessa näytetään yrityksen nimi (Name).
- Value: Mihin näkymämallin ominaisuuteen alasvetovalikon arvo viittaa.
- OptionsCaption: Alasvetovalikossa oletuksena näytettävä teksti.

6.5 Jatkokehitys

Esitelty toiminnallisuus on vain pieni osa suurta yhteystietojen ja dokumentinhallintajärjestelmää (Kuva 35). Kaikki sovelluksen web-käyttöliittymän osiot rakennetaan kuitenkin samaa periaatetta noudattamalla. Yhteystietojen muokkaus, poistaminen ja tarvittaessa monimutkaisempi suodattaminen rakennetaan samaa mallia noudattamalla.

Lisäksi opinnäytetyössä ei ole käsitelty kenttien validointia. Tämä kuitenkin toteutetaan jQuery validation-lisäkirjastolla (http://jqueryvalidation.org/).

Knockoutin näkymämallikohtainen ajattelu helpottaa komentorivikoodin laajentamista ja pitää sen jäsenneltynä, vaikka toiminnallisuus monimutkaistuisikin.



Kuva 35. Valmis käyttöliittymä.

6.6 Yhteenveto

Puhtaan JavaScriptin kankeus responsiivisen käyttöliittymän rakentamisessa on mielestäni suuri ongelma nykypäivän web-ohjelmoinnissa. JavaScript tuntuu vanhentuneelta tekniikalta ja tästä kielii myös sen päälle rakennettujen apukirjastojen määrä. Vaikka käyttöönotto ja tekniikan opiskelu tuntuu aluksi työläältä, sen opetteleminen mielestäni kannattaa, sillä aikaa ja vaivaa säästää sovelluksen laajentuessa huomattavasti.

Knockout-kirjasto vähentää tarvittavan JavaScriptin määrää mielestäni merkittävästi. Lisäksi datasidonta ja näkymämalli-objektit tekevät komentorivikoodista

ylläpidettävämpää, jäsennellympää ja nopeammin käsiteltävää kuin puhdas JavaScript. Knockoutin toimintaperiaate on mielestäni helppo sisäistää, varsinkin jos web-käyttöliittymän toteutuksen puutteisiin ja vaikeuksiin on törmännyt web-ohjelmoinnissa. Mielestäni Knockout varteenotettava työkalu web-käyttöliittymien ohjelmoinnin tehostamiseksi.

LÄHTEET

Fowler, M. 2004 Presentation Model. Viitattu 14.10.2013. Saatavissa:

http://martinfowler.com/eaaDev/PresentationModel.html

Gossman, J. 2005. Introduction to Model/View/ViewModel pattern for building WPF apps. Viitattu: 14.10.2013. Saatavissa:

http://blogs.msdn.com/b/johngossman/archive/2005/10/08/478683.aspx

Gossman, J. 2006. Advantages and disadvantages of M-V-V-M. Viitattu: 14.11.2013. Saatavissa:

hhttp://blogs.msdn.com/b/johngossman/archive/2006/03/04/543695.aspx

Mozilla. 2013. JavaScript Overview. Viitattu: 21.10.2013. Saatavissa:

https://developer.mozilla.org/en/JavaScript/Guide/JavaScript_Overview#JavaScript_and_the_ECMAScript_Specification

Papa, J. 2012. Getting Started with Knockout. Viitattu: 28.10.2013. Saatavissa:

http://msdn.microsoft.com/en-us/magazine/hh781029.aspx

Sanderson, S. 2013. Documentation. Viitattu: 6.11.2013. Saatavissa:

http://knockoutjs.com/documentation/introduction.html

Smith, J. 2009. WPF Apps With The Model-View-ViewModel Design Pattern. Viitattu 14.10.2013. Saatavissa:

http://msdn.microsoft.com/en-us/magazine/dd419663.aspx

Vollmer, M. 2011. Understanding callback functions in JavaScript. Viitattu: 7.1.2014. Saatavissa:

http://recurial.com/programming/understanding-callback-functions-in-javascript/