HAAGA-HELIA
ammattikorkeakoulu

AngularJS sovelluksen kehittäminen – CASE: Hierojan-kortti

Iiro Nurmi

Opinnäytetyö
Tietojenkäsittelyn koulutusohjelma
2015

HAAGA-HELIA
ammattikorkeakoulu

Tiivistelmä

14.05.2015

Tekijä(t)
Iiro Nurmi

Koulutusohjelma
Tietojenkäsittelyn koulutusohjelma

Opinnäytetyön otsikko
AngularJS sovelluksen kehittäminen – CASE: Hierojankortti

Sivu- ja liitesivumäärä
54

Opinnäytetyön otsikko englanniksi
Developing AngularJS application – CASE: Hierojankortti

Opinnäytetyö käsittelee HTML5-sovelluksen toteuttamista AngularJS-sovelluskehyksellä. Opinnäytetyö tehtiin projektityyppisenä ja sen aikana toteutettiin hierojan asiakkuudenhallintaan tarkoitettu sovellus, Hierojankortti. Sovellus oli aloitettu ryhmätyönä osana HAAGA-HELIAn kurssia ja ryhmän muut jäsenet toimivat työn toimeksiantajana.

Opinnäytetyön tavoitteena oli toteuttaa hierojille työpöytä- sekä mobiilikäyttöön tarkoitettu asiakkuudenhallintasovellus. Opinnäytetyö käsittelee sovelluksen asiakaspään AngularJS-ratkaisua sekä HTML5-sovelluksen keskeisimpiä tekniikoita. Sovelluksen taustajärjestelmä rajattiin pois opinnäytetyöstä toimeksiantajan pyynnöstä.

Opinnäytetyössä jatkettiin aiemmin aloitetun sovelluksen kehitystä ja viimeisteltiin lopulliseksi produktiksi. Sovelluksen kehitys oli aloitettu syksyllä 2014 ja opinnäytetyö tehtiin tammikesäkuussa 2015. Sovellus toteutettiin käyttäen AngularJS JavaScript-sovelluskehystä ja toteutettiin samanaikaisesti sekä työpöytä- että mobiilikäyttöön responsiivisen käyttöliittymän avulla.

Opinnäytetyön lopputuloksena syntyi viimeistelty kaupallinen produkti, Hierojankortti. Hierojankortilla hieroja voi hallita asiakkaitaan ja heille tehtyjä hoitotoimenpiteitä. Sovellus oli toimeksiantajan mielestä onnistunut ja valmistui vaatimusmäärittelyiden mukaan.

Asiasanat
AngularJS, HTML5-sovellus, Hierojankortti, asiakkuudenhallinta, responsiivinen suunnittelu, Mobile Angular UI

HAAGA-HELIA
University of Applied Sciences

Abstract

14.05.2015

Author(s)
Iiro Nurmi

Degree programme
Degree programme in Business Information Technology

Report/thesis title	**Number of pages and appendix pages**
Developing AngularJS application – CASE: Hierojankortti	54

This thesis covers principles and technologies of HTML5 application development with AngularJS framework. During the thesis, the author developed an application to ease massage therapists customer management process. The application was initiated as a group project and the group members commissioned the thesis.

The goal of this thesis was to develop a customer relationship management application for massage therapists. The application had to work both on mobile and desktop devices. The theoretical part of this thesis consists of the client-side technologies used in the development process, mostly AngularJS. The software server-side implementation was excluded from the thesis upon the commissioner's request.

The project finalizes the application development, which had started upon the thesis was launched, to a commercially viable product. The development started in the autumn of 2014, and the thesis was done between January and June 2015. The application uses AngularJS client-side framework and is designed to take advantage of its responsive layout and be used across a range of devices.

A commercial application was developed as a result of the thesis. With the application a massage therapist can manage his/hers customers and customer treatments. The application was completed according to the commissioner's definitions and the commissioner was very satisfied with the result.

Keywords
AngularJS, HTML5 application, CRM, Mobile Angular UI, responsive design

Sisällys

1 Johdanto

Keväällä 2014 aloin kahden kanssaopiskelijan kanssa tutkimaan markkinoilta löytyviä asiakkuudenhallinnan ratkaisuja hierojille. Hierojalla on lain mukaan velvollisuus pitää asiakkaistaan rekisteriä, johon merkitään esimerkiksi potilaan henkilötiedot, osoitetiedot ja sairaudet (Laki terveydenhuollon ammattihenkilöistä 28.6.1994/559). Myös hieronnoista on pidettävä kirjaa ja rekisteristä on löydyttävä potilaalle annetut hoidot ja niiden kestot (Laki terveydenhuollon ammattihenkilöistä 28.6.1994/559). Halusimme tarjota hierojille sähköisen ratkaisun potilaskannan ylläpitämiseen, sillä mielestämme markkinoilta ei löytynyt kevyttä asiakkuudehallintajärjestelmää ja suurin osa hierojista piti edelleen asiakasrekisteriään paperisessa muodossa (Tiainen 5.10.2014).

Suunnittelimme sovelluksemme olevan kevyt asiakkuudenhallintasovellus, joka toimisi sekä työpöytä- että mobiilikäytössä. Koimme idean olevan toteuttamiskelpoinen pienelle kohderyhmälle, hierojille, sillä pienelle kohderyhmälle on helppo tarkentaa sovelluksen ominaisuuksia ja niitä ei tarvitse olla kovin paljon. Suuremman kohderyhmän tarpeita on vaikeampi hahmottaa ja verrattuna pienempään kohderyhmään joudutaan sovellukseen kehittämään useampi ominaisuus. Kehitystyötä voitiin tehdä osana HAAGA-HELIAn kurssia, josta oli apua sovelluksen idean hiomiseen.

Opinnäytetyössä käsitellään toteutettavan sovelluksen, Hierojankortin, asiakaspuolen toteutusta. Asiakaspää on kehitetty käyttäen AngularJS-sovelluskehystä. AngularJS on Googlen ylläpitämä JavaScript sovelluskehys, jonka avulla verkkosovelluksien asiakaspään toteutuksia on helppo toteuttaa ja ylläpitää. AngularJS ottaa myös kantaa sovelluksen rakenteeseen ja tarjoaa tähän ratkaisuja esimerkiksi moduuleilla. AngularJS valittiin projektiin entisen kokemuksen sekä suosion takia. Sen soveltuminen CRUD-sovellukseksi esimerkiksi tiedon sidonnan, direktiivien ja reitityksen myötä, vaikutti myös päätökseemme.

Opinnäyteyön toimeksiantajana toimii minun ja kahden opiskelijakaverin perusteilla oleva yritys Sovelluksen on tarkoitus toimia yrityksemme ensimmäisenä virallisena tuotteena. Opinnäytetyön tavoitteena on toteuttaa hierojan asiakkuudenhallintaan soveltuva Hierojankortti-palvelu julkaistavaan kuntoon.

Opinnäytetyö koostuu pääasiassa neljästä pääluvusta. Tietoperusta käsittelee HTML5 sovelluksen määritelmää ja työpöytä- sekä mobiiliystävällisen sovelluksen toteutusta responsiivisin keinoin. Lopuksi luvussa syvennytään AngulariJS-sovelluskehykseen tarkastelemalla sen yleisimpiä konsepteja ja ominaisuuksia. Sovelluksen suunnittelu -luku määrit-

telee toteutettavan sovelluksen raamit sekä kuvailee opinnäytetyön tavoitteet ja sovelluksen teknologiavalinnat. Sovelluksen toteutuksessa kuvaillaan opinnäytetyössä valmistunutta produktia ominaisuuksien näkökulmasta. Siinä perehdytään sovelluksen tiedostorakenteeseen, asiakaspään tekniseen toteutukseen ja käyttöliittymiin ja käytettävyyteen. Pohdinta-luku käsittelee opinnäytetyön haasteita, sovelluksen jatkokehitysehdotuksia sekä yleistä pohdintaa sovelluksesta.

Opinnäytetyö rajautuu sovelluksen asiakaspään toteutukseen. Toimeksiantajan pyynnöstä en juurikaan avaa taustajärjestelmää ja aihe koettiin myöskin olevan liian laaja yhteen opinnäytetyöhön. Tahdoin myös itse perehtyä enemmän AngularJS toteutukseen kiinnostuksesta johtuen.

1.1 Käsitteet

Ajax (Asynchronous JavaScript and XML) = tekniikka, jolla voidaan tehdä asynkronisia palvelinkutsuja. Asynkroninen kutsu ei vaadi sivun uudelleenlatausta, eikä keskeytä muuta ohjelmakoodia

Asiakaspää (client-side) = asiakas-palvelin sovellusarkkitehtuurissa käytettävä termi, jossa asiakaspäällä viitataan sovellukseen tai sovelluksen osaan, jonka ohjelmakoodi pyörii käyttäjän lokaalilla tietokoneella tai päätteellä. Tämä voi olla esimerkiksi selaimessa pyörivä JavaScript-koodi

Asiakkuudenhallinta (Customer Relationship Management) = asiakaslähtöinen ajattelutapa, jossa asiakassuhdetta huomioidaan liiketoiminnassa. Asiakkuudenhallinta käsittää alleen asiakkuudenhallintaan soveltuvan järjestelmän, jolla asiakassuhdetta voidaan ylläpitää ja parantaa

API (Application programming interface) = ohjelmointirajapinta. API:n avulla eri ohjelmat voivat keskustella keskenään

CRUD (Create Read Update Delete), CRUD sovellus = termi tyypillisimmille tiedon funktioille tietosäiliössä. CRUD-sovelluksen on tarjottava käyttäjälle mahdollisuus vähintään lukea, luoda, muokata ja poistaa sovelluksessa käsiteltävää dataa.

DOM (Document Object Model) = tapa jolla pystytään näyttämään ja vuorovaikuttamaan objekteihin HTML, XHTML ja XML dokumenteissa. Dokumentista muodostuu puumainen rakenne jota kutsutaan DOM-puuksi (DOM-tree)

Fork, forkata = alkuperäisen lähdekoodin kopioimista ja kopion muokkaamista omaksi versioksi

HTTP (Hypertext Transfer Protocol) = protokolla tiedon liikuttamiseen internetin yli hypertext muodossa

HTTP-pyyntö (HTTP-Request) = HTTP-siirtoprotokollan pyyntökutsu palvelimelle, johon palvelin vastaa lähettämällä sopivan vastauksen. HTTP-pyynnön GET-metodissa lähetetään viesti osana URL:ia ja POST-tyyppinen metodi kuljettaa datan HTTP-pyynnön rungossa ei osana URL:ia.

JSON (JavaScript Object Notation) = kieliriippumaton syntaksi tiedon tallettamiseen ja välittämiseen

Palvelinpää (server-side) = asiakas-palvelin sovellusarkkitehtuurissa käytettävä termi, jossa palvelinpäällä tarkoitetaan kaikkia operaatioita jotka tapahtuvat palvelimella. Esimerkiksi tietokanta ja sen operaatiot mielletään palvelinpään operaatioiksi

REST (Representational state transfer) = ohjelmiston arkkitehtuurimalli, jossa kuvataan ohjelmiston ja palvelimen välistä kommunikointia. Tyypillisesti REST-arkkitehtuurimallia käyttävä ohjelmisto kommunikoi palvelimelle käyttäen HTTP-protokollaa,

Selainmoottori (browser engine) = ohjelma joka osaa tulkita HTML, CSS ja JavaScript tekniikoita ja toteuttaa niillä kuvatut toiminnot

URL = Uniform resource locator. Osoite, jolla voidaan viitata esimerkiksi sivuun internetissä

2 Tietoperusta

Tässä luvussa käsitellään Hierojankortin tekniseen toteutukseen liittyviä teknologioita ja tietoperustaa. Luku keskittyy HTML5 sovellukseen käsitteenä, kertoo yleiset periaatteet responsiivisesta suunnittelusta sekä syventyy AngularJS JavaScript-sovelluskehykseen esitellen sen yleisimpiä konsepteja ja ominaisuuksia.

2.1 HTML5-sovellus

HTML5 voi merkitä kahta eri asiaa riippuen kontekstista. Se tarkoittaa nykyistä HTML-kielen viidettä kehitysversiota ja sovelluksen kehittämisen tapaa (Lehdonvirta & Korpela 2013, 12). HTML5 standardi on HTML-kielen viides versio ja sen kehittäminen aloitettiin jo vuonna 2004 (W3C 2014a). HTML5 kieli tuo mukanaan lukuisia uusia HTML-elementtejä ja ominaisuuksia. Eri selainten välillä kaikki HTML5 ominaisuudet eivät toimi aina samalla tavalla tai lainkaan. Selainten erilaiseen käyttäytymiseen on kehitetty Modernizr JavaScript-kirjasto, jolla on mahdollista saada HTML5 ja CSS3 ominaisuudet toimimaan niitä tukemattomissa selaimissa (Ate 22.6.2010).

HTML5-sovellus tarkoittaa käsitteenä sovellusta joka on toteutettu HTML, CSS ja JavaScript tekniikoilla uusimpien HTML5 standardien mukaan ja joka on tarkoitettu toimimaan selainmoottorilla. Selainmoottori on osa selainta ja se on vastuussa ohjelmakoodin tulkitsemisesta ja näyttämisestä. HTML5-sovelluksen perustuminen avoimiin tekniikkoihin ja selainmoottorin käyttöön tekee siitä lähes alustariippumattoman, sillä melko lailla kaikki nykyiset työpöytä- ja mobiililaitteet tukevat HTML5-sovelluksia. HTML5-sovellus ei ole siis riippuvainen laitteesta tai käyttöjärjestelmästä vaan selaimen suorituskyvystä ja sen ominaisuuksista. Selaimen suorituskyky ei ole päässyt siihen pisteeseen, että se voisi kilpailla natiivisovellusten kanssa. (Lehdonvirta & Korpela 2013, 13–15.)

Natiivista mobiilisovelluksesta puhuttaessa tarkoitetaan sovellusta, joka on toteutettu kielellä, jota alusta tukee suoraan. Natiivi mobiilisovellus toteutetaan alustakohtaisesti ja ei ole näin monistettavissa useisiin laitteisiin. Natiivi rajapinta mahdollistaa pääsyn käsiksi alustan komponentteihin kuten kameraan ja paikannukseen, ja sovelluksen suoritusteho ja nopeus ovat huimasti edellä HTML-5 sovellusta. (Ballve 2013; WhatIs.com 2013.) Usein kuitenkin HTML5-sovelluksen tekeminen on natiivisovellusta nopeampaa sen alustariippumattomuuden takia.

Mobiilisovellukset natiivin ja HTML5 toteutuksen rajoilla ovat hybridisovelluksia. Hybridisovellus on pakattu HTML5-sovellus, joka käyttää hyväkseen alustan natiivia rajapintaa jonkin toiminnallisuuden toteuttamiseen. Hybridisovelluksissa natiiviin toiminnallisuuteen

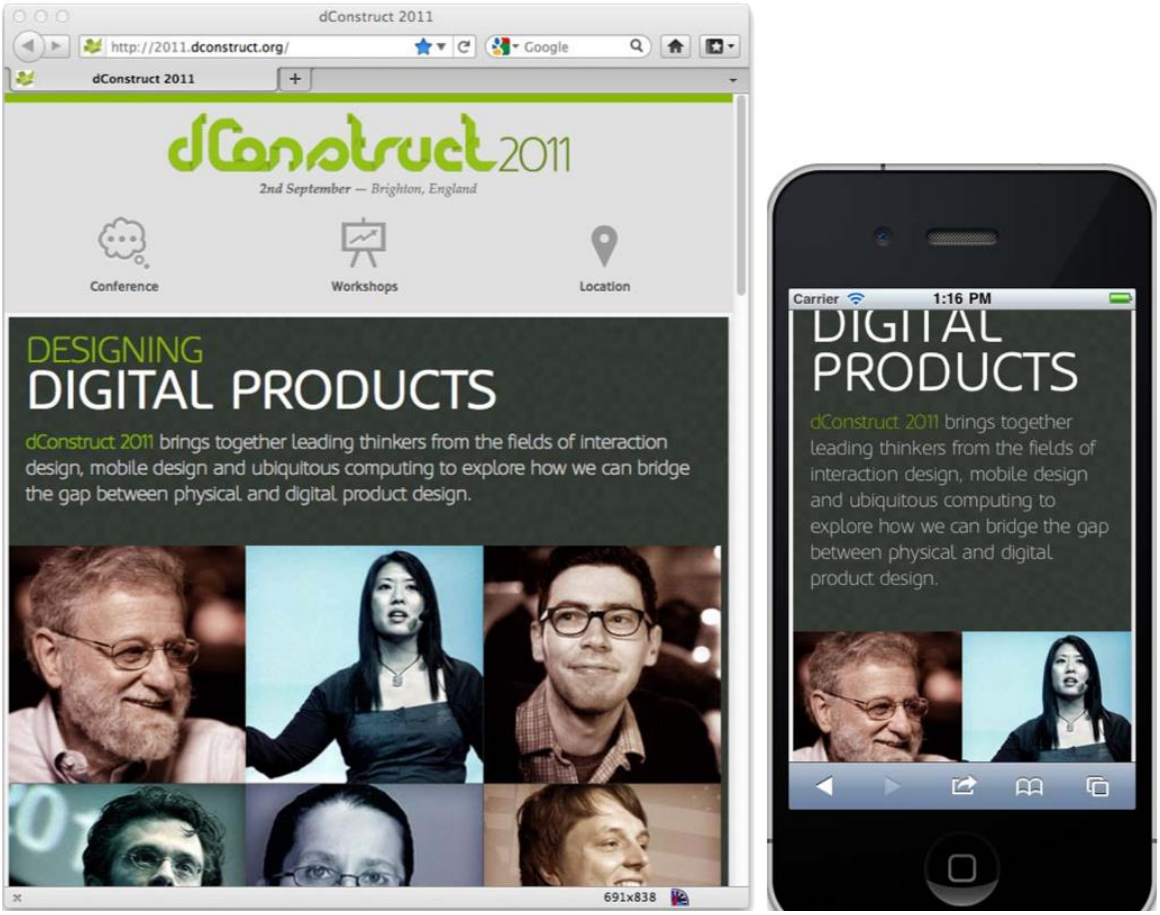
päästään käsiksi esimerkiksi Phonegap sovelluskehyksellä, joka mahdollistaa hybridisovellusten ohjelmoinnin JavaScriptilla. Tällöin kehys vastaa JavaScriptin tulkkaamisesta alustan natiiviin muotoon (Phonegap 2015). Pelkkä sovelluksen paketoiminen esimerkiksi Phonegapin avulla ei Lehdonvirran ja Korpelan mukaan riitä nimittämään sitä hybridisovellukseksi (Lehdonvirta & Korpela 2013, 13–15). Termiä hybridisovellus kuitenkin käytetään usein myös paketoiduista HTML5-sovelluksista ja hybridisovelluksen rajan määrittäminen HTML5-sovelluksen kanssa on hankalaa.

2.2 Responsiivinen suunnittelu

Nykyään internetsivut suunnitellaan yleensä responsiivisiksi, mikä tarkoittaa sitä, että sivun elementit pienenevät ja suurenevat (skaalautuvat) laitteen näytön koon mukaan. Ennen responsiivisen suunnittelun yleistymistä, sivut suunniteltiin käyttäen kiinteitä leveyksiä ja korkeuksia elementtien kokojen määrittelyssä. Kiinteä leveys ei ota huomioon näytön kokoa, vaan pysyy samankokoisena näytöstä riippumatta. Mobiililaitteiden yleistyessä kiinteät rajat eivät enää toimineet, sillä mobiilissa sivut menivät näytön rajojen ulkopuolelle ja elementit olivat joko liian pieniä tai suuria (Marcotte 2010). Sivuista saatettiin tehdä toinen versio mobiililaitteelle ja mahdollisesti vielä kolmas tabletille, jolloin sivujen ylläpito on vaikeaa.

Responsiivisen sivun rakentaminen mahdollistaa sen, ettei sivuista tarvitse rakentaa eri versioita eri laitteille, vaan sama versio toimii erikokoisilla laitteilla. Responsiivinen suunnittelumalli sopii hyvin HTML5-sovelluksille, jotka pyritään muutenkin rakentamaan kaikille laitteille sopivaksi yhdellä kertaa. HTML5-sovelluksen responsiivisuus toteutetaan samalla tavalla, kuin verkkosivujen responsiivisuus ja käytännössä niillä ei ole eroa. (Marcotte 2010.)

Responsiivisessa sivussa pyritään tekemään elementit niin, ettei niille ole kiinteitä leveyksiä, vaan elementtien koot määritellään prosentteina. Näin elementin koko on yhteydessä selaimen ikkunan kokoon ja pienenee tai laajenee sen mukaan. (Marcotte 2010).



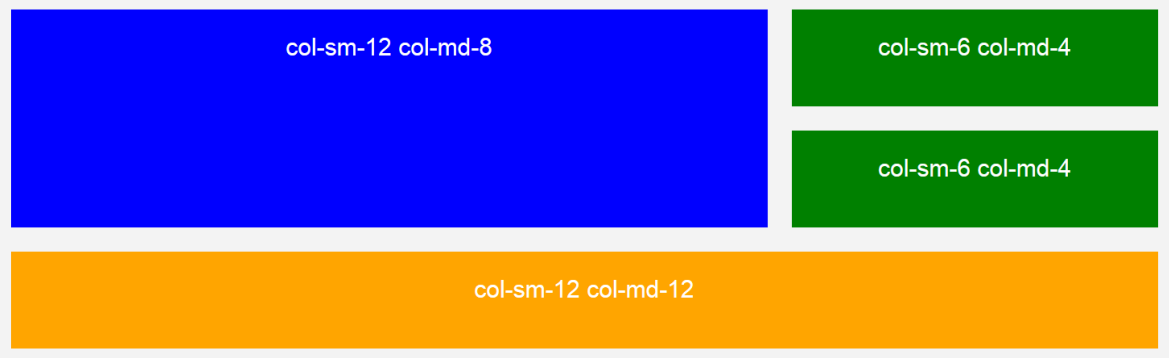
Kuva 1 Tabletti- ja mobiilinäkymä dConstruct-verkkosivusta (Frain 2012)

Eri laitteille, näytönko'oille ja orientaatioille voidaan määrittää omia tyylejä mediakyselyillä (media-query). Mediakysely on CSS:n kolmannessa versiossa (CSS3) tullut ominaisuus ja sen avulla ohjelmoija voi määritellä elementeille eri CSS-tyylejä esimerkiksi näytönkoosta riippuen. (MDN 2015). Kuvassa 1 selaimen ikkunaa on raahattu pienemmäksi (oikealla), jotta se näyttäisi tabletin ruudulta. Sivulle on määritelty mahtuvan kolme kuvaa per rivi. Mobiilin näytöllä mahtuu kuvia vierekkäinen kaksi ja sisältö on supistunut vastaamaan ruudun kokoa. Muutokset tabletin ja puhelimen välillä on tehty käyttäen mediakyselyitä CSS-määrityksissä.

Responsiivisien sivujen tekoon on luotu monenlaisia valmiita HTML, CSS- ja JavaScript-sovelluskehyksiä. Yksi suosituimmista kehyksistä on avoimella lähdekoodilla tehty Bootstrap. Bootstrap on kirjoitushetkellä Github -palvelun pidetyin ohjelma miltei 80 tuhannella tähtimerkinnällä ja noin 30 tuhannella forkkauksella (Github 2015). Sen nykyinen versio Bootstrap 3 on suunniteltu mobiili ensin (mobile first) -kehitysfilosofialla. Mobiili ensin on suunnittelufilosofia, jossa käyttöliittymä ja käytettävyys suunnitellaan ensin mobiilille ja tämän jälkeen vasta työpöydälle.

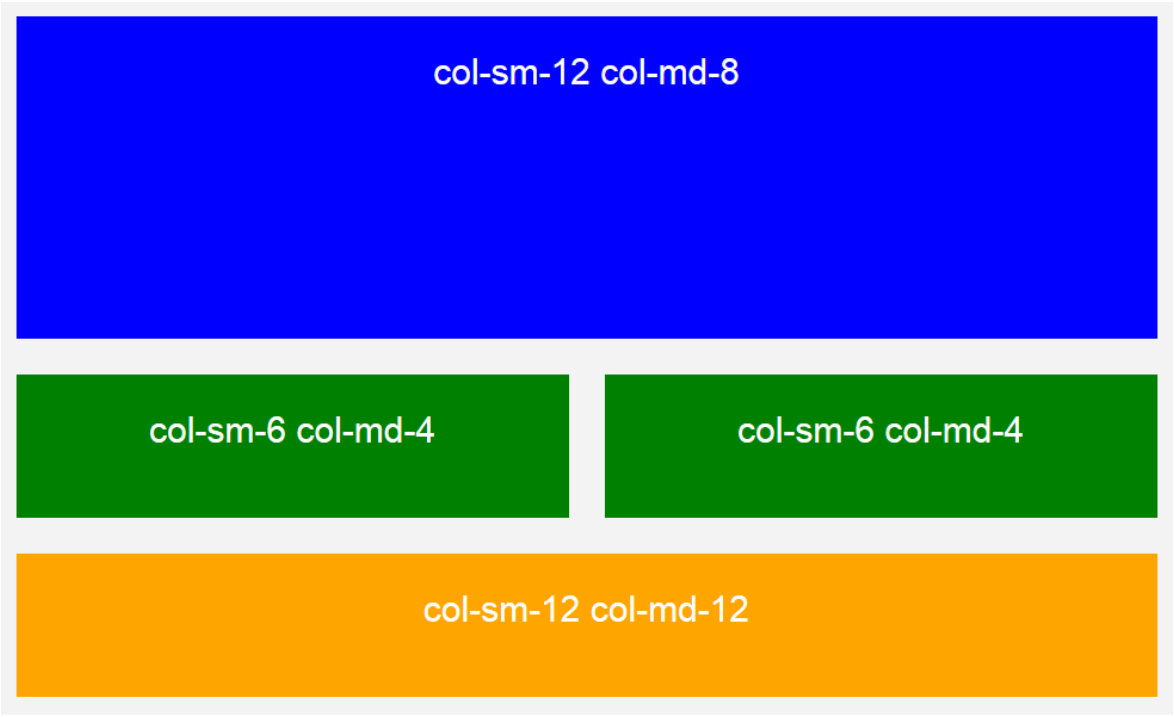
Bootstrap on flat design -tyylinen. Flat design taas tarkoittaa tyyliä, jossa pyritään yksinkertaisiin ja minimalistisiin elementteihin ja jossa ei käytetä esimerkiksi tekstuurisia ja kolmiulotteisia elementtejä (Bootstrap 19.8.2013; Cousins 2013).

Yksi tärkeimmistä ominaisuuksista responsiivisissa sovelluskehyksissä on skaalautuvat ruudukot (grid system). Tämä ominaisuus löytyy myös Bootstrapista. Bootstrapissa skaalautuvat ruudukot ovat mediakyselyihin rajattuja CSS-luokkia, jotka vaikuttavat elementin kokoon riippuen näytön leveydestä. Yhdelle ruudulle annetaan määreenä sen tähtäämä ikkunan koko, jotka voivat olla xs (erittäin pieni), sm (pieni), md (keskikokoinen) ja lg (suuri). Ruudulla on myös sille määrättävä koko-attribuutti, joka on maksimissaan 12 (100% isäntä-elementistään) ja minimissään 1 (10% isäntä-elementistään). Luokat nimetään alkamaan ”col” alkuisiksi ja esimerkiksi col-lg-6 tekee ruudun, joka on puolet isäntä-elementistään. (Bootstrap 2015.)



Kuva 2 Bootstrapin ruudukko keskikokoisessa ikkunassa (ikkunan koko ⩾ 992px)

Kuvassa 2 on tehty neljä ruutua, jotka mukautuvat kooltaan erilailla eri ikkunan kooissa. Koska käytämme keskikokoista ikkunan kokoa, joka on 992 pikseliä ja sitä suurempi, vaikuttavat col-md alkuiset luokat. Esimerkissä sininen ruutu on määritelty korkeammaksi kuin muut ruudut ja sille on annettu leveys-attribuutiksi 8 (col-md-8). Sen viereiset vihreät ruudut mahtuvat näin sen oikealle reunalle, sillä ne on määrätty kokoon 4 (col-md-4). Tämä muodostaa koko rivin, sillä ruutuja yhteiskoko on 12 (8+4). Keltainen ruutu on määrätty koko rivin kokoiseksi määreellä col-md-12. (Bootstrap 2015.)



Kuva 3 Bootstrapin ruudukko pienessä ikkunassa (ikkunan koko ⩾ 768px)

Kun selainikkunan kokoa raahataan alle 992 pikselin, hyppää sininen ruutu kokolevyiseksi ja vihreät ruudut sen alle (kuva 3). Tämä tapahtuu luokkien määrittelyillä sillä nyt sininen ruutu on määritelty kokolevyiseksi (col-sm-12) ja vihreät ruudut vievän puolet tilasta (col-sm-6). Keltainen ruutu pysyy edelleen kokolevyisenä. (Bootstrap 2015.)

2.3 AngularJS

AngularJS on Googlen ylläpitämä JavaScript-sovelluskehys. Se kehitettiin web-sovellusten kehittämisen ja ylläpidon helpottamiseksi. Sen ensimmäisen version kehittivät Misko Hevery and Adam Abrons. Myöhemmin kaksikosta Abrons lopetti projektissa työskentelyn. Hevery jatkoi työtä perustamansa ryhmän kanssa Googlella ja projekti sai nykyisen nimensä AngularJS. (Austin 2014.)

AngularJS-sovelluskehys on tarkoitettu dynaamisten web-sovellusten kehitykseen. Se soveltuu erityisen hyvin CRUD-sovellukseen (Create Read Update Delete) sen tarjoaman ominaisuuksien kuten tiedon sidonnan, direktiivien ja AJAX-palveluiden myötä. Kaikki Angularin ohjelmakoodi pyörii selaimessa, eikä näin ollen ole riippuvainen palvelinpuolesta.

Angular-sovellus muodostuu moduuleista, jotka ovat voivat olla toisistaan riippuvaisia. Moduulit helpottavat ohjelmakoodin hallittavuutta ja testattavuutta, koska ohjelmakoodi jakautuu palasiin, joita on helppo testata ja käyttää uudelleen. (Bégaudeau 24.4.2014.)

Angularin avulla DOM:iin on helposti liitettävissä omia rakenteita, joita Angular kutsuu direktiiveiksi (directive). Direktiivit voivat toimia pohjina HTML-elementeille, jolloin direktiivi voi koostua monesta HTML-elementistä ja syntaksi on yksinkertaisempaa. Yleensä direktiivit kuitenkin sisältävät DOM:in manipulointiin liittyvää logiikka, jolloin ne muuttuvat tavallisista staattisista HTML-elementeistä, interaktiivisiksi komponenteiksi.

Yksi Angularin peruspilaresta on myös tiedon sidonta (data-binding), joka pitää huolen siitä, että datan muutokset näkyvät suoraan loppukäyttäjälle. Tähän ei tarvitse ylimääräistä ohjelmakoodia, vaan Angular tarkastelee muuttujia ja automaattisesti päivittää niiden muutokset. (AngularJS 2015a; Bégaudeau 24.4.2014.)

2.3.1 Yksisivuinen sovellus (single-page application)

Web-sovelluksia on perinteisesti tehty niin, että palvelimelle tehtävät pyynnöt päivittävät sivun kokonaan. Näin ollen sivulla olevat tiedot katoavat ja ne on pakko rakentaa uudelleen. Yksisivuisessa sovelluksessa sivua ei päivitetä käytön aikana, vaan se hyödyntää usein Ajax-tekniikkaan ladatakseen tietoa palvelimelta. (Mikowski & Powell 2013.)

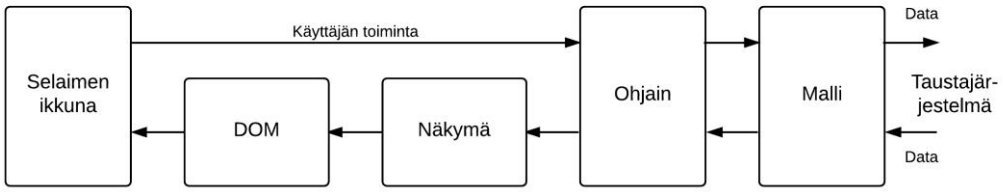
Yksisivuiset sovellukset ovat raskaampia selaimelle, koska suurin osa sovelluksen logiikasta toteutetaan selaimen koodina. Palvelimenkoodin tehtäväksi jää yleensä autentikointi, tiedon validointi sekä tietokantahaut. Perinteisemmässä sovellusarkkitehtuurissa asiakaspään puolella tehdään useimmiten vain toimenpiteitä elementtien ja sivujen visualisoimiseen liittyen. (Mikowski & Powell 2013.)

AngularJS-sovelluskehyksessä yksisivuisen sovelluksen toteuttaminen on helppoa. Siihen tarvitaan reititin (router), joka vastaa sivujen hallinnoimisesta. Reitittimen tehtävänä on ladata sivuun HTML-pohja (template) sivun osoitteen mukaan. Pohjan lataus ei aiheuta sivun uudelleenlatausta ja yhdellä sivulla voi olla useita pohjia. Pohjan lisäksi reitittimeen voi asettaa muitakin arvoja kuten pohjan ohjaimen. Näin sivu voi muodostua ikään kuin pienistä ohjelmista, joiden logiikka voi olla riippumatonta sivun muista palasista. (AngularJS 2015b.)

2.3.2 Angular MVC

MVC (Model-View-Controller), tarkoittaa sovellusarkkitehtuuria, jossa sovelluslogiikka ja näyttölogiikka pyritään erottamaan toisistaan. Tämä tarkoittaa asiakaspuolen sovelluksessa sitä että tieto, tiedon käsittelylogiikka ja HTML-elementit erotetaan toisistaan. (Freeman 2014.)

MVC muodostuu kolmesta osasta: mallista (model), näkymästä (view) ja ohjaimesta (controller). Malli pitää sisältää sivun datan ja logiikan siitä, kuinka mallia voi muokata. Ohjain toimii mallin ja näkymän välissä ja on vastuussa näkymälle näytettävästä datasta ja sen muokkaamisesta. Näkymä pitää sisällään logiikan datan näyttämisestä ja sen muotoilusta.



Kuva 4 Kaavio AngularJS MVC arkkitehtuurista (Freeman 2014)

AngularJS:n MVC arkkitehtuurin mukaan asiakaspuolen toteutus saa datan taustajärjestelmästä yleensä RESTful API:n kautta. Kuvassa 4 esitetään tiedon liikkumista MVC-arkkitehtuurisessa sovelluksessa. Taustajärjestelmästä tuleva data tuodaan malliin josta ohjain voi muokata ja ohjata dataa logiikkansa mukaan näkymälle. Näkymä näyttää dataa ja manipuloi DOM-elementtejä sille määrättyjen ehtojen mukaan. Lopulta ohjain saa käyttäjän toiminnasta palautetta ja riippuen toiminnosta, se voi muuttaa mallia, jonka yhteydessä näkymän tiedot muuttuvat. (Freeman 2014.)

Selventääkseen MVC:n osien tehtäviä voidaan ajatella mitä osat eivät tee. Malli ei saa sisältää tietoa, jonka mukaan on mahdollista selvittää palvelinpuolen logiikka. Mallin ei käytännössä pitäisi olla tietoinen taustajärjestelmän toiminnoista. Sen ei pidä sisältää ehtoja, jotka muokkaavat tietoa käyttäjän toiminnan mukaan. Lisäksi malli ei saa sisältää logiikkaa tiedon näyttämiseen liittyen. (Freeman 2014.)

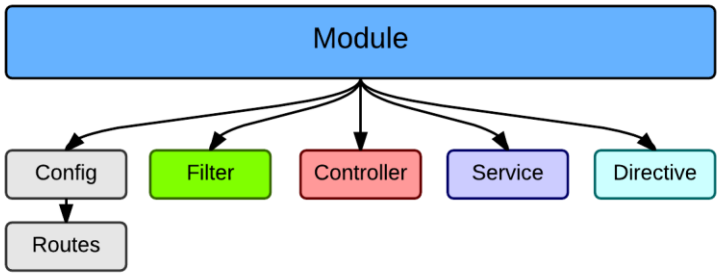
Ohjaimen ei pitäisi sisältää logiikkaa, joka hallinnoi tiedon eheyttä, malli pitää tästä huolen. Ohjaimen ei myöskään tulisi manipuloida DOM:ia, kaikki nämä operaatiot tulisi suorittaa näkymä. On myös muistettava, että ohjaimen ei kuulu muokata tietoa skoopin (scope) ulkopuolelta. (Freeman 2014.)

Näkymän ei tule sisältää monimutkaista logiikkaa, sillä tämä kannattaa silloin kuormittaa ohjaimen vastuulle. Näkymällä ei myöskään ole asiaa mallin tietoihin: se ei saa luoda, muuttaa tai tallettaa sitä. Näkymän sisältää tiedon siitä mitä käyttäjä näkee ja logiikan DOM manipuloinnista. (Freeman 2014.)

Jos malli, näkymä ja ohjain muutetaan ohjelmakoodiksi, malli olisi JavaScript objekti, näkymä DOM ja ohjain JavaScript luokka. Tämän jaottelun pohjimmainen tarkoitus on pitää JavaScript koodi testattavana, hallittavana, laajennettavana ja ymmärrettävänä. (Green.)

2.3.3 Moduulit

Moduuleilla pakataan eri ohjelmakokonaisuuksia paketeiksi, jolloin niitä on helppo käyttää muissakin sovelluksissa. Ne antavat ohjelmalle kontekstin, jolloin on helpompi ymmärtää kyseisen kokonaisuuden tehtävää, tämä voi auttaa myös tiedostorakenteen määrittelyssä. Moduuleita on helppo testata, kun ohjelmakoodi on jaettu pienemmiksi kokonaisuuksiksi, jotka voivat toimia riippumatta muusta ohjelmasta. (Bégaudeau 24.4.2014; Rammer & Weyer 2013.)



Kuva 5 Moduuli ja siihen yhdistettävissä olevia komponentteja (Wahlin 16.8.2013)

Kuvassa 5 demonstroidaan mitä kaikkia osia moduuliin voi yhdistää. Kaikkia osia ei kuitenkaan tarvitse käyttää, esimerkiksi jotkut moduulit voivat koostua pelkästään direktiivestä (directive) tai palveluista. Esimerkiksi DOM:in animointiin liittyvä moduuli voisi koostua pelkästään direktiiveistä.

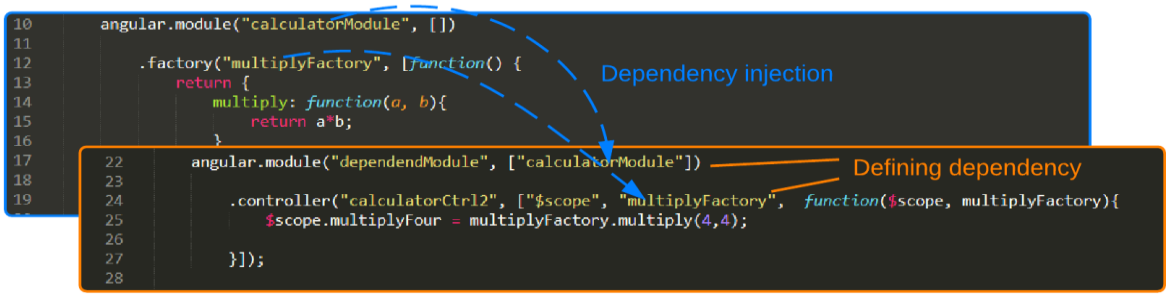
On järkevää rakentaa moduulit niin, että niiden alta löytyvät tarvittavat osat, jonkin tietyn ohjelman tai ominaisuuden toimimiseen. Mikäli jonkin ohjelman osa voidaan irrottaa kokonaisuudesta ja liittää toiseen ohjelmaan kannattaa siitä luultavasti rakentaa moduuli.. (Rammer & Weyer 2013.)

Angularissa moduuli määritellään angular.module-avainsanalla. Tämän jälkeen sille annetaan nimi ja riippuvuudet (dependencies). Viittaamalla moduuliin nimeen tai ketjuttamalla (chaining), voidaan sille sen jälkeen liittää komponentteja kuten ohjaimia ja palveluita.

```
10      angular.module("calculatorModule", [])
11
12          .factory("multiplyFactory", [function() {
13              return {
14                  multiply: function(a, b){
15                      return a*b;
16                  }
17              }
18          }])
19          .controller("calculatorCtrl", ["$scope", "multiplyFactory",  function($scope, multiplyFactory){
20              $scope.multiplyTwo = multiplyFactory.multiply(2,2);
21
22          }]);
```

Kuva 6 Moduulin määrittely ja komponenttien kiinnittäminen siihen

Kuvassa 6 demonstroidaan ohjaimen ja palvelun liittämistä moduuliin. Rivillä 10 luodaan moduuli, joka nimetään nimellä calculatorModule. Pilkun jälkeen tulevat hakasulkeet, joihin määritellään sen riippuvuudet. Tällä moduulilla ei ole riippuvuuksia, jolloin lista jää tyhjäksi. Moduulille on ketjutettu palvelu ja ohjain (factory, controller). Mikäli jokin toinen moduuli haluaa käyttää calculatorModule -moduulia, sille täytyy määritellä riippuvuus calculatorModule -moduuliin.



Kuva 7 Riippuvuuksien määrittely ja injektointi

Kuvassa 7 on luotu moduuli dependendModule ja sille on määritelty riippuvuus calculatorModule-moduuliin. Kun moduulille on määritelty riippuvuus toiseen moduuliin, se saa käyttöönsä siinä olevat komponentit. Riippuvaan moduuliin on määritelty ohjain, jolle on määritelty riippuvuus toisen moduulin palveluun. Angularin niin sanottu riippuvuuden injektointi (dependency injection), pitää huolta siitä, että ohjaimessa voidaan käyttää kyseistä palvelua. Riippuvuuksien injektoija on suuressa roolissa koko angular kehityksessä ja se mahdollistaa eri komponettien käytön muissa komponenteissa. (AngularJS 2015c.)

2.3.4 Ohjaimet ja skoopit

Ohjain (controller), on linkki näkymän ja mallin välissä, ikään kuin tulkki, joka voi myös muuttaa sille tullutta sanomaa. Näkymälle se tarjoaa eri toiminnallisuuksia ja näkymän datan. Ohjain muokkaa mallia käyttäjän toimintojen mukaan. Angular-sovelluksessa käytetään jatkuvasti ohjaimia ja se on yksi angularin peruskomponenteista. Ohjaimet käyttävät tiedon muokkaamiseen ja välitykseen skooppeja (scope). Skooppi on kontrolleriin in-

jektoitava objekti, jota voi pitää ”liimana” ohjaimen ja näkymän välillä. Kaikki näkymälle tuotava tieto on talletettava skooppiin. (Freeman 2014.)

```
 8    <script>
 9        angular.module("controllerExample", [])
10            .controller("yourFirstCtrl", ["$scope", function($scope) {
11
12                $scope.firstName = "James";
13
14                $scope.getLastName = function(firstName){
15                    return $scope.firstName === "James" ? "Bond" : "Somebody";
16                }
17
18            }]);
19    </script>
20
```

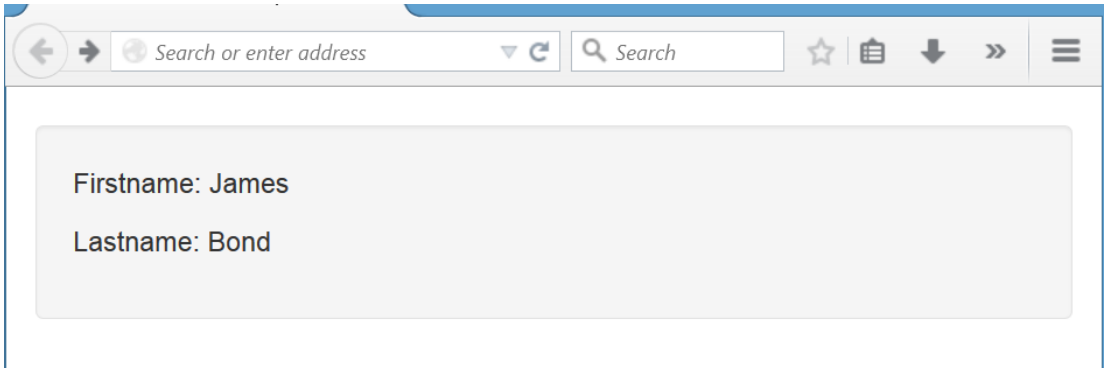
Kuva 8 Yksinkertaisen ohjaimen määrittely

Kuvassa 8 määritellään ensin moduuli controllerExample, jolle ei injektoida muita moduuleita (rivi 9). Tämän jälkeen moduulille määritellään yksinkertainen ohjain (rivi 10), jolle injektoidaan $scope-objekti. Skooppiin talletetaan muuttuja firstName (rivi 12), jota voidaan nyt käyttää suoraan näkymässä.

```
27    <div class="container" ng-controller="yourFirstCtrl">
28        <div class="well">
29            <p>Firstname: {{firstName}}</p>
30            <p>Lastname: {{getLastName(firstName)}}</p>
31        </div>
32    </div>
```

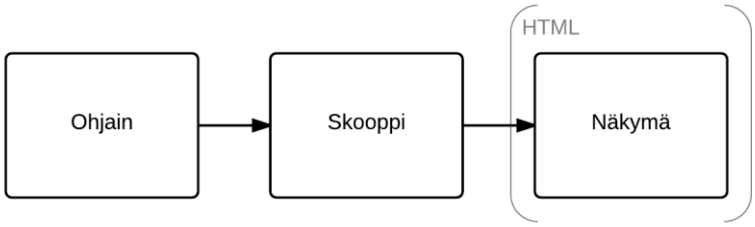
Kuva 9 Esimerkki siitä, kuinka skoopin muuttujia ja funktioita voidaan kutsua näkymässä (DOM:issa)

Kuvassa 9 määritellään ohjain avainsanalla ng-controller (rivi 27). Myöhemmin skoopin muuttujia ja funktioita voidaan kutsua näkymässä ohjaimessa määrätyillä nimillä (rivi 29-30). Lopputulokseksi selain näyttää ensin ohjelmassa ohjaimessa määrätyn nimen ja tämän jälkeen käyttäen ohjaimessa alustettua funktiota hakee sukunimen etunimen perusteella (kuva 10).



Kuva 10 Selaimen tuottama representaatio esimerkkikoodista

Esimerkissä käytetään niin sanottua monoliittista ohjainta. Tämä tarkoittaa sitä, että yksi ohjain on määritelty kaikille sovelluksen HTML elementeille, jolloin näkymän ja ohjaimen suhde on yksi yhteen (kuva 11). Tämä menetelmä on helposti ymmärrettävissä, mutta ohjainkoodi voi tulla liian suureksi ja näin vaikeaksi hallita ja jaotella.

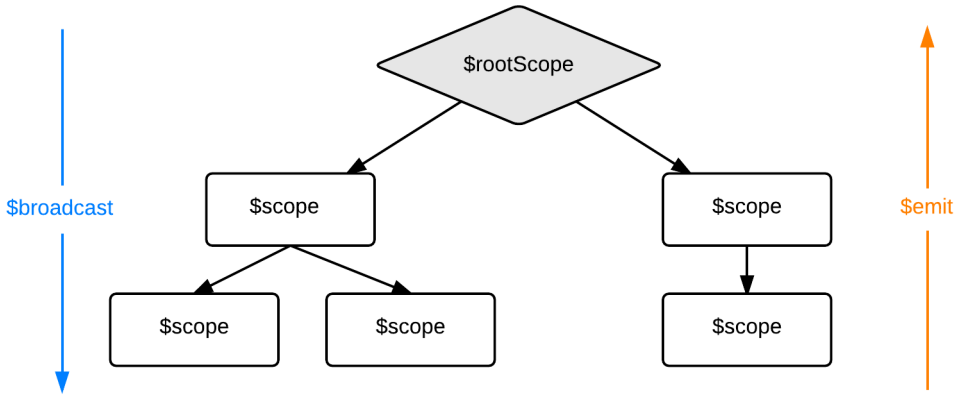


Kuva 11 Monoliittinen ohjain

Monoliittinen ohjain voi soveltua pieniin ohjelmiin ja esimerkkiohjelmiin, mutta useimmin ohjaimia on yhdessä sovelluksessa useita. Tiedon jakaminen eri ohjainten kesken voidaan toteuttaa vaikkapa periyttämisellä ja kaikille skoopeille yhteisen juuriskoopin (root scope) avulla. (Freeman 2014.)

Ohjainten periyttämisessä määritellään isäntäohjain (parent controller), jonka ominaisuudet ja tiedot periytyvät aliohjaimelle (child controller). Myös aliohjaimista voi periyttää lisää ohjaimia. Ohjainten määrää ei siis ole rajattu.

Periyttäminen ei ratkaise kaikkia ohjainten tiedon kanssakäymisen ongelmia. Periytys mahdollistaa muuttujien ja toiminnallisuuksien jakamisen yliohjaimesta alaspäin, mutta aliohjain ei esimerkiksi suoraan voi muuttaa isäntänsä tietoja. Kaikille skoopeille yhteisellä juuriskoopilla voidaan ratkaista tämä ongelma. (Freeman 2014.)



Kuva 12 Skooppien hierarkinen järjestys. Broadcast-metodi vie tietoa alaspäin sen alla oleville skoopeille ja emit-metodi vie alaskooopin tiedot ylöspäin

Juuriskooppia ei angular-sovelluksessa ole kuin yksi ja se on aina hierarkisesti skooppien ylin jäsen. Jokainen skooppi luodaan juuriskooppi -palvelussa ja se tarjoaa esimerkiksi metodeita tiedon välittämiseen eri skooppien välillä. Metodeista $broadcast:in avulla voidaan yliohjaimen skoopin tiedot välittää alaspäin sen alla oleville ohjaimille. $Emit -metodilla tiedot liikkuvat aliohjaimesta ylöspäin isäntäohjaimille. Kuva 6 demonstroi skooppien hierarkiaa ja skooppien välistä tiedon liikkumista $broadcast ja $emit metodeilla. (Freeman 2014.)

2.3.5 Tiedon sidonta (data binding)

Angularin vahvimpiin perusominaisuuksiin kuuluu tiedon sitominen (data binding). Tämä tarkoittaa sitä, että voidaan määritellä, että tiedon näkyy muutos suoraan näkymässä, jos tieto muuttuu. Tätä tiedon sidontaa kutsutaan kaksisuuntaiseksi tiedon sitomiseksi (two-way data binding). Kaksisuuntaisessa sitomisessa näkymässä tapahtuneet muutokset vaikuttavat suoraan skoopissa oleviin tietoihin ja skoopin manipulaatiot muuttavat näkymässä olevaa tietoa suoraan. (AngularJS 2015d; Bégaudeau 24.4.2014).

```
27      <div class="container" ng-controller="yourFirstCtrl">
28          <div class="well">
29              <p>Firstname: {{firstName}}</p>
30              <p>Lastname: {{getLastName(firstName)}}</p>
31              <input type="text" ng-model="firstName" />
32          </div>
33      </div>
```

Kuva 13 Input-elementtin on sidottu firstName-muuttuja ng-model-direktiivillä

Demonstroidakseen kahdensuuntaista tiedon sidontaa on kuvassa 13 esimerkkiohjelman DOM-puuhun lisätty input-elementti, johon on määritelty attribuutti ng-model. Ng-model on

direktiivi, joka on ottanut arvokseen skoopin firstName-muuttujan. Nyt käyttäjän on mahdollista muokata muuttujaa suoraan näkymässä ja tiedon muutos suoraan käyttäjälle, kuten kuva 14 esittää.

Kuva 14 Selaimen näkymä käyttäjän muuttaessa kentän arvoa

Tiedon sitominen tapahtuu Angularin lähdekoodissa niin, että aina kun näkymään asetetaan muuttuja, Angularin on talletettava se niin sanottuun katselulistaan (watchlist). Katselulista pitää huolen siitä, että tiedon muutokset rekisteröidään ja siihen voidaan reagoida niin sanotulla dirty checking -tekniikalla. Dirty checking tarkoittaa sitä, että kun tieto muuttuu mallissa, jokainen katselulistan muuttuja tarkistetaan muutoksen varalta. Muutosten arviointi vie sitä enemmän resursseja mitä enemmän muuttujia katselulistalla on. Mikäli tiedetään, ettei muuttujaa haluta sitoa kahdensuuntaisesti vaan asettamisen jälkeen sen muutoksia ei haluta seurata, voidaan se sitoa yksisuuntaisesti (one-time data binding). Yksisuuntainen sitominen ei talleta muuttujaa katselulistaan, jolloin sen muutos ei käynnistä listan läpikäymistä. Yksisuuntainen sitominen saadaan aikaan käyttämällä syntaksissa kahta kaksoispistettä muuttujan nimen edessä. (Precht 14.10.2014; Lerner 2013, 426-432.)

2.3.6 Direktiivit (directives)

Freemanin mukaan Angularin MVC-arkkitehtuurissa DOM-manipulaatiot kuuluvat näkymälle ja niitä ei jaottelun mukaan saisi tehdä ohjaimessa. Tätä näkymän logiikkaa suoritetaan direktiiveissä. Direktiivit ovat liitettävissä HTML-elementteihin tai niillä pystytään luomaan kokonaan uusia elementtejä. (Freeman 2015.)

```
2   <my-directive></my-directive>
3   <div my-directive></div>
4   <div class="my-directive"></div>
5   <!-- directive: my-directive -->
```

Kuva 15 Direktiivien määrittely HTML:ssä

Kuvassa 15 esitetään direktiivien määrittelyä HTML-dokumentissa. Direktiivi voidaan teh-

dä kokonaan uutena elementtinä (rivi 2), liittää HTML-elementin attribuutiksi (rivi 3), liitää HTML-elementin luokaksi (rivi 4) tai määrittää HTML-kommenttina (rivi 5).

```
7   <div my-directive my-url="https://google.com" my-link-text="Click to go Google"></div>
```

Kuva 16 Direktiivi, johon on määritelty kaksi arvoja vastaanottavaa attribuuttia

Direktiiveihin on mahdollista liittää arvoja, joita voidaan käyttää direktiivin ohjelmakoodissa. Kuvassa 16 direktiiville on määritelty kaksi attribuuttia my-url ja my-link-text. Normaalisti direktiivin käytössä ovat kaikki skooppiin liitetyt muuttujat ja arvot, mutta on järkevämpää lähettää direktiiville sen tarvitsema tieto, jotta direktiivi ei sekoittaisi muuta ohjelmaa. (Lerner 2013, 73–107.)

```
22  .directive('myDirective', function() {
23      return {
24          restrict: 'A',
25          replace: true,
26          scope: {
27              myUrl: '@',
28              myLinkText: '@'
29          },
30          template: '<a ng-href="{{myUrl}}">{{myLinkText}}</a>'
31      }
32  });
```

Kuva 17 Direktiivin JavaScript-koodi

Kuvassa 17 esitellään direktiivin määrittelyä koodissa. Ensimmäiseksi määritellään direktiivin nimi, jota Angular vertaa HTML:stä löytyvään nimeen (rivi 22). Jotta Angular pystyy yhdistämään ohjelmoijan luoman direktiivin ja HTML:ään määritellyn elementin, pitää direktiivin koodiin määritellä, minkälaista elementtiä Angularin tulee etsiä. Tämä yhdistäminen tapahtuu direktiivin restrict-avainsanalla. Direktiivi on määritelty sidotuksi HTML-attribuuttiin (A), muita mahdollisuuksia olisivat E (elementti), C (luokka) ja M (kommentti). Oletuksena Angular etsii saman direktiivin nimistä HTML-attribuuttia ja HTML-elementtiä. Rivillä 25 määritellään, että direktiivi korvaa normaalisi HTML:stä löytyvän elementin (replace) ja tämä korvataan direktiivin pohjalla (template).

Direktiiville on mahdollista luoda oma skooppi (isolated scope). Tätä demonstroi kuvan 17 rivit 26–29. Skoopin sisälle määritellään kaksi muuttujaa ja niiden arvoiksi @-merkki. Merkki määrittää kuinka skooppiin tullut muuttuja toimii. Vaikka tässä tapauksessa käyte-

tään @-merkkiä, voidaan myös direktiivin skoopin muuttujat määritellä eri tavoin. Merkkejä on kolmenlaista:

- =-merkki. Toimii kahdensuuntaisesti, eli muuttujan arvon muutos yläskoopissa ja direktiivin skoopissa vaikuttavat kumpaankin (kaksisuuntainen tiedon sidonta).
- @-merkki. Saa sisäänsä arvon ylhäältä päin, mutta sen muuttaminen ei vaikuta yläskooppiin (yksisuuntainensuuntainen tiedon sidonta). @-merkkiä käytettäessä direktiivi saa arvon aina string -tyyppisenä.
- &-merkki. Käytetään määritelmien (expression) kohdalla. &-merkin avulla voidaan esimerkiksi kutsua yläskoopin funktiota, annettaessa sille arvoksi yläskoopin funktio. (Lerner 2013, 73–107.)

Angular tuo mukanaan monia valmiita direktiivejä. Joitain direktiivejä voi käyttää jopa huomaamatta, sillä Angular laajentaa automaattisesti esimerkiksi HTML form-elementin ominaisuuksia, kuten validaatiota, joita ei elementistä normaalisti löytyisi. Angularin valmiit direktiivit on helppo erottaa, sillä ne määritellään aina ”ng” prefiksillä. Näitä direktiivejä ovat muun muassa ng-if, joka poistaa elementin sisälle määritellyt elementit, mikäli sille annettu arvo on epätosi sekä ng-repeat, joka iteroi sille annetun kokoelman DOM-puuhun. (Lerner 2013, 73–107; Smith 2014.)

2.3.7 Palvelut (services)

Palvelut ovat hyödyllisiä, kun halutaan tehdä funktioita ja operaatioita, joita voidaan käyttää ohjelmassa moneen kertaan. Ne ovat ainokaisia (singleton) eli ohjelmassa ei ole koskaan niistä yli yhtä instanssia (AngularJS 2015e). Tyypillisiä palvelu esimerkkejä ovat vaikkapa kirjautumisen ja Ajax-kutsujen hallinnointi.

Palvelut eivät kuulu MVC-mallissa mihinkään osaan. Freemain mukaan ne eivät kuulu malliin, ellei ohjelman bisneslogiikkaan kuulu kirjautumisen ja verkon hallinnointi. Niitä ei myöskään voi jakaa ohjaimiin, sillä ne eivät reagoi käyttäjän interaktioon, eivätkä muokkaa mallia. Palvelut eivät myöskään kuulu näkymään, sillä ne eivät näytä mallin tietoja käyttäjälle. (Freeman 2014, 473-474.)

Palvelutyyppejä ovat muun muassa service, factory ja provider. Service ja factory -palvelut sekoittuvat usein keskenään, sillä ne ovat hyvin samankaltaisia. Palveluiden käytännön erona on se, että factory-palveluun voidaan tehdä funktioita, joista voidaan luoda uusia objekteja new-avainsanalla. Servicessä tätä ei voi tehdä. (Meyer 2014). AngularJS tiimin mukaan, service-palvelua kannattaa käyttää silloin, kun luodaan mukautettuja objektin tyyppejä ja factory palvelua silloin, kun halutaan tuottaa JavaScriptin alkeistyyppisiä

muuttujia ja funktioita (AngularJS 2015f). Angularin lähdekoodissa service luodaan angularin injektorilla (injector), joka tekee palvelusta uuden instanssin new -avainsanan avulla.

```
10  angular.module("customServices", [])
11      .factory("multiplyFactory", function() {
12          return {
13              multiply: function(a, b){
14                  return a*b;
15              }
16          }
17      })
18      .service("multiplyService", function() {
19          this.multiply = function(a, b){
20              return a*b;
21          }
22      });
```

Kuva 18 Factory- ja service-palvelu customServices moduuliissa

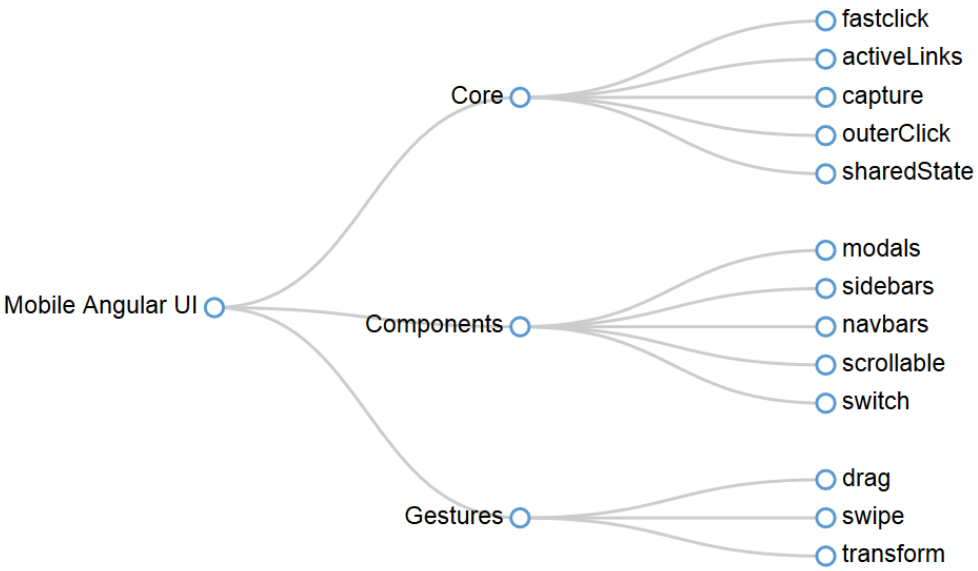
Kuvan 18 esimerkkikoodin palautus ei eroa tulokseltaan. Rivillä 11 luodaan factory-palvelu nimeltään multiplyFactory, joka palauttaa objektin, jolla on kiinni multiply-funktio. Tämä funktio palauttaa kahden muuttujan tulon. Service-palvelussa tämä tapahtuu kiinnittämällä palveluun multiply metodi this-avainsanalla (rivi 19). Service-palvelun konstruktori luodaan ohjelman esittelyvaiheessa ja factory:n palauttama objekti lähetetään aina kutsujalle injektoinnin aikana, mutta siitä on aina yksi instanssi olemassa. (Haggard 24.5.2013; Meyer 2014).

Provider-palvelu eroaa muista palvelutyypeistä sillä lailla, että sitä on mahdollista konfiguroida config-metodilla ennen sen käyttöä. Tätä ominaisuutta saatetaan tarvita esimerkiksi silloin, kun palvelua halutaan käyttää muissa sovelluksissa erilaisilla alkuarvoilla. (Tyler 4.5.2014.)

2.3.8 Mobile Angular UI

Mobile Angular UI on Maurizio Casin kehittämä käyttöliittymäkehys, joka on tarkoitettu ensisijaisesti mobiilikäyttöön. Mobile Angular UI:lla on Github palvelussa kirjoitushetkellä noin 1600 tähtimerkintää ja noin 300 forkkausta. Sen ulkoasu vastaa Bootstrapin tyyliä ja se yhdistelee kirjastoja, kuten Fastclick:ia, joka tekee mobiililla klikkauksista välittömiä. (Mobile Angular UI 2015a.)

Mobile Angular UI tuo mukanaan monenlaisia käyttöliittymäkomponenttia, kuten raahattavan sivupalkin, absoluuttisesti sijaitsevat navigaatiot, listanäkymän ja raahattavat alueet. Komponentit käyttävät Bootstrapin tyylejä, mutta Mobile Angular UI eroaa myös Bootstrapista; kehys ei tarvitse jQuerya vaan se korvaa jQuerya tarvitsevat elementit direktiiveillä. (Mobile Angular UI 2015a; Mobile Angular UI 2015b.)

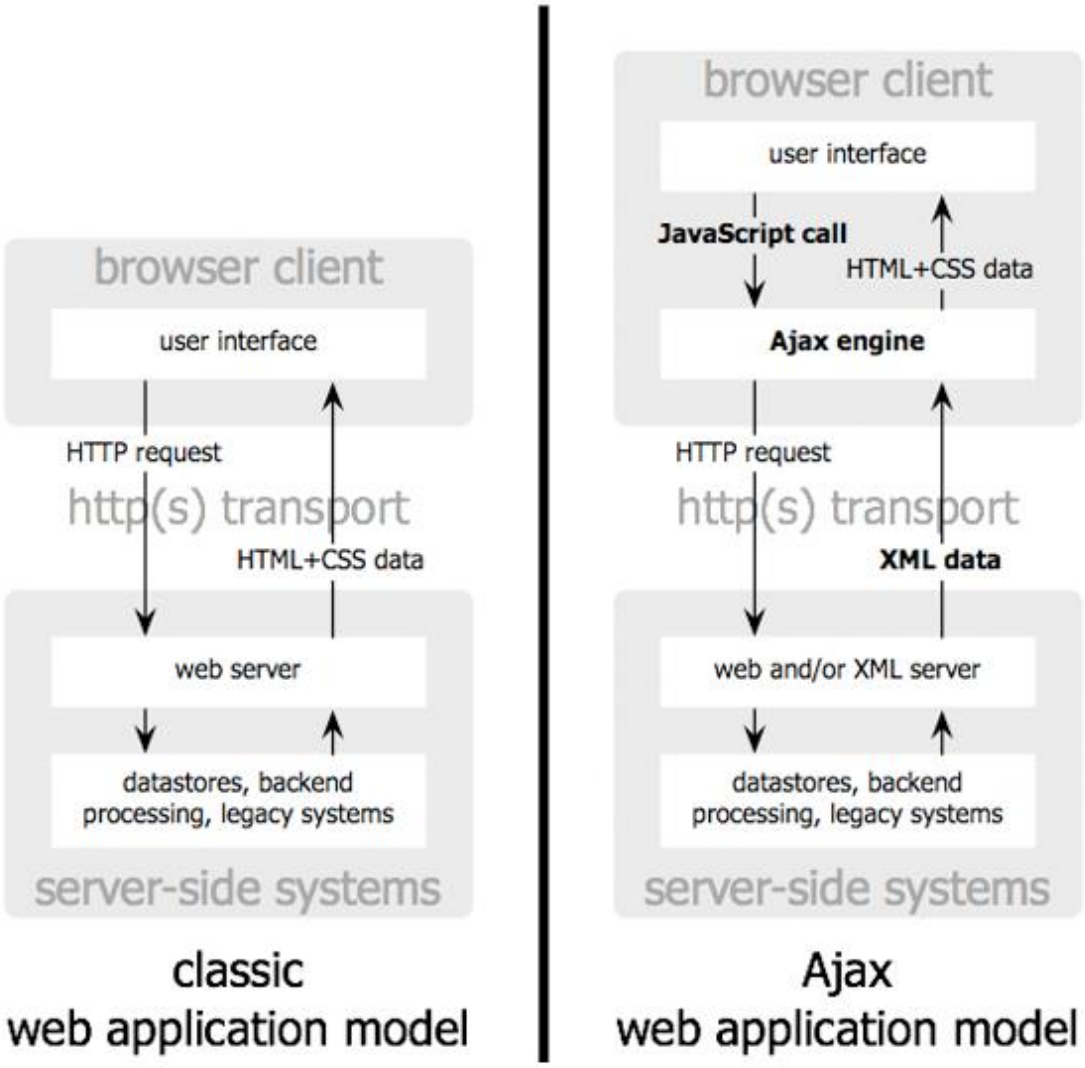


Kuva 19 Mobile Angular UI:n komponentit ja ominaisuudet (Mobile Angular UI 2015b)

Kuvassa 19 kuvataan Mobile Angular UI:n ominaisuuksia ja komponentteja. Mobile UI:n pääkomponentit (core) sisältävät kehyksen päätoimminnot eivätkä ne ole yhteydessä käyttöliittymän tyyleihin. Tällöin niitä on mahdollista käyttää jonkin toisen käyttöliittymäkehyksen kanssa. Komponentit (components) haarasta löytyvät käyttöliittymäkehyksen käyttöliittymä-komponentit. Ne tarvitsevat toimiakseen pääkomponentit ja ovat riippuvaisia kehyksen tyylitiedostoista. Mobile UI:n elehaarasta (gestures) taas löytyvät direktiivit esimerkiksi raahaukselle. Näiden komponenttien avulla on mahdollista tunnistaa esimerkiksi sormen näpäytykset ja raahaukset. (Mobile Angular UI 2015b.)

2.4 Ajax

Ajax on kokoelma tekniikoita, joiden avulla HTML-sivu on mahdollista päivittää osittain ilman koko sivun uudelleenlatausta. Sivun päivittyminen tapahtuu asynkronisesti, joka tarkoittaa, ettei muun ohjelmakoodin tarvitse odottaa asynkronisen kutsun päättymistä. Synkroninen palvelinkutsun tai ohjelmakoodin suorittaminen pysäyttää koko muun ohjelman etenemisen (Techopedia).



Kuva 20 Perinteinen malli verkkosovelluksissa (oikealla) verrattuna Ajax malliin (vasemmalla) (Garrett 2005)

Kuvassa 20 esitetään perinteisen verkkosovelluksen ja Ajaxia käyttävän verkkosovelluksen eroavaisuuksia. Ajax tekniikkaa käyttävä sovellus ei aiheuta sivun uudelleenlatausta jokaisen HTTP-pyynnön takia niin sanotun Ajax moottorin takia. Ajax moottori tekee pyynnöt palvelimelle yleensä JSON tai XML muodossa ja se palautuu samassa muodossa. (Garett 2005; W3C 2014b).

Angularissa asynkronisten palvelinkutsujen tekemiseen tarjotaan valmista palvelua. Angularin HTTP-palvelun avulla Ajax-kutsuja on helppo konfiguroida sekä palvelut abstraktoivat Ajax-kutsun käytön helpommaksi. Palvelu käyttää tekniikassaan JavaScript lupausobjekteja (promises). Lupaus-objektit helpottavat asynkronisten metodien tekoa poistamalle tarpeen lukuisille takaisinkutsu-metodeille (callback). Lupausobjektien tilasta on mahdollista nähdä kutsun tai funktion edistyminen. (AngularJS 2015g; Webb 4.5.2013).

3 Sovelluksen suunnittelu

Tässä luvussa käsitellään Hierojankortti -nimistä asiakkuudenhallintapalvelua. Luku keskittyy erityisesti projektin määrittelyyn, tavoitteisiin ja aiheen rajaamiseen sekä teknologiavalintojen perustelemiseen. Projektin tavoitteissa asetetaan opinnäytetyön tavoitteeksi tuottaa hierojille suunnattu asiakkuudenhallintasovellus. Sovelluksen teknologiavalinta-luvussa esitellään ja perustellaan sovelluksen teknologiavalintoja. Määrittelyosuudessa sovellusta tarkastellaan ensin valmiiden ratkaisujen ja hierojien tarpeiden pohjalta. Tämän jälkeen sovellus määritellään yleisellä tasolla ja lopulta Hierojankortin ominaisuudet määritellään.

3.1 Projektin tavoitteet

Projektin tavoitteena on viimeistellä kaupalliseen käyttöön tuleva hierojien asiakkuudenhallintaan soveltuva palvelu, Hierojankortti. Hierojankortin tuotantoversion kehitettäviin ominaisuuksiin kuuluvat potilastietojen ja hoitotapahtumien hallinta sekä niiden raportointi. Opinnäytetyön tavoitteena on kehittää sovellus määrittelyvaiheen ominaisuuksilla julkaistavavaksi produktiksi.

3.2 Sovelluksen ja opinnäytetyön rajaus

Opinnäytetyössä ei pureuduta Hierojankortin taustajärjestelmän toteutukseen toimeksiantajan pyynnöstä, vaan keskitytään sovelluksen asiakaspään toteutukseen. Taustajärjestelmän komponentteihin ja tekniseen toteutukseen saatetaan viitata opinnäytetyössä, mutta siihen ei syvennytä sen tarkemmin. Myöskään sovelluksen käyttämää tietokantaa ja autentikoinnin tekniikkaa ei toimeksiantajan pyynnöstä tarkastella kuin yleisesti.

Sovelluksen ominaisuuksiin rajoitetaan määrittelyvaiheessa kuvatut ominaisuudet. Määrittelyvaiheessa kuvaamattomat mahdolliset kehitysaskeleet eivät kuulu sovelluksen opinnäytetyössä esiteltäviin tietoihin. Sovelluksen julkaisua ei esitellä opinnäytetyössä.

3.3 Määrittely

Suomessa koulutetut hierojat kuuluvat terveydenhuoltolain piiriin, sillä heidän katsotaan olevan terveydenhuollon ammattilaisia. Terveydenhuollon ammattilaisen on lain mukaan pidettävä asiakasrekisteriä asiakkaistaan hoidon ja perustietojen osalta. Asiakasrekisterin ylläpitämisellä on tarkoitus turvata asiakkaalle mahdollisimman hyvä ja turvallinen hoito sekä toimiva asiakassuhde. Asiakasrekisteriin merkitään perustietoina esimerkiksi hoidettavan henkilötiedot, osoitetiedot. Hoitotapahtumista on pidettävä kirjaa ja rekisteristä on

löydyttävä asiakkaalle annetut hoidot ja niiden kestot. (Laki terveydenhuollon ammattihenkilöistä 28.6.1994/559.)

Hierojan ammattia harjoittavalle on tarjolla paperisia potilaskortteja, joiden avulla asiakasrekisteriä on mahdollista hallita paperisessa muodossa. Näitä potilaskortteja tarjoaa esimerkiksi koulutettujen hierojien liitto tai ammatinharjoittajalla voi olla oma muunnelma kyseisestä asiakaskortista (Tiainen 5.10.2014).

Projekti määrittyi tarkastellessa hierojien tarpeita ja markkinoilta löytyviä ratkaisuja asiakashallintaan. Hierojan työssä asiakasrekisterin ylläpito pitäisi helppoa ja nopeaa, sillä hierojilla ei välttämättä ole aikaa merkitä niitä hierontojen välillä. Merkintöjen tekeminen voi olla hankalaa, jos asiakasrekisteriä ei voida käyttää monessa laitteessa vaan esimerkiksi tietoihin päästään käsiksi vain työpöytäkoneelta. Merkintöjen tekemistä voidaan helpottaa monelta laitteella toimivalla palvelulla, sillä asiakasmerkinnät voidaan tehdä suoraan lähettyvillä olevalla laitteella**.** Tilanne on huono silloin kun koko asiakashallintaa ylläpidetään paperisessa muodossa. Paperisessa muodossa merkinnät ovat useammin virheellisiä ja niitä on vaikeampi korjata kuin sähköisessä muodossa. Paperin fyysinen olemus sitoo käyttäjän yhteen paikkaan, samoin käy jos sähköinen asiakkuudenhallintajärjestelmän käyttö on rajoitettu pöytätietokoneeseen.

Markkinoilta löytyy ratkaisu, Diarium, joka ottaa huomioon myös asiakashallinnan mobiilille (Diarium 2015). Sen palvelu toimii niin työpöytä- kuin mobiilikäytössäkin. Sen käyttökokemus mobiilille ei mielestäni ole vielä sillä tasolla, esimerkiksi vahvasti työpöytäkäyttöön tarkoitetun käyttöliittymän takia, että sitä olisi mielekästä käyttää.

Hierojien tarpeiden ja valmiiden ratkaisuiden pohjalta voidaan tilanne kiteyttää seuraaviin johtopäätöksiin:

- Asiakasrekisterin ylläpito ei saisi viedä liikaa aikaa.
- Asiakasrekisteriin olisi hyvä päästä käsiksi tilasta ja laitteesta riippumatta.
- Hierojien asiakashallintaan ei kirjoitushetkellä löydy käyttökokemukseltaan hyvää mobiiliratkaisua.

Hierojankortti haluaa vastata markkinoiden nykyiseen tarpeeseen. Tilannetta arvioimalla tulimme toimeksiantajan kanssa seuraaviin määrittelyihin sovelluksesta yleisellä tasolla:

- Sovelluksen on toimittava sekä työpöytä- että mobiilikäytössä, jolloin minimoidaan asiakashallintaan kuluva aika.
- Sovelluksen on käytettävä ulkoista tietokantapalvelinta tietojen hallitsemiseen, jolloin sovellusta voi käyttää laitteesta riippumatta.

- Sovellus toteutetaan palveluna (Software as a Service), jolloin ohjelmaa tarvitse asentaa.
- Sovellus toteutetaan käyttäen mobiililähtöistä suunnittelua (mobile first), jolloin pyritään takaamaan positiivinen käyttökokemus älypuhelimilla ja tableteilla.

Sovellus määriteltynä yleisellä tasolla vastaa kysymykseen miten sovellus pitäisi toimia. Sovellukseen ei ole kuitenkaan vielä määritelty mitä sovelluksella voi tehdä. Hierojankorttilla halutaan pääasiassa tarjota hierojille tapa hallita asiakasrekisteriään sähköisesti. Toimeksiantajan mukaan sovelluksen tulee sisältää ainakin nämä ominaisuudet:

- Sovelluksen avulla hieroja voi tallettaa, muokata ja poistaa asiakastietoja.
- Hieroja voi tallettaa, muokata ja poistaa asiakkaille tehtyjä hoitotapahtumia.
- Käyttäjä voi muuttaa omia käyttäjätietojaan ja asetuksia.
- Sovellukseen on mahdollista luoda hoitotapahtumapohjia, joilla helpotetaan samankaltaisten hoitojen merkintää.
- Sovelluksella on mahdollista tarkastella omia tapahtumia ja asiakkaitaan raporttimallina.

3.4 Sovelluksen teknologiavalinnat

Sovellusta alettiin suunnittelemaan toimeksiantajan ja omien tietotaitojen ja kokeiltujen teknologioiden pohjalta. Halusimme yhdistellä vanhaa tietämystämme, mutta kokeilla myös uusia teknologioita tietotaitomme kasvattamiseksi. Tiukan aikataulun puitteissa teknologiavalinnat oli tehtävä melko ripeästi. Sovellusta toteutettiin osana HAAGA-HELIAn kurssia ja aikataulu määräytyi kurssista. Kurssin aikana sovellus oli saatava esiteltävään kuntoon.

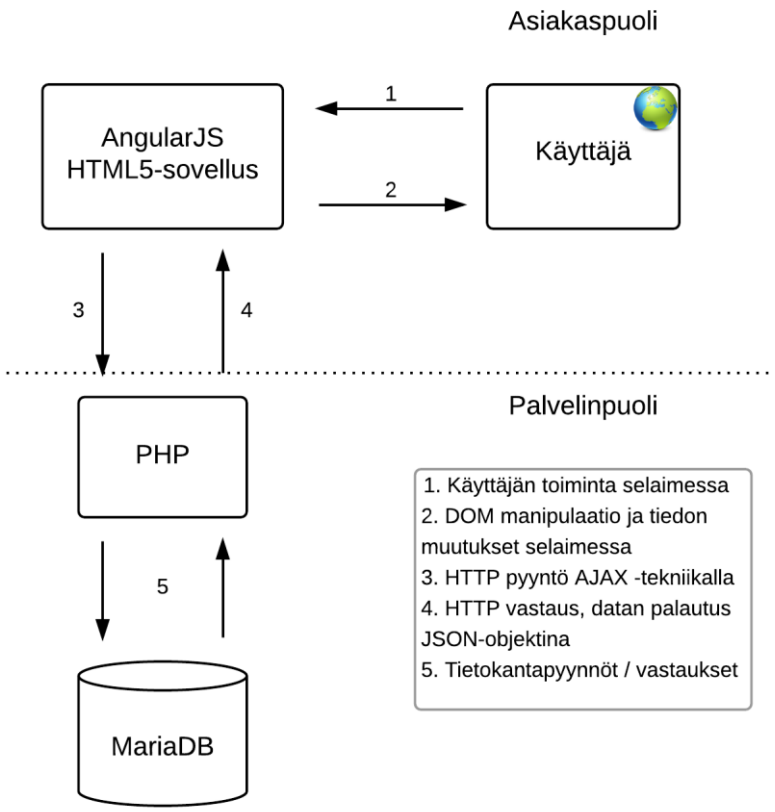
Sovelluksen toteutustavaksi valittiin HTML5-sovellus sen kehittämisnopeuden takia. HTML5-sovellus tai verkkosovellus voidaan toteuttaa kaikille alustoille samaan aikaan käyttäen hyväksi responsiivista suunnittelua. Natiivin sovelluksen kehitys ei tätä tarjonnut, joten natiivia sovellusta ei alettu rakentamaan vaikkakin natiivin sovelluksen suorituskyky olisi ollut sovelluksen eduksi.

Tietokantaratkaisuksemme päätyi MariaDB, joka on MySQL:lään pohjautuva avoimen lähdekoodin relaatiokantajärjestelmä (MariaDB 2015). Tämä vaihtoehto tuli miltei heti mieleen, sillä siitä oli aikaisempaa kokemusta. Aiempi kokemus teknologiasta riitti perusteluksi valinnasta.

Toinen palvelinpuolen teknologia, joka vastaanottaa tiedot relaatiokannalta ja tekee lopul-

liset kyselyt, päätyi olemaan PHP. Käytännössä ratkaisussa ei ollut juurikaan erimielisyyksiä, vaan se koettiin tutuksi ja toimivaksi ratkaisuksi tällaisessa projektissa. Miltei jokainen pitkälle kehitetty palvelinpään teknologia osaa palauttaa ja kääntää JSON-objekteja, eikä PHP ole tässä poikkeus.

Syy miksi emme lähteneet niin sanotusti ”full-stack” JavaScript ratkaisuun, jossa myös palvelinpuoli toteutuu JavaScriptilla, oli se, ettei meillä ollut tarpeeksi kokemusta näistä ratkaisuista (Node.js, MongDB, ExpressJS) ja tällöin teknologian opetteluun saattaisi kulua turhaa aikaa.



Kuva 21 Sovelluksen yleinen arkkitehtuuri

Kuva 21 esittää sovelluksen rakennetta. Asiakaspäässä käyttäjän toiminnot vaikuttavat AngularJS-sovelluskehykseen, jossa näkymä suorittaa DOM manipulaatiota ja tekee HTTP-pyyntöjä taustajärjestelmälle (1 - 3). HTTP-pyynnön mukaan palvelinään PHP tekee tietokantahakuja (5). Tietokannasta palautuneet tiedot lähetetään HTTP-vastauksena JSON -formaattina (4). JSON-objektit näytetään sovelluskehyksen kautta käyttäjälle.

Asiakaspuolen teknologioissa meillä ei ollut hirveästi kokemusta eri JavaScript sovelluskehyksistä. Olimme tehneet enimmäkseen muita projekteja puhtaalla JavaScriptilla ja

jQuery-kirjastolla. Olimme kuulleet paljon AngularJS-kehyksestä ja muutaman artikkelin ja opetusvideon kautta saimme käsityksen, että sovelluskehys sopisi hyvin projektiimme. AngularJS:n soveltuminen CRUD-sovelluksiin ja sen jatkuva kehitys, sekä iso kehittäjä sen taustalla saivat meidät päätymään siihen. Olimme tietoisia sen tulevasta versiosta, joka julkaistaisiin tulevaisuudessa projektin jälkeen. Uudessa versiossa AngularJS tulisi muuttumaan melko paljon, mutta luotimme kykyymme sopeutumaan tähän muutokseen.

Käyttöliitymä-sovelluskehyksissä tarkastelimme ensin Ionic:ia, joka on yksi suositummista käyttöliittymä-sovelluskehyksistä hybridisovelluksille (Ionic 2015). Ionic ei kuitenkaan päätöshetkellä tukenut Windows-puhelinta eikä työpöytäkäyttöliittymää, joten sitä ei otettu käyttöön. Emme myöskään halunneet käyttää jQuery mobiilikäyttöliittymää, sillä se soveltuu huonosti yhteen AngularinJS:n kanssa ja omakohtaisen kokemuksen mukaan käyttöliittymä oli raskas. Päädyimme lopulta Mobile Angular UI:hin, joka oli mielestämme tyylikäs ja toimiva. Angular Mobile UI oli toteutettu Angularilla, joten takasi käyttöliittymäkehyksen yhteensopivuuden Angularin kanssa sekä käyttöliittymä soveltui myös työpöytäkäyttöön.

4 Sovelluksen toteutus

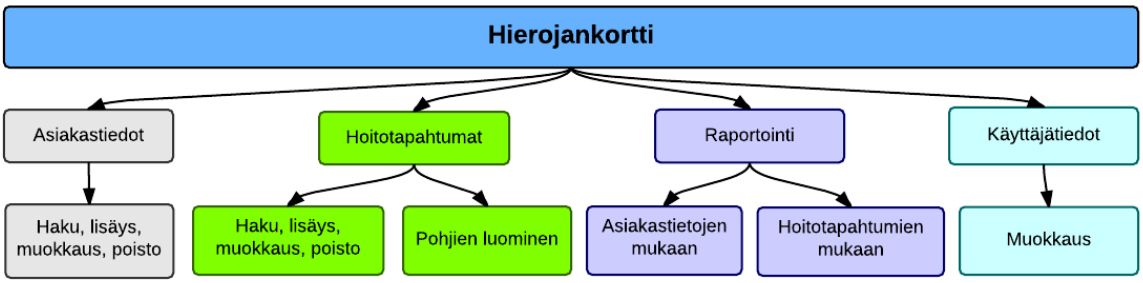
Tässä luvussa kuvataan toteutetun sovelluksen asiakaspään ratkaisuja tekniseltä ja visuaaliselta näkökannalta. Toteutusta käsitellään sovelluksen ominaisuuksien näkökulmasta.

Sovelluksen yleiskuvauksessa käydään läpi sovelluksen ominaisuuden tietokannasta löytyvien tietojen näkökulmasta. Sovelluksen tiedostorakenne -luvussa käsitellään sovellusta tiedostojen jaottelun mukaan. Tiedostorakenteen jälkeen esitellään sovellusta teknisesti, jonka jälkeen käydään läpi Hierojankortin käyttöliittymää ja sivuja.

4.1 Sovelluksen yleiskuvaus

Projektissa toteutettiin HTML5-sovellus joka nimettiin Hierojankortiksi. Hierojankortin avulla hieronta-alalla toimiva henkilö voi pitää kirjaa omista asiakkaistaan sekä hieronnan yksityiskohdista.

Hierojankortti koostuu hierojan asiakkaiden hallinnasta, asiakkaille hoitotapahtumien hallinnasta, raportoinnista sekä käyttäjän käyttäjätietojen hallinnasta. Asiakastietoihin käyttäjä voi merkitä asiakkaan perustiedot, sairaustiedot sekä kuvata hoitosuunnitelman ja toteutuneen hoidon. Hoitotapahtumiin merkitään yksittäisen käynnin tiedot. Näitä ovat esimerkiksi hinta ja kesto. Käyttäjätiedoista käyttäjä voi vaihtaa omia tietojaan sekä salasanaansa. Raportoinnin avulla käyttäjä pystyy tarkastelemaan lisäämiään asiakkaita ja hoitotapahtumia raporttinäkymästä. Raportin voi järjestellä ja luoda eri tietoja painottaen. Raporttia voidaan tarkastella esimerkiksi luotujen asiakkaiden määrän mukaan tietyllä aikavälillä. Kuva 22 jäsentää sovelluksen ominaisuudet kokonaisuuksittain.



Kuva 22 Hierojankortin ominaisuudet

Hierojankortin ominaisuudet määräytyivät pitkälti taustalla olevan tietokannan mukaan. Tietokannasta saatavat tietomallit rakentuvat yksinkertaistettuna käyttäjästä ja käyttäjän asiakkaista ja hoitotapahtumista. Käyttäjätiedot koostuvat henkilön omista tiedoista ja käyttäjän yrityksen tiedoista. Käyttäjä on tässä tilanteessa palvelun käyttäjä eli oletetta-

vasti hieronta-alan yrittäjä ja hänen asiakkaansa ovat hierottavia ja hoitotapahtumat ovat hierontakäyntejä.

Tieto (Malli)	Sovelluksen ominaisuus / kokonaisuus	Lisäselitys
Käyttäjätieto	Autentikointi	Käyttäjä voidaan tunnistaa kirjautumisen yhteydessä annetulla salasanalla, jota verrataan käyttäjätiedon salasanaan
	Omien tietojen muokkaus	Käyttäjällä on mahdollisuus muokata omia tietojaan ja vaihtaa salasanaansa sovelluksessa
Asiakastieto (potilas)	Asiakastietojen hallinta	Käyttäjä voi hakea, tarkastella, lisätä, muokata ja poistaa omia asiakastietojaan.
	Raportointi	Käyttäjä voi hakea asiakkaitaan ja analysoida tietoa
Hoitotapahtuma	Hoitotapahtumien hallinta	Käyttäjä voi hakea, tarkastella, luoda ja muokata asiakkaiden hoitotapahtuma
	Raportointi	Käyttäjä voi hakea hoitotapahtumia ja analysoida tietoa

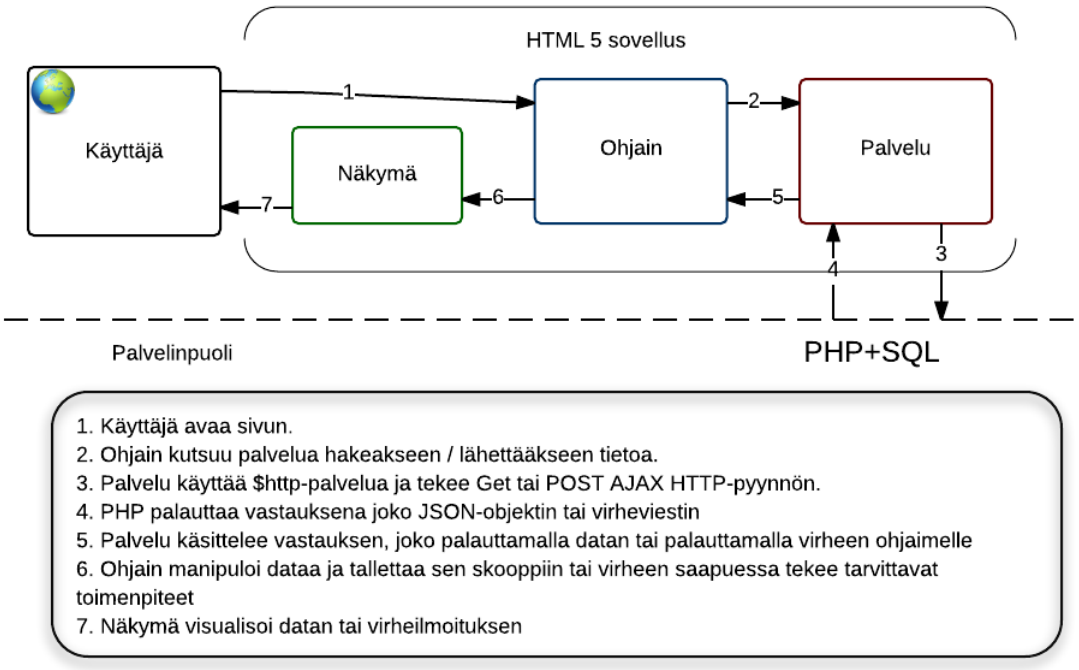
Taulukko 1 Hierojankortti-sovelluksen ominaisuudet tietokannasta löytyvien tietojen näkökulmasta

Taulukossa 1 esitetään sovelluksesta löytyviä ominaisuuksia ja kokonaisuuksia sovelluksessa käsiteltävien tietojen mukaan. Käytännössä kaikki sovelluksen osat tarvitsevat käyttäjätietoja, sillä jo taustajärjestelmään yhteydenotto tarvitsee tarkistuksen käyttäjästä. Sovelluksessa kokonaisuudet voidaan jakaa tiedon välittömiin ja välillisiin kokonaisuuksiin. Tietoa välillisesti käyttävät kokonaisuudet ovat esimerkiksi raportointi ja autentikointi, sillä ne tarvitsevat tietoa välillisesti toimiakseen, mutta eivät tee itse tiedolle mitään, kun taas

esimerkiksi asiakastiedon hallinta koostuu välittömästi tiedosta ja sen muutokset vaikuttavat suoraan tietoon.

Sovelluksen arkkitehtuuri soveltaa teoriaosuudessa käsiteltyä Angularin MVC-mallia. Yleisimmin sovellus toimii niin että mallit saadaan tietokannasta taustaustajärjestelmän kautta JSON-objekteina. Mallit liitetään skooppiin ohjaimessa ja ohjain on vastuussa mallin muuttamisesta. Käyttäjän toiminnot muokkaavat mallia ohjaimen kautta. Näkymässä tehdään skoopin tiedolle tyylimuutoksia, joka vaikuttaa siihen kuinka käyttäjä näkee skoopin tiedon. Näkymien ja ohjainten suhde sovelluksessa on lähes yksi yhteen. Tämä tarkoittaa, että ohjelma perustuu suurimmalta osalta monoliittisten ohjainten käyttöön. Jokaisella näkymällä on siis useimmiten yksi ohjain eikä ohjainten periyttämistä juurikaan käytetä.

Sovelluksessa pyrittiin käyttämään tiedon hakemiseen ja lähettämiseen palveluita, jotka käyttävät Ajax-kutsuille tarkoitettua Angularin omaa HTTP-palvelua. HTTP-pyyntöjen tekeminen ylemmällä tasolla omissa palveluissa eikä ohjaimissa, vähensi ohjaimissa tarvitun koodin määrää ja hakuja voitiin tehdä ympäri sovelluksen ohjelmakoodia. Tiedon liikkuminen sovelluksessa kuvataan kuvassa 23.

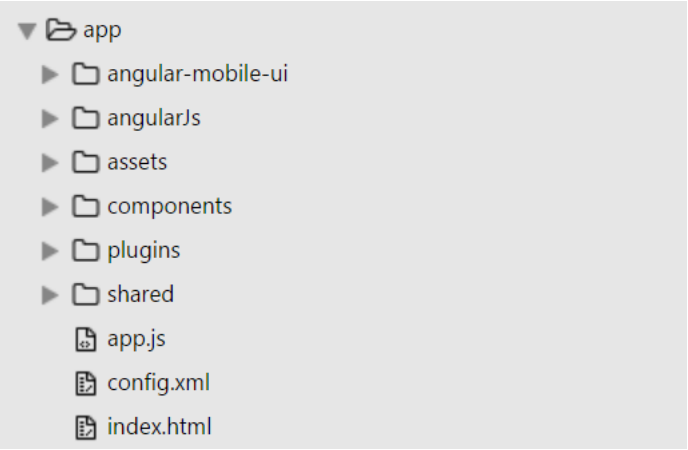


Kuva 23 Tiedon liikkuminen sovelluksessa

Sovellus on yksisivuinen sovellus, jonka mahdollistaa Angularin routeprovider-reitityspalvelu. Reititin pitää huolen siitä, että sivun URL:in vaihtuessa, vaihtuu myös näkymä.

4.2 Sovelluksen tiedostorakenne

Sovelluksen kansiorakenteessa sovellettiin Angularin käytäntöjä Adnan Kukicin kirjoittamien parhaiden käytäntöjen mukaan. Hänen mielestään Angularin komponentit tulisi jakaa toiminnoittain, eikä niiden tyypin mukaan. Toiminnoittain jakaessa kansiot on helpommin hahmotettavissa tyyppijakoon verrattuna. Tyypin mukaan jakaessa tiedostot jaetaan esimerkiksi ohjaimet, näkymät, direktiivit ja palvelut -kansioihin. Lopulta kansiosta on hankala löytää etsimäänsä tiedostoa tiedostojen määrän kasvaessa. Toiminnoittain jakaessa kansiosta on selkeämmin erotettavassa haettava tiedosto, koska samaan toimintoon kuuluvat tiedostot ovat samassa kansiossa ja tiedostolla on näin selkeä konteksti. Tyyppijaossa konteksti on liian laaja. (Kukic 2014).



Kuva 24 Sovelluksen kansiorakenne

Kuva 24 esittää sovelluksen kansiorakennetta päätasolta. Päätasolta sovelluksen alta löytyvät kansiot assets, components, shared ja plugins sekä kansio AngularJS-sovelluskehykselle ja Mobile Angular UI:lle. Components -kansiosta löytyvät omat Angular-tiedostot kuten ohjaimien koodi sekä ohjaimien näkymät. Assets kansiossa sijaitsevat muokatut CSS tiedostot, kuvat sekä Angulariin kuulumattomat JavaScript kirjastot Shared-kansio kuuluu Angular -tiedostoille, joita käytetään useassa paikassa eikä niitä voi kategorisoida mihinkään komponenttiin kuuluvaksi. Shared kansio pitää sisällään esimerkiksi Ajax-kutsuja hallinnoivat palvelut. Plugins kansio liittyy sovelluksen Phonegap mobiilisovellus -versioon, jossa on mobiilisovellukseen vaikuttavia lisäosia. Phonegapin osat eivät liity verkkoselaimelta käytettävään verkkosovellukseen.

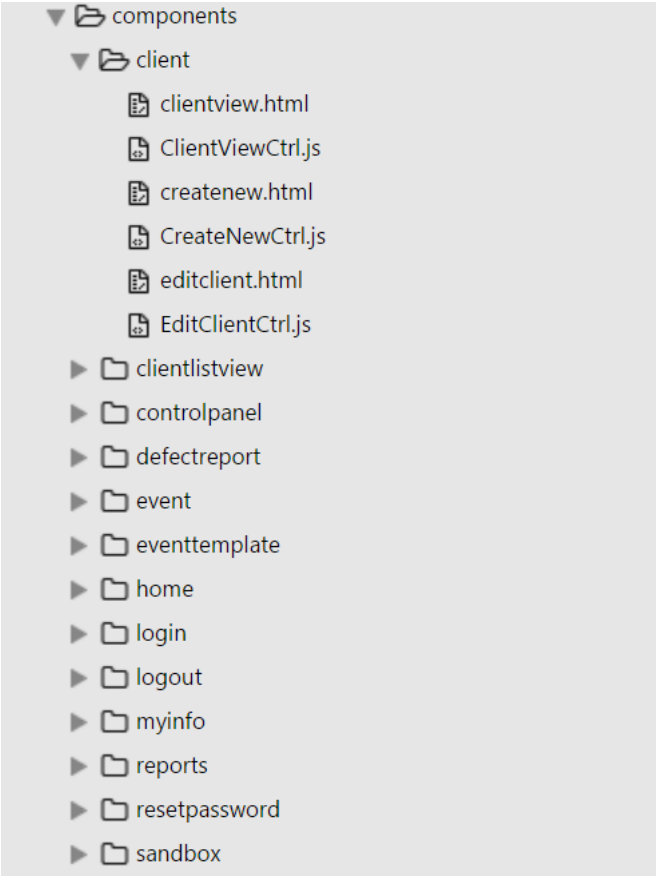
Kansioiden lisäksi päätasolta löytyy index.html, app.js ja config.xml. App.js -tiedosto sisältää sovelluksen keskeisen ohjelmakoodin sovelluksen alustukselle. Siellä määritellään päämoduuli ja sen riippuvuudet sekä reititys. Index.html tiedosto toimii sovelluksen

päänäkymänä ja sinne on määritelty sivun raamit kuten sivupalkki ja yläpalkit. Tiedostoon määritellään myös kaikki JavaScript kirjastoriippuvuudet. Config.xml tiedosto liittyy Phonegap mobiilisovellukseen ja sieltä löytyy määrittelyjä, kuten sovelluksen nimi, Phonegap versio ja käytettävät lisäosat.

```
44  angular.module('app', [
45      "ngRoute",
46      "ngTouch",
47      "ngCookies",
48      "mobile-angular-ui",
49      "xeditable"
50  ]);
51
52  angular.module('app')
53  .config(['$routeProvider',
54    function($routeProvider) {
55      $routeProvider.
56        when('/createNew', {
57          templateUrl: 'components/client/createnew.html',
58          controller: 'CreateNewCtrl'
59        }).
60        when('/AddEvent', {
61          templateUrl: 'components/event/addevent.html',
62          controller: 'AddEventCtrl'
63        }).
64         when('/editEvent', {
65          templateUrl: 'components/event/editevent.html'
66        }).
67        when('/clientListView', {
68          templateUrl: 'components/clientlistview/clientlistview.html',
69          controller: 'ClientListViewCtrl'
70        }).
71        when('/clientView', {
72          templateUrl: 'components/client/clientview.html',
73          controller: 'ClientViewCtrl'
74        }).
```

Kuva 24 Riippuvuuksien määrittely päämoduulissa sekä reititys

Kuvassa 24 syvennytään app.js -tiedoston moduulin määrittelyyn ja reititykseen. Rivillä 44 määritellään moduuli nimeltä app, jolle annetaan riippuvuuksiksi muita moduuleita, kuten ngRoute ja ngTouch. Riippuvuudet ovat päämoduulista erillisiä moduuleita, joita injektoinnin avulla voidaan käyttää app-moduulissa. App-moduuliin injektoitu ngRoute on Angular-tiimin kehittämä reitin (routeprovider). Reititin vastaa näkymän vaihtamisesta sovelluksen sisällä. Rivillä 53 kutsutaan reitittimen config-metodia ja tämän jälkeen annetaan reitittimelle oletusreittejä, jonka avulla se lataa tietyn näkymän ja ohjaimen näkymälle. Reitin lukee arvoja selaimen osoitteesta ja vertaa niitä ohjelmoijan antamiin arvoihin. Mikäli osoite vastaa annettua vaihtoehtoa, ladataan sille määritellyt komponentit ja toiminnot ja näkymä vaihtuu.



Kuva 25 Components kansio ja client kansio avattuna

Kuvassa 25 esitetään components -kansion rakennetta tarkemmin. Toiminnoittain jakaessa, JavaScript-tiedostot ja näkymien HTML-tiedostot löytyvät samaan toimintokokonaisuuteen kategorisoidusta kansiosta. Kuvassa 25 client-kansio on avattu paremmin tarkasteltavaksi ja se sisältää potilaan katsomiseen, luomiseen ja muokkaamiseen liittyvät tiedostot. Client-kansio toimii siis ikään kuin pienenä sovelluksena itsessään.

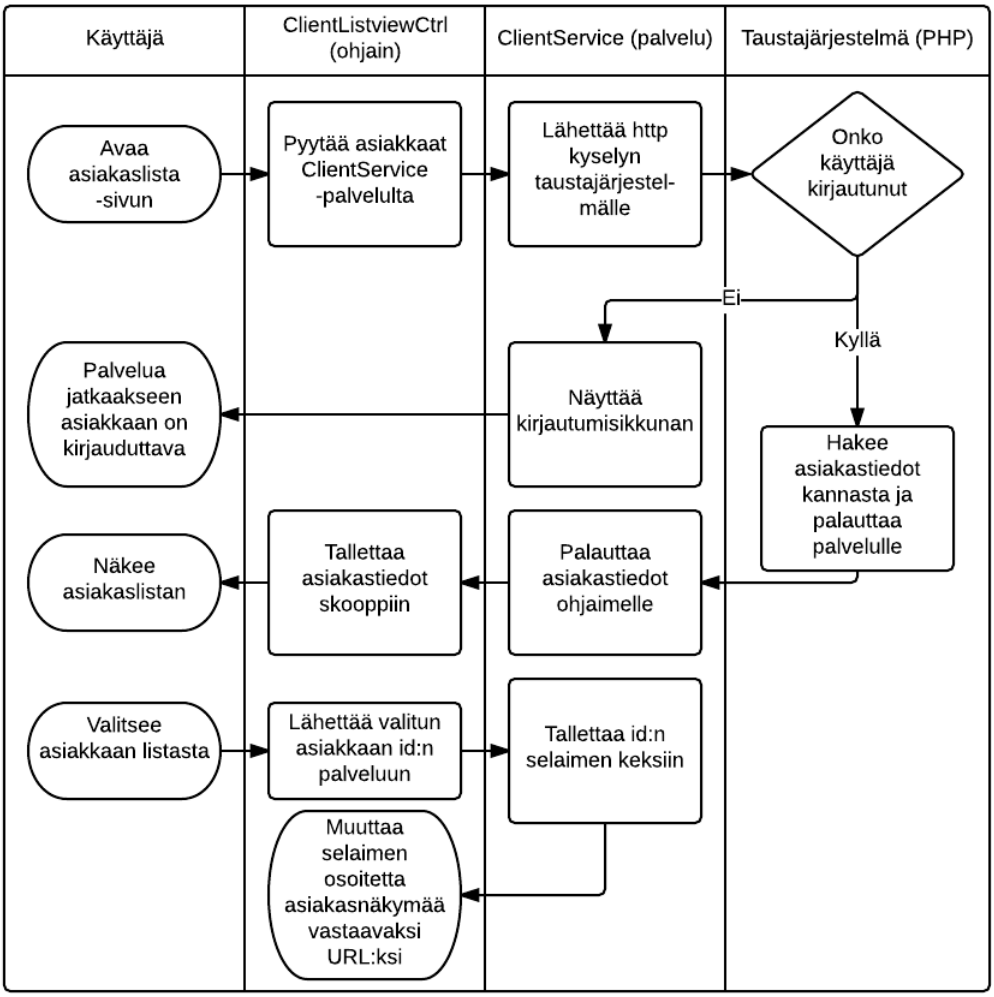
4.3 Asiakastietojen hallinta

Sovelluksen keskeisimpänä elementtinä on asiakaskantojen ylläpito. Asiakastietojen hallinta käsittää alleen CRUD-sovelluksen periaatteet, sillä asiakastietoja on mahdollista luoda, katsoa, muokata ja poistaa.

4.3.1 Asiakastietojen haku

Asiakastietojen katselemiseen toteutettiin kaksi näkymää. Ensimmäinen näkymä listaa käyttäjän kaikki asiakkaat mobiilioptimoituun listanäkymään. Listanäkymässä on mahdollista etsiä potilaita etu- ja sukunimen mukaan sekä osoitetiedoilla. Näkymän asiakaslistaa voi järjestää eri tavoilla. Tämä toteutuu käyttäen Angularin valmista suodatinta (filter). Suodatin muokkaa listaa sille antamien hakutermien perusteella.

Listanäkymästä on mahdollista päästä yksittäisen asiakkaan tarkastelunäkymään, mikä onnistuu painamalla kyseisistä asiakasta. Jokaisen näkymän tiedon latauksen yhteydessä tarkastetaan onko käyttäjä kirjautuneena.



Kuva 26 Asiakaslistan haku ja näyttäminen sekä yhden asiakkaan haku prosessikaaviona

Kuvassa 26 olevassa prosessikaaviossa esitetään asiakaslistauksen prosessia. Kuvasta on nähtävissä, että ohjaimella on roolinaan viestittää käyttäjän pyyntöjä palvelulle ja tallettaa palvelulta saatua tietoa skooppiin. Sovelluksen keskustelu PHP-taustajärjestelmän kanssa toteutettiin palveluina. Näin taattiin, että http-kutsuja voitiin käyttää monessa paikassa. PHP:n palauttamille JSON-objekteja manipuloitiin mahdollisesti jo palvelussa. Esimerkiksi haluttaessa asiakkaan syntymäpäivätietoja, jouduttiin aika muuttamaan aina suomalaiseen muotoon. Angularin MVC-mallin mukaan voidaan katsoa, että päivämäärän muuttaminen suomalaiseen muotoon kuuluu näkymälle sillä päivämäärän muodon muuttaminen on tiedon ”visualisointia”, eikä se vaikuta ohjelman logiikkaan. Päädyimme kuitenkin pitämään ajan muodon muuttamisen palvelussa, jolloin muotokonversiota ei tarvinnut kirjoittaa aina sivun HTML-pohjaan direktiivillä.

```
7           getClient: function(id){
8               var data = {id: id};
9               var url = ROOT_URL+'getCustomerById.php';
10
11              var deferred = $q.defer();
12
13              $http({withCredentials: true, method: 'GET', url: url, params: data}).
14                  success(function(data) {
15                      // if client has birthdate, format it to finnish format
16                      if (data.birthdate){
17                          data.birthdate = DateService.convertDate(data.birthdate);
18                      }
19
20                      // build conditions to readable list
21                      data.conditionList = ListBuilder.getConditionList(data);
22                      data.examinationList = ListBuilder.getExaminationList(data);
23
24                      deferred.resolve({data: data});
25                  }).
26                  error(function(data, status) {
27                      deferred.reject({ message: "Failed to load client data" });
28
29                      if (status === 401) {
30                          $rootScope.toggle('loginOverlay', 'on'); // user not logged in
31                          deferred.reject({ message: "Unauthorized" });
32                      }
33                  });
34
35              return deferred.promise;
36          },
37
38          setId: function(id) {
39              $cookies.customerid = id;
40          },
```

Kuva 27 ClientService-palvelun ohjelmakoodia

HTTP-pyyntöjä hallitsevat palvelut muodostettiin palauttamaan lupaus-objekteja (promise) viivästäjä-objektilla (deferred). Viivästäjä-objektin avulla voidaan saada asynkronisen metodin tilasta tietoa, sen tarjoaman lupaus-objektin avulla. Kuvassa 27 näkyvä ohjelmakoodi on palanen ClientService-palvelua, jossa esitellään yhden asiakkaan hakemista sen id:n perusteella, getClient-metodilla. Metodissa käytetään Angularin HTTP-palvelua asynkronisen HTTP-pyynnön tekemiseen. HTTP-palvelua voidaan konfiguroida ja rivillä 13 konfiguroidaan sen olevan GET-pyyntö. Params-määritelmä määrittelee pyynnön URL-parametrit. Koodissa success ja error -lohkot merkitsevät HTTP-pyynnön onnistumista ja epäonnistumista. Mikäli pyyntö onnistuu, tehdään palvelussa tiedon manipulaatiota: päivä muutetaan suomalaiseen muotoon ja sairauksista tehdään helpommin käsiteltävä lista ListBuilder-palvelun avulla. Lopuksi viivästäjä-objektin lupaus ”ratkaistaan” (resolve), ja siihen kiinnitetään palvelimelta tullut tieto. Error-lohkossa laitetaan lupaukseen kiinni epäonnistumisen syy ja lupauksen tila muuttuu epäonnistuneeksi reject-metodilla.

```
5       $scope.client = {};
6
7       $scope.getClient = function(){
8           ClientService.getCustomer(ClientService.getId()).then(
9               function(success){
10                  // talletetaan potilaaksi palvelulta saatu data
11                  $scope.client = success.data;
12              }, function(error){
13                  console.log(error.message);
14              });
15      };
16      $scope.getClient(); // kutsutaan ohjaimen käynnistyessä
```

Kuva 28 Yksinkertaistettu versio yhden asiakkaan näkymän ohjainkoodista

Kuvassa 28 haetaan yhden asiakkaan tiedot ohjaimessa käyttäen ClientService-palvelua. Palvelun palauttaa lupaus-objektin ja getClient-metodi jatkuu vasta lupauksen onnistumisen tai epäonnistumisen jälkeen. Ohjain on vastuussa asiakkaan tallettamisesta skooppiin (rivi 11) ja asiakashaun logiikka on siis pitkälti ClientService-palvelun varassa.

4.3.2 Potilaan lisäys ja muokkaus

Potilaan lisäämisessä ja muokkaamisessa toimivat samat periaatteet kuin potilaan hakemisessa. Ohjain on vastuussa tiedon tallettamisesta ja hakemisesta skooppiin ja tieto haetaan ja lähetetään palvelun kautta. Asiakkaan luonti ja muokkaus ovat lähes identtiset: kummatkin käyttävät samaa HTML-pohjaa ja palvelua hyväkseen. Muokkauksen erona on se, että pohjaan ladataan valmiin potilaan tiedot, kun taas uuden potilaan luomisessa alustetaan tiedot tyhjiksi.

Ero asiakkaan haun suhteen lisäyksessä ja muokkaamisessa on se, että HTTP-pyyntö tehdään POST-tyyppisenä, jolloin lisättävän potilaan tiedot kulkevat JSON-objektina palvelimelle.

```
7   createClient: function(client){
8
9       var url = ROOT_URL+'NewCustomer.php';
10      var deferred = $q.defer();
11
12      $http({withCredentials: true, method: 'POST', url: url, data: client}).
13          success(function(data, status) {
14
15              deferred.resolve({data: data});
16          }).
17          error(function(data, status) {
18
19              if (status === 401) {
20                  $rootScope.toggle('loginOverlay', 'on'); // user not logged in. Show login overlay
21              }
22              deferred.reject({ status:status, message: "Asiakkaan tallennus ei onnistunut" });
23          });
```

Kuva 29 Asiakkaan lisäyksen toteutus ClientService-palvelussa

Kuvassa 29 esitellään ClientService-palvelun asiakaslisäyksen toteutusta. Rivillä 12 määritellään http-pyynnön olevan POST-tyyppinen sekä pyynnön runkoon talletetaan palveluun parametrina lähetetty asiakas (data: client). Funktio palauttaa lupausobjektin, joka käsitellään samalla tavalla kuin asiakkaan haussa.

4.3.3 Asiakkaan arkistointi ja poisto

Asiakastiedon poisto ei ole suoraan mahdollista, vaan käyttäjän halutessa poistaa asiakastiedon, se siirtyy ensin arkistoiduksi tiedoksi, jonka jälkeen se on mahdollista poistaa kokonaan järjestelmästä. Arkisto toimii niin sanotusti roskakorina, josta on mahdollisuus ottaa asiakastieto takaisin käyttöön ja varmisteena sille etteivät tiedot josta muut sovelluksen osat voivat olla riippuvaisia, pääse katoamaan. Asiakastiedon suora poisto aiheuttaisi myös asiakkaan hoitotapahtumien katoamisen ja esimerkiksi raportointiin saattaisi aiheutua vääristymiä.

Kun asiakas arkistoidaan, merkitään tietokannan tietoihin kyseisen asiakkaan kohdalle archived. Arkistoidut asiakkaat on mahdollista poistaa kokonaan järjestelmästä, jolloin myöskin heidän hoitotapahtumat poistuvat. Arkistointiin ja poistoon käytetään ClientService-palvelun metodeita ja ne tekevät GET-pyyntöjä taustajärjestelmälle. Pyynnössä lähetetään mukana asiakkaan id, jolla taustajärjestelmä tunnistaa kyseisen asiakkaan.

Arkistoiduille asiakkaille on oma sivunsa ja sivulla ovat samat ominaisuudet asiakkaiden järjestämiselle ja hakemiselle kuin arkistoimattomilla asiakkailla. Arkistoiduissa asiakkaassa käytetään samaa näkymää kuin normaalissa asiakkaassa, mutta arkistoiduissa asiakkaissa on poistettu mahdollisuus normaaleihin toimintoihin kuten asiakkaan muokkaukseen. Tämä tehdään näkymässä Angularin ng-if -direktiivillä, jolla piilotetaan ja näytetään eri tietoa riippuen siitä, onko asiakas arkistoitu.

4.4 Hoitotapahtumien hallinta

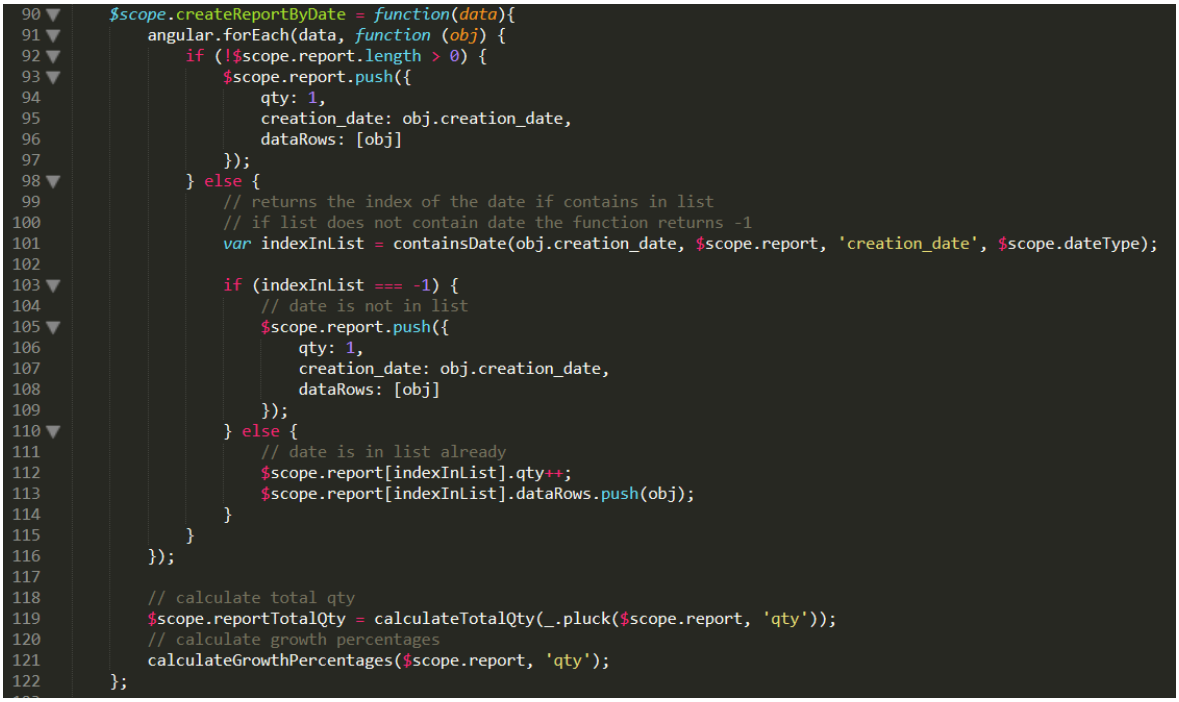
Hoitotapahtumien hallinta toimii sovelluksessa melko lailla yhtäläisesti kuin potilastietojen hallintakin. Hoitotapahtumia on mahdollista katsoa, luoda, muokata ja poistaa. Hoitotapahtuma on aina riippuvainen asiakkaasta: sitä ei voi luoda yksinään vaan hoitotapahtumaan on liitettävä aina asiakas.

Hoitotapahtumaan talletetaan hoitotapahtuman nimi, kesto, hinta, veroprosentti sekä lisätiedot. Talletetut tiedot lähetetään ensin ohjaimelta palvelulle (EventService), jonka jälkeen palvelu lähettää tiedot HTTP-POST pyyntönä taustajärjestelmälle. Palvelu toteuttaa sovelluksessa käytettyä kaavaa ja palauttaa ohjaimelle lupausobjektin.

Sovelluksessa hoitotapahtumien tietoja on mahdollista tallettaa valmiiksi pohjiksi. Valmiit pohjat haetaan taustajärjestelmästä GET-pyyntönä ja ne ovat valittavissa hoitotapahtumaa luodessa.

4.5 Raportointi

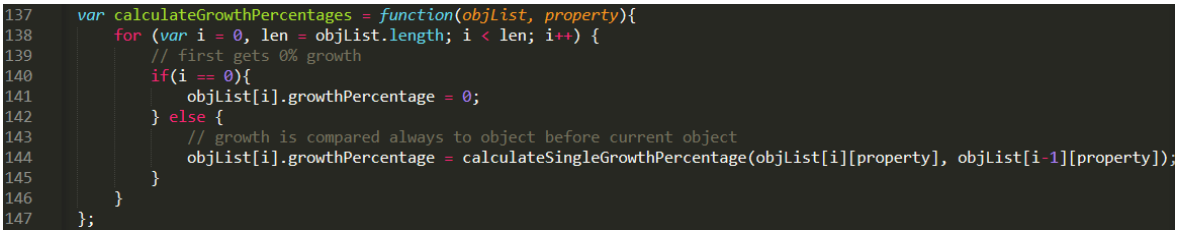
Hierojankortin raportointiominaisuus toteutettiin päämoduulista erillisenä moduulina. Raportoinnin avulla käyttäjä voi nähdä ajan mukaan uusien asiakkaiden luontipäivät ja hoitotapahtumien määrät ja tuotot. Raportissa lasketaan myöskin rivien muutosprosentti, joka kertoo tarkasteltavan yksiköiden, kuten uusien asiakkaiden määrän muutoksesta tietyllä aikavälillä. Laskeminen ja datan operaatiot tapahtuvat asiakaspäässä ja taustajärjestelmän rooliksi jää tiedon haku ja palautus.



```
90    $scope.createReportByDate = function(data){
91        angular.forEach(data, function (obj) {
92            if (!$scope.report.length > 0) {
93                $scope.report.push({
94                    qty: 1,
95                    creation_date: obj.creation_date,
96                    dataRows: [obj]
97                });
98            } else {
99                // returns the index of the date if contains in list
100               // if list does not contain date the function returns -1
101               var indexInList = containsDate(obj.creation_date, $scope.report, 'creation_date', $scope.dateType);
102
103               if (indexInList === -1) {
104                   // date is not in list
105                   $scope.report.push({
106                       qty: 1,
107                       creation_date: obj.creation_date,
108                       dataRows: [obj]
109                   });
110               } else {
111                   // date is in list already
112                   $scope.report[indexInList].qty++;
113                   $scope.report[indexInList].dataRows.push(obj);
114               }
115           }
116       });
117
118       // calculate total qty
119       $scope.reportTotalQty = calculateTotalQty(_.pluck($scope.report, 'qty'));
120       // calculate growth percentages
121       calculateGrowthPercentages($scope.report, 'qty');
122   };
```

Kuva 30 Ohjelmakoodia uusia asiakkaita edustavasta raportti-objektista

Raportointi toteutettiin käyttäen JavaScript-objektia, johon käärittiin kaikki raportin sisältämä tieto. Raportin tiedot jaettiin käyttämällä avaimena hoitotapahtuman tai asiakkaan päiväystä, jolloin kukin rivi raportissa yksilöitiin päivän mukaan. Kuvassa 30 esitetään uusien asiakkaiden raportin luontikoodia. Rivillä 91 käydään kaikki data nimisen muuttujan tiedot, jotka tässä tapauksessa ovat palvelimelta saadut asiakastiedot. Raportti-objekti alkaa tämän jälkeen koostumaan asiakkaiden luontipäivämääristä, niiden määristä sekä luontipäivänä luodusta asiakkaista (dataRows). Mikäli asiakkaan luontipäivä löytyy jo raportista, nostetaan sen päivän kappalemäärää sekä lisätään luontipäivältään sama asiakas kyseiseen riviin, muussa tapauksessa lisätään kyseinen päivä uutena rivinä raporttiin.



```
137   var calculateGrowthPercentages = function(objList, property){
138       for (var i = 0, len = objList.length; i < len; i++) {
139           // first gets 0% growth
140           if(i == 0){
141               objList[i].growthPercentage = 0;
142           } else {
143               // growth is compared always to object before current object
144               objList[i].growthPercentage = calculateSingleGrowthPercentage(objList[i][property], objList[i-1][property]);
145           }
146       }
147   };
```

Kuva 31 Raportin muutosprosentin lisäävä funktio

Raporttiin toteutettiin myös muutosprosentin laskeminen, joka asiakasraportissa tarkoitti uusien asiakkaiden määrän muutosta. Esimerkiksi mikäli edellisessä kuussa oli lisätty 2 asiakasta ja seuraavassa kuussa 4 olisi muutosprosentti 100. Kuvassa 31 esitetään ohjelmakoodia, joka laskee muutosprosentin koko raportille. CalculateSingleGrowthPercen-

tage-funktio palauttaa sille parametrina annettavien arvojen muutosprosentin. Rivillä 145, sille lähetetään silmukan nykyinen arvo ja sitä edeltävä arvo.

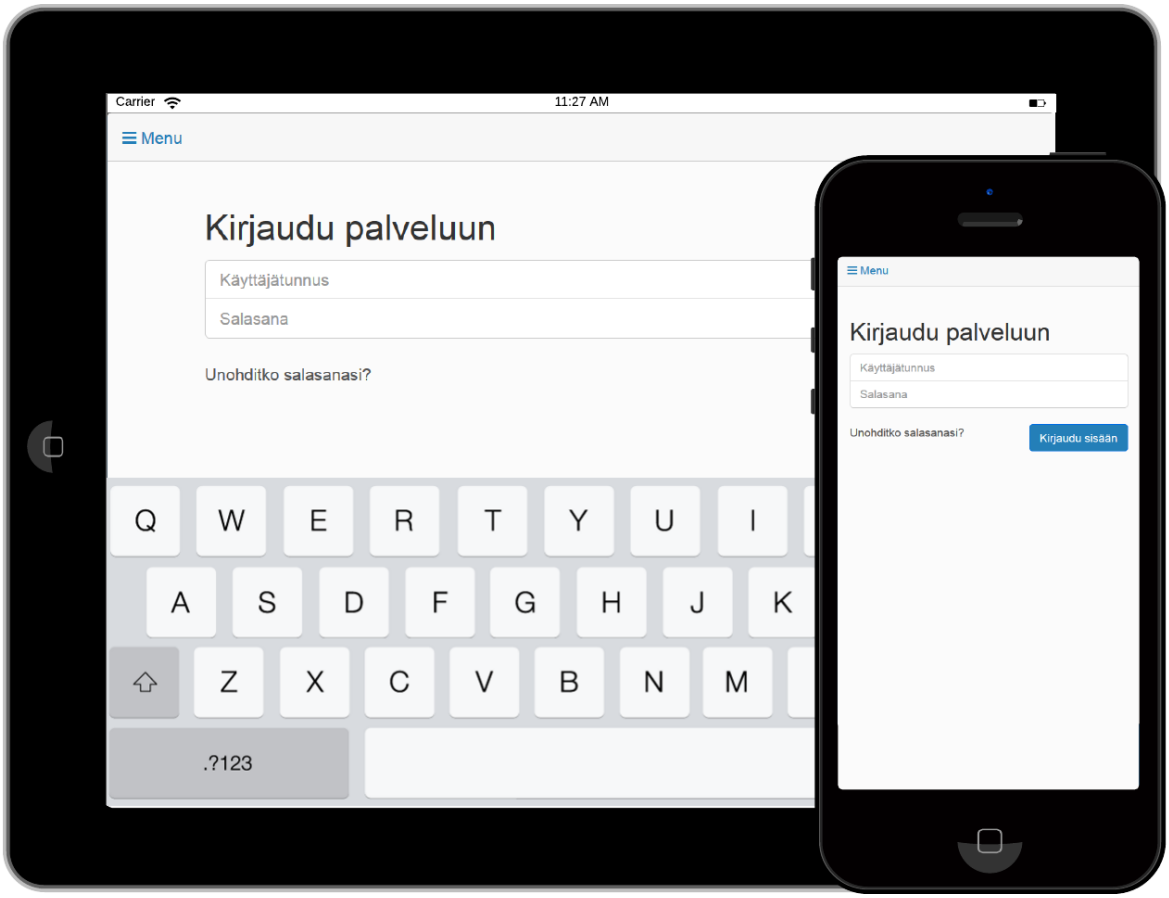
4.6 Käyttäjätietojen hallinnointi

Hierojankortissa käyttäjän on mahdollista muuttaa omia käyttäjätietojaan. Käyttäjätiedot lähetetään objektina taustajärjestelmälle, jossa taustajärjestelmä tallettaa muutokset tietokantaan. Asiakaspään toteutus tehtiin samalla reseptillä jota käytetään pitkin sovellusta. Reseptissä ohjain lähettää skoopin tiedot palvelulle, joka vastaa HTTP-pyynnön tekemisestä. Palvelun saadessa skoopin tiedot, se palauttaa ohjaimelle lupausobjektin, johon liitetään HTTP-pyynnön onnistumisen tai epäonnistumisen tila.

Käyttäjätiedoista on mahdollista myös vaihtaa salasanaa, jossa Angularilla tehdään validaatio salasanojen samankaltaisuudesta. Mikäli salasana toistuu samana kahdesti, annetaan käyttäjän lähettää salasananvaihto-pyyntö. Lopullinen validaatio tehdään palvelinpäässä, jossa lähetettyä vanhaa salasanaa ja palvelimelta löytyvää salasanaa verrataan, sekä tarkistetaan että uudet salasanat täsmäävät.

4.7 Käyttöliittymä ja käytettävyys

Käyttöliittymän toteutuksessa käytettiin apuna Angular Mobile UI:ta. Käyttöliittymä tehtiin responsiivisen suunnittelun pohjalta ja jolloin sen sai toteutettua samalla työpöytä- sekä mobiilikäyttöön. Kehitys tehtiin mobiili ensin, jolloin mobiilikäyttöliittymä suunniteltiin ennen työpöytäkäyttöliittymään keskittymistä, sillä toimeksiantajalle oli erittäin tärkeää sovelluksen toimivuus mobiilissa. Responsiivisuus toteutettiin pitkälle Bootstrapin ruudukoiden avulla. Kuvassa 32 esitetään responsiivisuutta tabletin ja puhelimen näkökulmasta, jossa kirjautumissivun kentät pienenevät ja suurenevat näytön koon mukaan.

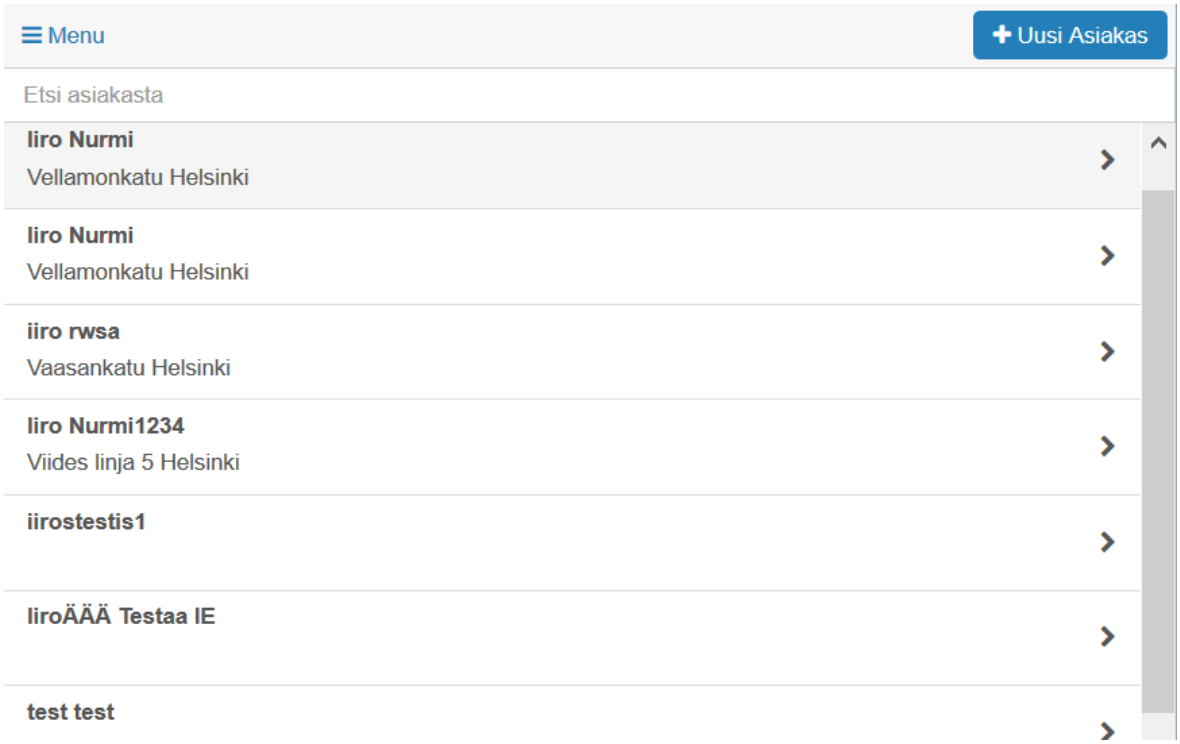


Kuva 32 Hierojankortin kirjautumissivu tabletilla ja älypuhelimella

Käytettävyyteen kiinnitettiin huomiota projektissa huomattavasti. Toimeksiantaja halusi minimoida kirjoittamisen määrää sovelluksessa, sillä etenkin mobiililla kirjoittaminen on hankalaa. Tämä osoittautui jokseenkin hankalaksi tehtäväksi, sillä asiakasrekisteriin kirjauksien tekemistä oli vaikeaa automatisoida. Pyrimme käyttämään valmiita valintoja esimerkiksi potilaan sairauksien kohdalla, jolloin käyttäjä voi merkitä sairauden potilaalle valitsemalla sairauden valmiiksi määritellyistä vaihtoehdoista. Hoitotapahtumien merkitsemiseen teimme mahdollisuuden käyttäjälle lisätä hoitotapahtumapohjia usein toistuvien hoitotapahtumien kohdalla.

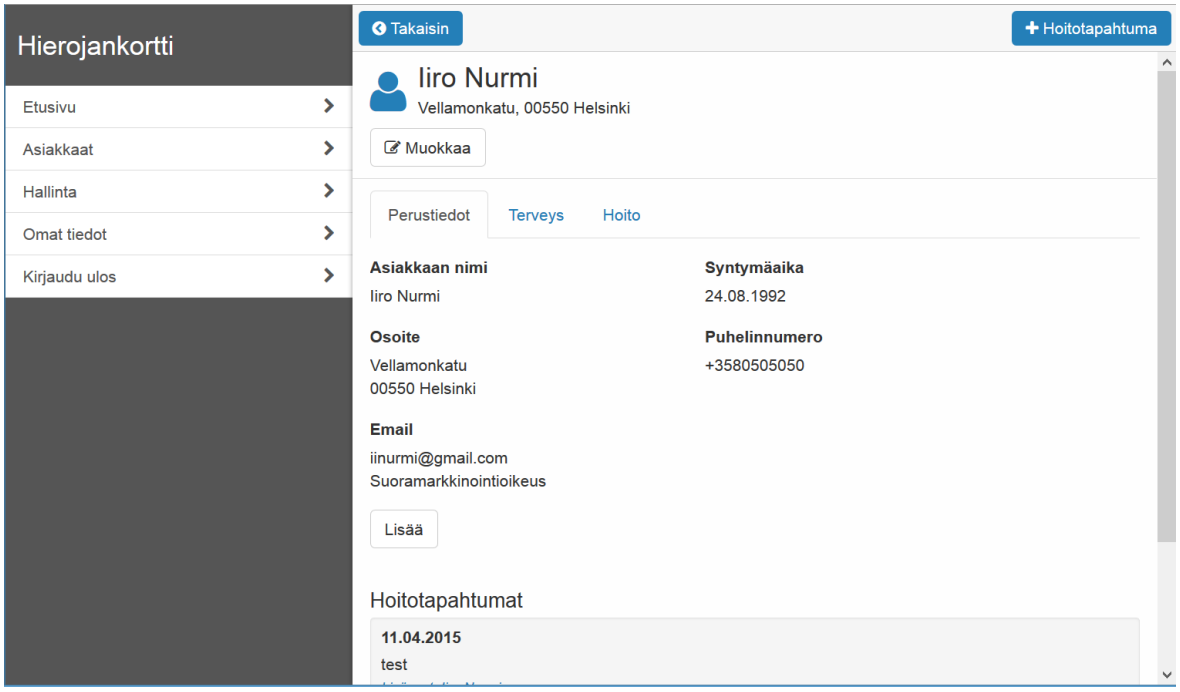
4.7.1 Potilaan hallinnan näkymät

Käytimme paljon aikaa asiakasnäkymien hiomiseen. Asiakasnäkymissä oli tärkeää että tiedot on järjestetty tärkeysjärjestykseen ja osioitu järkevästi, jotta käyttäjä pystyy helposti havaitsemaan itselleen tärkeän tiedon.



Kuva 33 Kaikkien asaikkaiden listanäkymä

Kuvassa 33 esitetään Hierojankortin asiakaslistanäkymää. Tältä sivulta käyttäjän on mahdollista selata, hakea sekä järjestää omaa asiakaslistaansa. Hakusanana on mahdollista käyttää asiakkaan etu- tai sukunimeä, kaupunkia tai katuosoitetta.

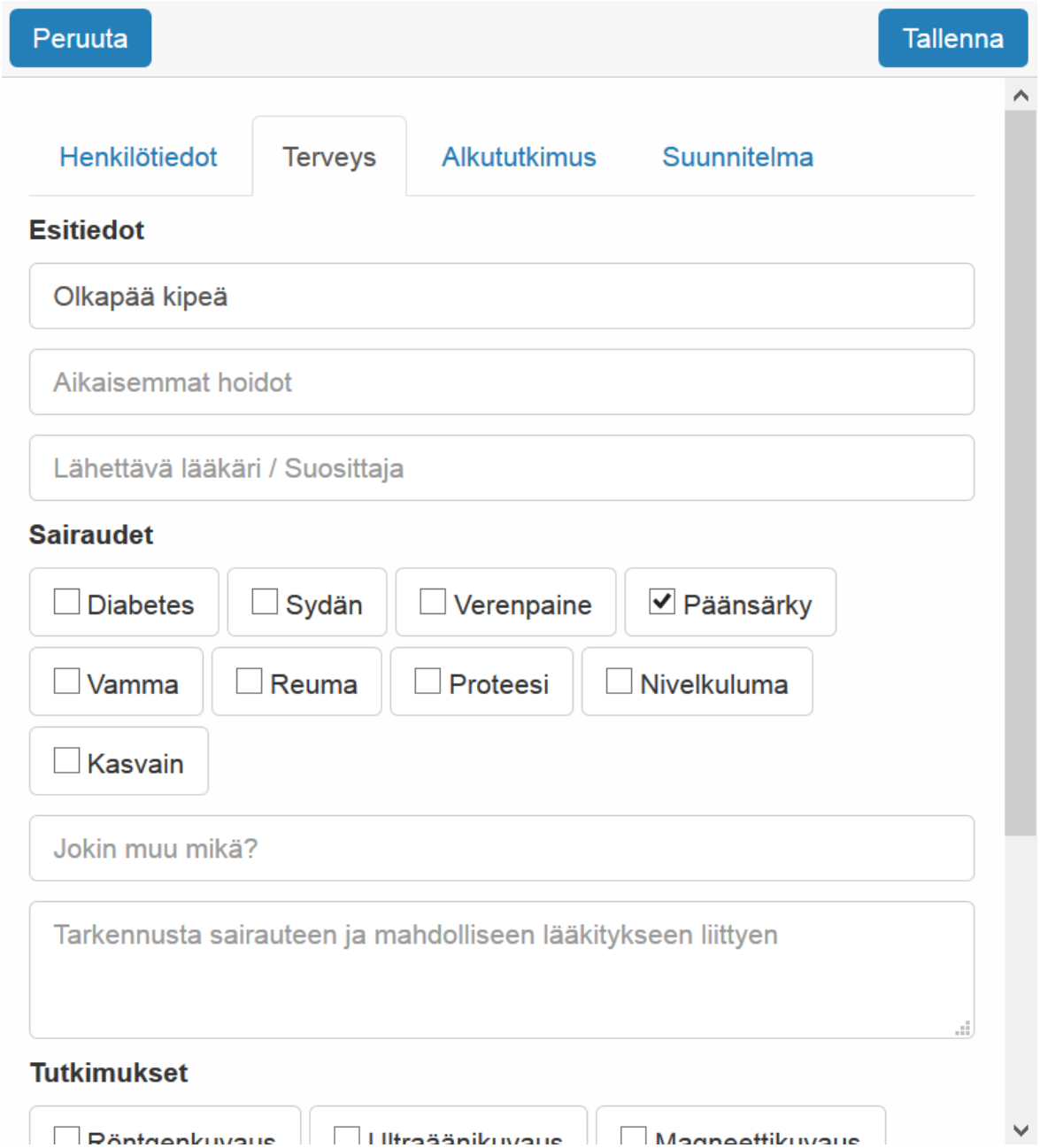


Kuva 34 Yhden asiakkaan näkymä työpöytäkäyttöiittymässä

Kuvassa 34 näkyy yhden asiakkaan näkymä ja tietojen ja toiminnallisuuksien osiointi. Vasemmalla on Hierojankortin päänavigaatio. Se liukuu mobiililla sormen raahauksella ja on automaattisesti piilossa. Yläpalkissa on napit navigaationa edelliselle sivulle ja jollekin sivulla tehtävälle toiminnolle. Yläpalkin oikealla puolella on hoitotapahtuman lisäys, joka on tärkein toiminto jota asiakasnäkymässä tehdään. Yläpalkista alaspäin katsottuna on alue johon merkittiin aina käyttäjälle sivun kategorisoiva tieto. Yläpalkin alla lukee asiakkaan nimi, jolloin on helppo hahmottaa olevansa kyseisen asiakkaan näkymässä.

Projektin keskivaiheella huomasimme ikonien tuovan sovellukselle hyvää visuaalista ilmettä, joten aloimme käyttämään niitä aina tietyn tyyppisen näkymän kohdalla. Esimerkiksi kuvassa 34 asiakkaan nimen vasemmalta löytyy hahmo-ikoni.

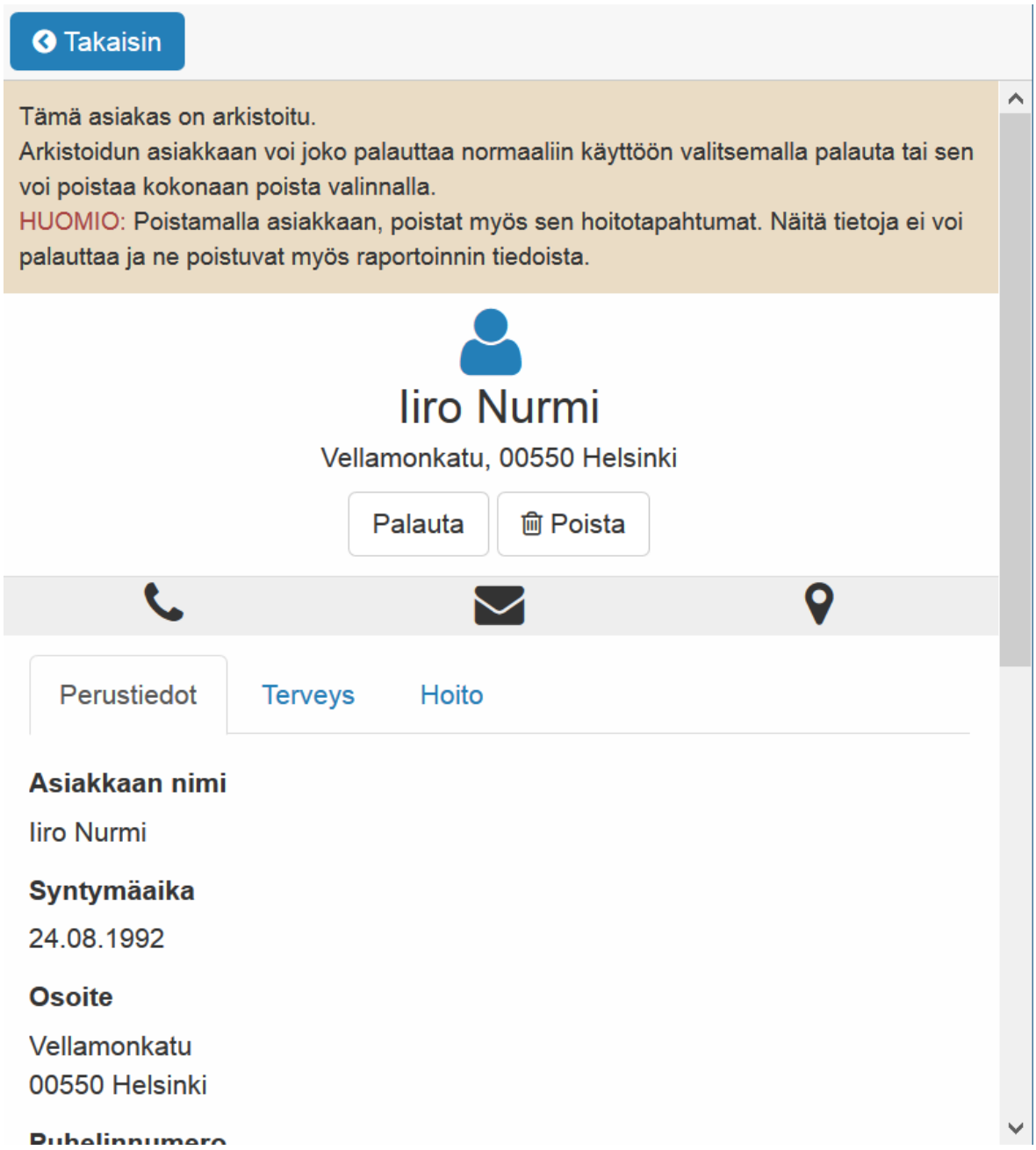
Asiakkaan tietoja pyrittiin ryhmittelemään ja piilottamaan painikkeiden taakse. Tämä tehtiin selkeyden takia. Välilehtiä käytettiin tiedon ryhmittelyyn ja asiakkaan kaikki tiedot eivät olleet näkyvissä pitkillä sivuilla. Lisää-napilla vähennettiin sivun pituutta ja sitä painamalla käyttäjä pystyy näkemään esimerkiksi asiakkaalle merkityt lisätiedot. Tietojen piilotusta tehtiin Angularin ng-if, ng-show sekä ng-hide -direktiiveillä. Nyrkkisääntönä pidimme sitä, että ng-if oli käytössä mikäli halusimme poistaa elementit kokonaan DOM:ista, joita olivat esimerkiksi kuvat ja isommat datamäärät, ja ng-hide ja ng-show:ia käytettiin pienempien osioiden piilottamiseen ja näyttämiseen.



Kuva 35 Asiakkaan lisäyksen ja muokkauksen näkymä pienessä ruudussa

Käytimme asiakkaan lisäykseen ja muokkaamiseen samaa näkymää. Ne erosivat toisistaan ainoastaan sillä tavalla, että muokkauksessa asiakkaan tiedot täytettiin automaattisesti muokattavan asiakkaan tiedolla. Kuvassa 35 esitetään asiakkaan muokkauksen ja lisäyksen näkymää. Tiedot pyrittiin jaottelemaan välilehdillä eri kategorioihin. Tietojen ryhmittelyyn välilehdillä käytimme väliotsikoita kuten kuvassa (numero) esitiedot ja sairaudet. HTML:n placeholder-tekstiä (kentän sisällä olevaa tekstiä) käytettiin yksittäisten kenttien otsikointiin. Tämä metodi säästi ruudulta tilaa, mutta oli myös vaikeasti hahmotettavissa sillä käyttäjän kirjoittaessa kenttään, placeholder-teksti katoaa.

Kuvasta 35 voi nähdä yläpalkin päätoiminto-napin muutoksen. Asiakasnäkymässä vasemmalla oli ”takaisin” ja oikealla ”uusi hoitotapahtuma” -nappula. Asiakkaan luomisessa ja muokkaamisessa yläpalkista löytyvät peruuta ja tallenna toiminnot. Dynaamiset yläpalkit toteutettiin Angular Mobile UI:n contentFor-direktiivillä, jonka avulla HTML-elementin sisältöä voidaan muuttaa helposti.



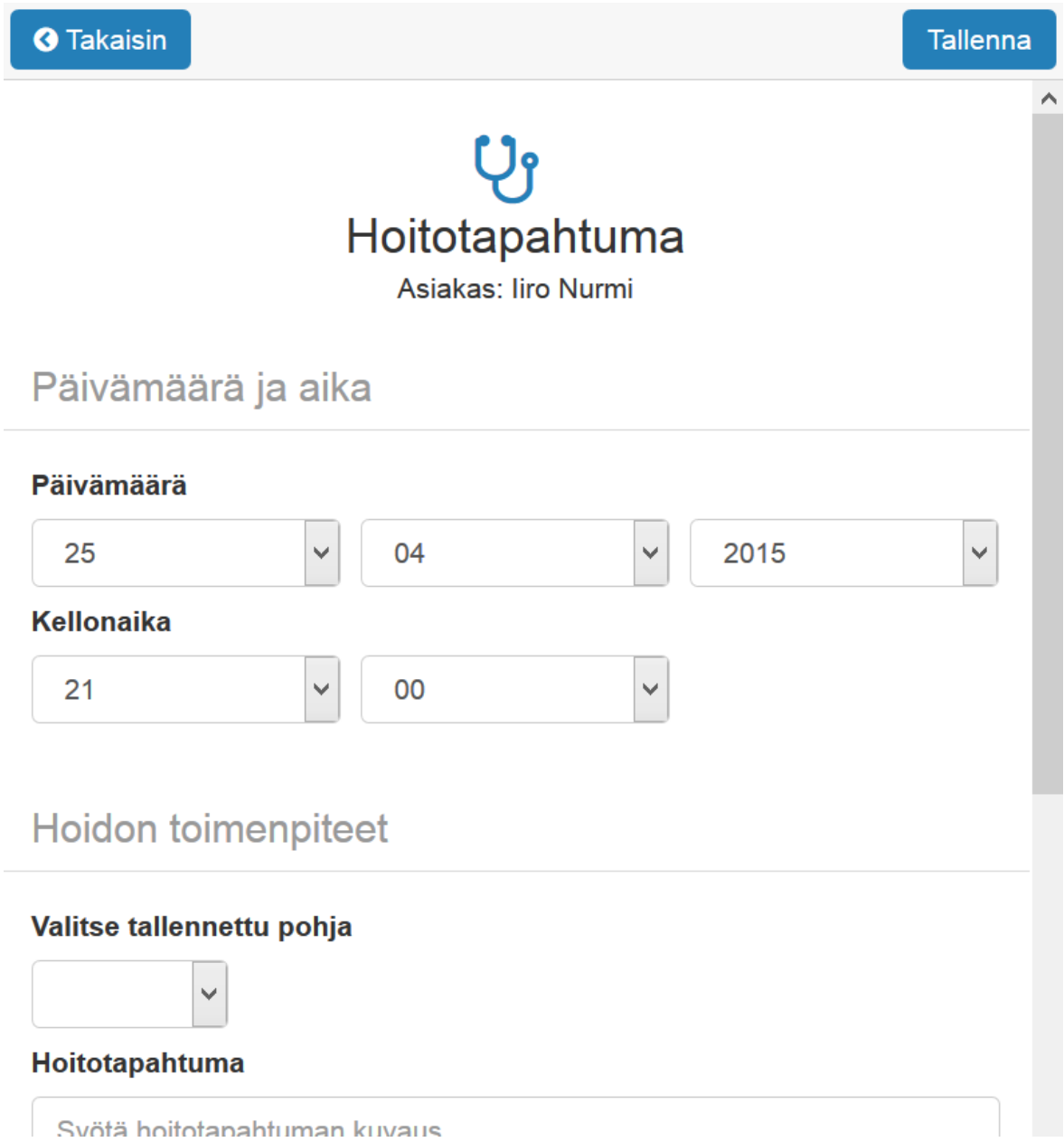
Kuva 36 Arikistoidun asiakkaan näkymä pienessä ruudussa

Kuvassa 36 esitellään arkistoidun asiakkaan näkymää. Asiakkaasta on heti nähtävissä, että se on arkistoitu ”arkistointivärillä” korostetun inforuudun avulla. Kuvassa voi huomata käyttöliittymän mobiiliryhmittäytymisen verrattuna työpöytäversion ryhmitykseen. Asiakkaan tiedot eivät rivity kahdelle kolumnille, asiakkaan nimi ja tiedot keskitetään sekä yli-

määräinen rivi lisätään näkymään. Tämä toteutettiin Bootstrapin ruudukolla, jolloin yksi rivi on mobiililla täysikokoinen (col-xs-12) ja työpöydällä n. yksi kolmasosa ruudusta (col-md-4). Ylimääräisellä rivillä on toiminnot asiakkaalle soittamiselle, sähköpostin lähettämiselle ja reitin hakemiselle.

4.7.2 Hoitotapahtumien näkymä

Hoitotapahtumille sovellettiin samankaltaisia käyttöliittymäsääntöjä kuin asiakkaillekin. Hoitotapahtumilla ei kuitenkaan ole samankaltaista hakunäkymää kun asiakkailla, josta voisi selata kaikkia hoitotapahtumia. Hoitotapahtumat löytyvät aina asiakasnäkymässä alhaalla sijaitsevasta listasta.



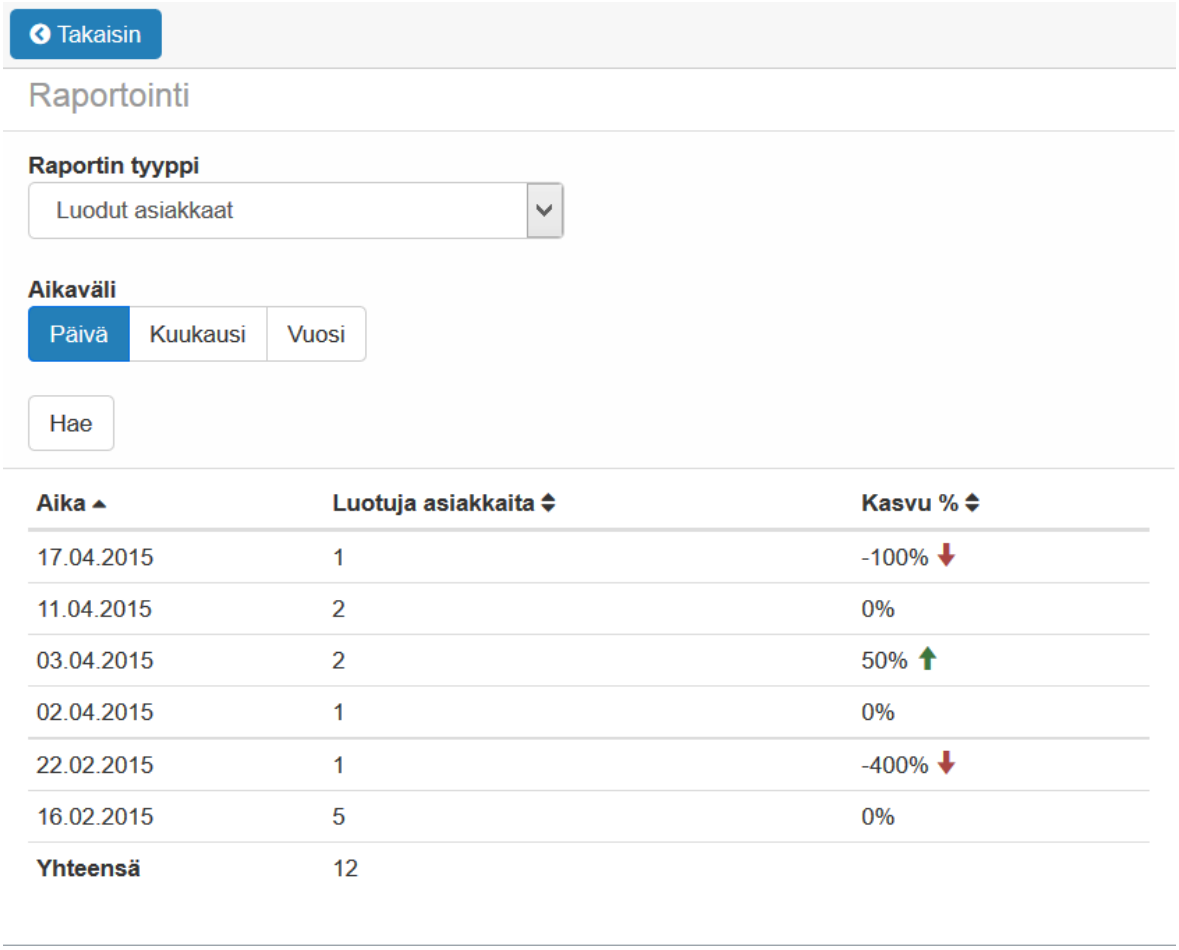
Kuva 37 Hoitotapahtuman lisäys- ja muokkausnäkymä pienessä ruudussa

Asiakasnäkymästä tuttua reseptiä käytetään hoitotapahtuman näkymässä (kuva 37): ikoni kuvaa näkymän tyyppiä ja sen alla on kyseisen asiakkaan nimi. Tiedot jaettiin suurilla väliotsikoilla toisin kuin asiakasnäkymässä, jossa käytettiin välilehtiä. Hoitotapahtuman sivu pystyttiin pitämään lyhyenä, jolloin tietoa ei tarvinnut jakaa vaakatasolla (välilehdet). Hoitotapahtuman ajan merkitsemiseen käytettiin HTML-alasvetovalikoita, jotka korvasivat kirjoittamisen tarpeen. Alasvetovalikot toimivat mobiililla ja eri selaimilla suhteellisen samankaltaisesti ja hyvin, jolloin selainten väliset eroavaisuudet eivät päässeet vaikuttamaan käyttökokemukseen.

Hoitotapahtumanäkymän käyttäjäystävällisyyttä pyrittiin lisäämään ajan ja päivän asettumisella oletuksena käyttäjän lokaalin ajan mukaan sekä hoitotapahtumapohjilla. Hoitotapahtumapohjat mahdollistavat käyttäjää asettamaan pohjia usein tapahtuville hoitotapahtumille. Käyttäjän valitessa valmiin pohjan tapahtuman nimi, hinta, kesto sekä veroprosentti täyttyvät automaattisesti pohjan mukaan. Hoitotapahtumapohjilla säästytään turhalta kirjoittamiselta ja etenkin mobiilikäyttö helpottuu.

4.7.3 Raportoinnin näkymä

Raportin näkymään päädyimme visualisoimaan raportin taulukkona. Taulukkoon voi ladata informaatiota hoitokäyntien ja asiakkaiden perusteella.



Kuva 38 Raportoinnin käyttöliittymä pienennetyssä ruudussa

Kuvassa 38 esitetään uusien asiakkaiden määrää päivien perusteella. Sivu on jaettavissa kahteen osaan: raportoinnin kontrolleihin ja raportin taulukkoon. Kontrolleista käyttäjä voi valita raportin tyypin ja aikavälin. Taulukko visualisoi haetun tiedon.

Taulukossa haettu tieto näytetään riveittäin ajan mukaan ja näytetään sen muutosprosentti. Taulukko muodostetaan käymällä 4.5 luvussa esiteltyä report-objektia läpi. Taulukon muutosprosenttien nuolien logiikka on toteutettu näkymään käyttäen Angularin ng-class-direktiiviä. Direktiivi määrittelee kasvuprosenttiriville CSS-luokan kasvuprosentin mukaan: mikäli kasvuprosentti on negatiivinen, riville määritellään punainen alaspäin suuntaava nuoli, mikäli positiivinen, nuoli on vihreä ja osoittaa ylös.

5 Pohdinta

Tämä luku käsittelee opinnäytetyössä viimeistellyn sovelluksen lopputuloksia. Luvussa pohditaan lopullisen sovelluksen toteutumista ja haasteita sekä jatkokehitysmahdollisuuksia. Luvussa myös käsitellään opinnäytetyöprosessin aikana tapahtunutta oppimista.

5.1 Lopputuloksen arviointi

Hierojankortti projektina on ollut itselleni hieno kokemus. On ollut mielenkiintoista nähdä kuinka sovellus on kehittynyt ideasta lopulliseksi produktiksi. Lopputuotos on niin itseni kuin toimeksiantajan mieleen.

Opinnäytetyön tavoitteena oli toteuttaa määrittelyvaiheen ominaisuudet ja kehittää niitä julkaistavaan kuntoon. Ominaisuuksiin kuuluivat asiakastietojen ja hoitotapahtumien hallinta, raportointi sekä mahdollisuus käyttäjätietojen muuttamiseen. Kaikki edellä mainitut ominaisuudet toteutettiin opinnäytetyön aikana julkaistavaan kuntoon.

Toimeksiantajalle opinnäytetyöstä oli erittäin paljon hyötyä, sillä sovelluksen on tarkoitus toimia toimeksiantajan ensimmäisenä myytävänä tuotteena. Opinnäytetyössä opitut parhaat käytännöt ja teoria edesauttavat toimeksiantajaa, sillä Hierojankortin ohjelmakoodi on opinnäytetyön jälkeen selkeämmin jaoteltu ja tulevat lisäominaisuudet osataan toteuttaa paremmin tietotaidon kasvettua.

Hierojankortin tuotantoversio ei sisällä hirveästi ominaisuuksia, mutta toisaalta se sopii tarkoitukseensa sillä Hierojan tarpeet asiakkuudenhallintajärjestelmän suhteen eivät ole kovinkaan suuret. Hierojankortin yksinkertaisuudella on etunsa, sillä käyttäjän on helppo ymmärtää mitä sovelluksella on tarkoitus tehdä. Hierojankortti on ikään kuin seuraava luonnollinen askel paperisista asiakaskorteista sähköiseen järjestelmään.

Sovelluksen toteutuksen haasteellisena osuutena sovelluksen rakentamisen mobiiliystävälliseksi. Mobiiliystävällisyyden toteuttaminen ei ollut teknisesti haastavaa, vaan tarvitsi enemmän visuaalista silmää ja käyttöliittymäsuunnittelua. Asiakashallinnan valitettavana piirteenä on, ettei asiakkaiden tietojen kirjoittamiselta voi välttyä ja asiakkaisiin liittyvää tietoa on paljon. Tämä ei tule luultavasti jatkossakaan muuttumaan, sillä asiakkaista halutaan kerätä yhä enemmän ja enemmän tietoa. Ratkaisuna tiedon paljouteen pyrittiin kirjoittamisen tarvetta vähentämään valmiilla valinnoilla, sekä piilottamalla ruudulta mahdollisimman paljon ylimääräistä tietoa. Mielestäni tehtävässä onnistuttiin hyvin. Tablettikäytön yleistyessä tulevaisuudessa, näen tabletin olevan suosituin tapa käyttää Hierojankorttia.

Tablettia on helppo kantaa mukana ja sen suuremmalla näppäimistöllä on helpompi kirjoittaa verrattuna puhelimella kirjoittamiseen.

Projekti toteutui lopulta mallikkaasti huolimatta siitä, että Angularin tuntemus oli projektin alussa heikkoa ja ohjelmakoodia jouduttiin refaktoroimaan paljon projektin edetessä. Angularin valitseminen asiakaspään teknologiaksi osoittautui hyödylliseksi. Angular-sovelluskehys sopi erityisen hyvin Hierojankortin tarpeisiin sen tarjoaman tiedon sidonnan, valmiiden direktiivien ja palveluiden takia. Sovelluskehystä voi suositella Hierojankortista pohjatuvan kokemuksen myötä etenkin CRUD-sovellusten rakentamiseen.

5.2 Oma oppiminen

Projekti oli oppimisen kannalta itselleni mieluisa. Käytän työelämässä samaa sovelluskehystä ajoittain ja opinnäytetyön syventävät tiedot tulivat tarpeen. AngularJS vaikuttaa päällepäin helpolta sovelluskehykseltä ja niin se onkin. Sovelluskehyksen ymmärtäminen vaatii kuitenkin syventymistä kehykseen ja jotkut konseptit vaativat edistyneempää ymmärrystä JavaScriptista. Esimerkiksi ymmärtääkseni service- ja factory-palvelun eron luin useamman aihetta käsittelevän artikkelin, sillä palvelutyyppien eroa on vaikea osoittaa käytännössä.

Opinnäytetyö pakotti minut selvittämään asioita mihin en luultavasti perehtyisi niin tarkasti normaalissa ohjelmointiprojektissa, sillä olen enemmän tekijä kuin lukija. Perehtyminen Angularin dokumentaatioon ja kirjallisuuteen paransi huomattavasti ohjelmointitaitojani Angularin parissa, sillä tutustuin konsepteihin syvällisemmin. Teorian ja Angularin parissa toimivien asiantuntijoiden mielipiteiden lukeminen on opettanut minua Angularin parhaista käytännöistä paljon. Eniten opin verkkosovelluksen arkkitehtuurista. Angularin moduulit ja ohjelmarakenteet paransivat minun kykyäni hahmottelemaan sovelluksia osakokonaisuuksina. En sanoisi, että olen projektin jälkeen parempi ohjelmoija teknisesti, sillä Angularilla ohjelmointi ei loppujen lopuksi ole kovinkaan vaikeaa. Oma kehitys on tapahtunut enemmänkin kyvyssä hallita sovelluksen kokonaisarkkitehtuuria.

5.3 Jatkokehitys

Hierojankortin jatkokehitysmahdollisuuksia on monia. Ensimmäiset kehitysaskeleet ovat luultavasti palvelun nykyisten ominaisuuksien kehittäminen. Palvelua on vielä hiottava, ennen uusien ominaisuuksien tuomista sovellukseen. Palvelua parannetaan käyttäjiltä tulevan palautteen mukaan.

Tulevaisuuden suurena kehitysaskeleena näen Hierojankorttiin liitettävän ajanvarausjärjestelmästä. Hierojankorttiin liitettävää ajanvarausjärjestelmää voitaisiin myös myydä irrallisena tuotteena. Ajanvarausjärjestelmä on kuitenkin suuri kokonaisuus ja sille on varattava paljon kehitysaikaa.

Kehitystä tullaan tekemään myös sovelluksen yleistettävyyden kannalta. Potilasrekistereitä on käytössä monella muullakin alalla kuin hieronnassa kuten fysio- ja psykoterapiassa. Ohjelman laajentaminen useammalle alalle mahdollistaisi sovellukselle suuremman asiakaskunnan ja sovelluksen yleistettävyys on tulevaisuudessa yksi kehityskohteistamme.

Lähteet

Austin A. 2014. An Overview of AngularJS for Managers. Luettavissa: http://andrewaustin.com/an-overview-of-angularjs-for-managers/. Luettu: 12.2.2015.

AngularJS. 2015a. Introduction. Luettavissa: https://docs.angularjs.org/guide/introduction. Luettu: 12.2.2015.

AngularJS. 2015b. Routing & Multiple Views. Luettavissa: https://docs.angularjs.org/tutorial/step_07. Luettu: 12.2.2015.

AngularJS. 2015c. Dependency Injection. Luettavissa: https://docs.angularjs.org/guide/di. Luettu: 12.2.2015.

AngularJS. 2015d. Two-way Data Binding. Luettavissa: https://docs.angularjs.org/tutorial/step_04. Luettu: 12.2.2015.

AngularJS. 2015e. Services. Luettavissa: https://docs.angularjs.org/guide/services. Luettu: 12.2.2015.

AngularJS. 2015f. Providers. Luettavissa: https://docs.angularjs.org/guide/providers. Luettu: 12.2.2015.

AngularJS. 2015g. $http. Luettavissa: https://docs.angularjs.org/api/ng/service/$http. Luettu: 12.2.2015.

Ate, F. 22.6.2010. Taking Advantage of HTML5 and CSS3 with Modernizr. Luettavissa: http://alistapart.com/article/taking-advantage-of-html5-and-css3-with-modernizr. Luettu: 04.03.2015.

Ballve, M. 7.8.2013. The Future Of Mobile Development: HTML5 Vs. Native Apps. Business Insider. Luettavissa: http://www.businessinsider.com/html5-vs-native-apps-for-mobile-2013-6?op=1&IR=T. Luettu 15.04.2015.

Bégaudeau, S. 24.4.2014. Everything you need to understand to start with AngularJS. Generating blog posts. Luettavissa: http://stephanebegaudeau.tumblr.com/post/48776908163/everything-you-need-to-understand-to-start-with. Luettu: 23.03.2015.

Frain, B. 2012. Responsive Web Design with HTML5 and CSS3. Packt Publishing.

Bootstrap. 2015. CSS. Luettavissa: http://getbootstrap.com/css/. Luettu: 23.3.2015.

Bootstrap. 19.8.2013. Bootstrap 3 released. The Official Bootstrap Blog. Blogi. Luetttavissa: http://blog.getbootstrap.com/2013/08/19/bootstrap-3-released/. Luettu 23.03.2015.

Cousins, C. 28.5.2013. Flat design principles. Luettavissa: http://designmodo.com/flat-design-principles/. Luettu: 23.03.2015.

Diarium 2014. Fysioterapia-ohjelma. Luettavissa: http://www.diarium.fi/. Luettu: 16.2.2015.

Freeman, A. 2014. Pro AngularJS. Apress.

Garret, J. 18.2.2005. Ajax: A New Approach to Web Applications. Luettavissa: http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications/. Luettu 2.5.2015.

Github. 2015. Luettavissa: https://github.com/search?l=&o=desc&q=stars%3A%3E1&ref=advsearch&s=stars&type=Repositories. Luettu: 23.03.2015.

Green, B. Intro to AngularJS. Google Docs presentaatio. Luettavissa: https://docs.google.com/presentation/d/1H9u9xd0OE0W1o_5Aeug7uSR0AXXPidtWBQNmpi_ZP9I/edit?pli=1#slide=id.p. Luettu: 14.3.2015.

Haggard, M. 24.5.2013. Angular service or factory? Iffy Can. Blogi. Luettavissa: http://iffycan.blogspot.com.ar/2013/05/angular-service-or-factory.html. Luettu: 16.3.2015.

Ionic. 2015. Luettavissa: http://ionicframework.com/. Luettu: 01.04.2015.

Kukic A. 10.9.2014. AngularJS Best Practices: Directory Structure. Luettavissa: https://scotch.io/tutorials/angularjs-best-practices-directory-structure. Luettu: 4.4.2015.

Laki terveydenhuollon ammattihenkilöistä (28.6.1994/559). Luettavissa: http://www.finlex.fi/fi/laki/ajantasa/1994/19940559. Luettu: 16.2.2015.

Lehdonvirta, P., Korpela, J. 2013. HTML5 sovellusalustana. RPS-yhtiöt. Helsinki.

Lerner, A. 2013. ng-book – The Complete Book on AngularJS. Fullstack.io.

MariaDB 2015. Luettavissa: https://mariadb.org/en/about/. Luettu: 01.04.2015.

Marcotte E. 25.5.2010. Responsive Web Design. Luettavissa: http://alistapart.com/article/responsive-web-design. Luettu: 23.03.2015.

MDN. 2015. CSS media queries. Luettavissa: https://developer.mozilla.org/en-US/docs/Web/Guide/CSS/Media_queries. Luettu: 23.03.2015.

Meyer M. 2014. The Top 10 Mistakes AngularJS Developers Make. Luettavissa: https://www.airpair.com/angularjs/posts/top-10-mistakes-angularjs-developers-make. Luettu: 23.03.2015.

Mikowski, M. S. & Powell, J.C.2013. Single Page Web Applications JavaScript end - to - end. Manning Publications Co. Greenwich.

Mobile Angular UI. 2015a. Mobile Angular UI. Luettavissa: http://mobileangularui.com/. Luettu: 11.04.2015.

Mobile Angular UI. 2015b. Docs. Luettavissa: http://mobileangularui.com/docs/. Luettu: 11.04.2015.

Phonegap 2015. Luettavissa: http://phonegap.com/about/. Luettu: 04.03.2015.

Precht, P. 14.10.2014. Exploring Angular 1.3: One-time bindings. Thoughtram Blog. Blogi. Luettavissa: http://blog.thoughtram.io/angularjs/2014/10/14/exploring-angular-1.3-one-time-bindings.html. Luettu: 05.04.2015.

Rammer, I. & Weyer, C. 2013. Modularizing AngularJS Applications. Luettavissa: http://henriquat.re/modularizing-angularjs/modularizing-angular-applications/modularizing-angular-applications.html. Luettu 11.3.2015.

Smith D. 2014. Dave Smith - Deep Dive into Custom Directives - NG-Conf 2014. Verkko-video. Katsottavissa: https://www.youtube.com/watch?v=UMkd0nYmLzY. Katsottu: 16.1.2015.

Techopedia. Asynchronous Method Call. Luettavissa: http://www.techopedia.com/definition/25584/asynchronous-method-call. Luettu 06.04.2015.

Tiainen, S. 5.10.2014. Hierojakouluttaja. HAAGA-HELIA ammattikorkeakoulu. Haastattelu. Vierumäki.

Tyler M. 4.5.2015. AngularJS: Factory vs Service vs Provider. Blogi. Luettavissa: http://tylermcginnis.com/angularjs-factory-vs-service-vs-provider/. Luettu: 16.3.2015.

Wahlin D. 16.8.2013. Using an AngularJS Factory to Interact with a RESTful Service. Dan Wahlin Blog. Luettavissa: http://weblogs.asp.net/dwahlin/using-an-angularjs-factory-to-interact-with-a-restful-service. Luettu: 23.03.2015.

Webb C. 5.4.2013. Promise & Deferred objects in JavaScript Pt.1: Theory and Semantics. Blogi. Luettavissa: http://blog.mediumequalsmessage.com/promise-deferred-objects-in-javascript-pt1-theory-and-semantics. Luettu: 06.04.2015.

WhatIs.com 2013. Native App. Luettavissa: http://searchsoftwarequality.techtarget.com/definition/native-application-native-app. Luettu: 15.04.2015.

W3C. 2014a. HTML5 A vocabulary and associated APIs for HTML and XHTML W3C Recommendation 28 October 2014. Luettavissa: www.w3.org/TR/html5/introduction.html. Luettu: 04.03.2014.

W3C. 2014b. XMLHttpRequest Level 1. Luettavissa: http://www.w3.org/TR/XMLHttpRequest/#responsetexts. Luettu: 3.5.2015.