jamk.fi

ANALYSOINTIJÄRJESTELMÄN TOTEUTUS JA HYÖDYNTÄMINEN PELIALALLA

Annika Salo

Opinnäytetyö
Marraskuu 2014

Mediatekniikan koulutusohjelma
Tekniikan ja liikenteen ala

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

KUVAILULEHTI

Tekijä(t) Salo, Annika	Julkaisun laji Opinnäytetyö	Päivämäärä 5.11.2014
	Sivumäärä 63	Julkaisun kieli Suomi
		Verkkojulkaisulupa myönnetty (X)

Työn nimi
ANALYSOINTIJÄRJESTELMÄN TOTEUTUS JA HYÖDYNTÄMINEN PELIALALLA

Koulutusohjelma
Mediatekniikka

Työn ohjaaja(t)
Manninen, Pasi

Toimeksiantaja(t)
Poppaa Entertainment Oy

Tiivistelmä
Opinnäytetyössä keskitytään pelimaailmaan hieman tavallista poikkeavalla tavalla. Pelit ovat siirtyneet olohuoneista ihmisten mukana kulkeviksi, sillä nykyaikana pelit kuuluvat lähes poikkeuksetta älylaitteiden omistajien arkeen. Pelimarkkinat ovat pullollaan enemmän tai vähemmän erilaisia pelejä, jotka eivät ole vain sattumalta saaneet suurta suosiota. Yksi tärkeä osa pelin kehitystä on seurata kuka pelaa, millä laitteella, milloin ja miten. Näin osataan markkinoida oikeille henkilöille tai korjata tehtyjä virheitä.

Työssä käydään läpi erinäisten analysointipalveluiden ominaisuuksia ja tutustutaan olemassa oleviin tuotteisiin pelidatan analysoinnin saralla sekä vertaillaan näitä pääpiirteittäin. Toteutettavaan järjestelmään käytetyt teknologiat esitellään ja näiden perusteet käydään läpi. Varsinaisen toteutuksen rakenteet ja ratkaisut avataan esimerkein ja näiden kommunikointitavat ja kriittiset liittymäkohdat tutkitaan. Työssä käydään läpi myös varsinaiseen peliin yhdistettävä liitännäinen ja tämän testaus Unity-pelimoottorilla toteutetulla pelillä.

Toteutuksen jälkeen syvennytään miettimään pelidatan analysoinnin pohjimmaista merkitystä. Pohditaan, kuinka kerättyä tietoa voidaan oikeasti käyttää hyödyksi niin yrityksen puolesta taloudellisesti kuin käyttäjäkokemusten parantamiseksi. Lisäksi katseet käännetään tulevaisuuteen ja mietitään, miltä pelialan tulevaisuus näyttää ja erityisesti mihin pelien analysointijärjestelmät tulevat suuntaamaan.

Työn tuloksena syntyy analysointijärjestelmälle perusominaisuudet omaava kokonaisuus, joka muodostaa helposti ymmärrettäviä kaavioita pelialalle tärkeistä tiedoista liittyen pelaajien koukuttamiseen sekä pelitapoihin. Tiedot pelaajien toiminnoista ovat erittäin tarpeellisia Poppaa Entertainment Oy:n tapaiselle aloittelevalle pelitalolle, jolle järjestelmä tuotettiin.

Avainsanat (asiasanat)
Web-kehitys, Pelianalysointi, MongoDB, Go, AngularJS, Javascript, REST, Unity

Muut tiedot

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

DESCRIPTION

Author(s)	Type of publication	Date
Salo, Annika	Bachelor´s Thesis	5.11.2014
	Pages 63	Language Finnish
		Permission for web publication (X)

Title
DEVELOPMENT AND USE OF ANALYZING SYSTEM IN GAME INDUSTRY

Degree Programme
Media Engineering

Tutor(s)
Manninen, Pasi

Assigned by
Poppaa Entertainment Oy

Abstract
This thesis focuses on game industry from a slightly different point of view than usually seen. Games have moved from living rooms to traveling with people, since nowadays the majority of smart device owners play games daily. Game industry offers all kinds of games of which have gained their fame and players with good reasons. One of the most important aspects of game development is to find out by whom, when and how their games are played. With this kind of information games can be advertised to right people and occurred errors can be fixed as soon as they are found.

The thesis goes through features of different analyzing systems and opens up existing game analyzing systems and compares their pros and cons. The technologies used in the thesis are introduced and some of their basic usage is discussed. The structure of the actual implementation and some solutions are focused on more in detail and the communication ways between used technologies and some critical points are explained. The making of the Unity plugin is also explained along with the testing part with a real game.

After the implementation part the analyzing of data is reflected on. How to really benefit from the collected data not only from business aspect but also by improving user experience. The future of the game industry and especially how game analyzing systems are going to evolve is also discussed.

This thesis produces a game analyzing system with the most typical features possible. The system uses collected data to generate informative charts about players and their habits, which present quite critical knowledge in game industry. Such information is really important to start-up companies like Poppaa Entertainment Oy, by whom this system was assigned.

Keywords
Web development, Game analyzing, MongoDB, Go, AngularJS, Javascript, REST, Unity

Miscellaneous

Sisältö

Kuviot

Käsitteet

CSS

Cascading Style Sheets. Kieli, jolla HTML-sivujen muotoilu kuvataan.

DOM

Document Object Model eli puurakenne, jonka HTML-sivun elementit muodostavat.

FREE-TO-PLAY, F2P

Pelin rahoitusmalli, jossa itse pelistä ei makseta mitään vaan pelin sisällä voi ostaa erilaisia pelaamista helpottavia tavaroita.

PLUGIN

Sovelluksen liitännäinen, joka tarjoaa jonkun tietyn toiminnon tai ominaisuuden.

REST

Ohjelmointirajapintojen toteuttamiseen käytetty HTTP-protokollaan perustuva arkkitehtuurimalli.

RETENTION

Prosentuaalinen luku, josta käy ilmi, kuinka vanhojen käyttäjien pitäminen on onnistunut.

SVG

XML-pohjainen vektoroitu kuvaformaatti.

UNITY

Unity Technologiesin kehittämä pelimoottori, jolla voidaan tehdä 2D- ja 3D-pelejä eri alustoille.

WEBSOCKET

Tekniikka, joka antaa mahdollisuuden reaaliaikaiseen kaksisuuntaiseen viestintään eri osapuolten välillä

1 Työn lähtökohdat

1.1 Toimeksiantaja

Idea opinnäytetyön aiheeksi tuli Poppaa Entertainment Oy:ltä (”Poppaa”), joka on aloitteleva mobiilipelejä tuottava yritys Pohjanmaalta. Peleissään Poppaa haluaakin keskittyä hauskuuteen ja pyrkiä tuomaan kaikille mahdollisuuden pelata pelejä. Tästä syystä he tuottavatkin pelejä F2P-mallilla, jossa pelaaja saa peruspelin ilmaiseksi ja rahaa käyttämällä voi ostaa erilaisia pelaamista helpottavia tavaroita pelin sisällä.

Vaikka pelitalon toiminta on vasta alkukuopissaan, on ensimmäisen pienehkön pelin julkaiseminen opettanut viidestä hengestä koostuvalle tiimille paljon jo pelkästään siitä, kuinka saada peli tarjolle sovelluskauppaan. Työn alla onkin jo useampia samantyyppisiä pieniä projekteja, ja pöytälaatikosta löytyy ideoita enemmän kuin tarpeeksi.

Yritys keskittyy ainakin alkutaipaleellaan luomaan pelejä ensisijaisesti iOS-mobiilikäyttöjärjestelmille eli Applen iPad-taulutietokoneille sekä iPhone-älypuhelimille. Tämä helpottaa ja nopeuttaa pelien tekemistä, sillä laitekohtaisia vaatimuksia tehokkuuden, näytön koon ja muistin koon suhteen on suppeammin.

1.2 Tavoitteet ja menetelmät

Opinnäytetyö lähestyy peliteollisuutta hieman toisenlaisella tavalla. Työn aiheena oli luoda järjestelmä, jonka avulla peleistä voidaan kerätä tietoa sekä visualisoida siitä kaaviota hyödynnettäväksi peliä ja pelaajia tutkittaessa ja käyttötapoja selvittäessä. Järjestelmää testattiin ja kehitettiin Poppaan työn alla olevalla pelillä, jotta saataisiin mahdollisimman paljon realistista tietoa kerättyä. Yritys on jo saanut kokemusta alan muista analysointityökaluista, joiden perusteella on todettu näiden järjestelmien olevan syystä tai toisesta riittämättömiä heidän tarpeisiinsa.

Työn pääpaino oli tietokannan, palvelimen ja web-käyttöliittymän luomisessa, mutta koko ajan seurattiin myös sitä, kuinka asiat hoidetaan pelattavan pelin puolella. Työn toteutuksessa panostettiin reaaliaikaisuuteen sekä nopeuteen unohtamatta järjestelmän

käytön helppoutta. Työssä myös pohdittiin, kuinka kerättyä tietoa voidaan hyödyntää sekä tehokkaasti pelikokemusten osalta että taloudellisia hyötyjä tavoitellen.

Järjestelmän palvelinpuolen toteutukseen valikoitui Go-ohjelmointikieli ja selainpuolella www-sivujen luomisessa käytettiin JavaScript-pohjaista AngularJS-ohjelmistokehystä. Näitä valintoja täydensivät erilaiset apukirjastot ja liitännäiset. Tieto talletetaan MongoDB-tietokantaan, joka on lyömässä itseään läpi isommissakin julkisen tahon palveluissa (Production Deployments 2014). Käytettyihin teknologioihin perehdytään luvussa 3.

Tietoa lähetetään eri tahojen välillä REST-arkkitehtuuria hyödyntäen, jotka mahdollistavat erilaisten pelimoottorien ja ohjelmointikielien käytön pelitoteutuksissa. Tieto liikkuu nykyaikaisena JSON-muotoisena tekstinä eri tahojen välillä vanhahtavien XML-pohjaisten dokumenttimallien sijaan.

Ensisijaisesti opinnäytetyötä lähdettiin työstämään opiskelumielessä. Kaikista käytetyistä teknologioista oli hallussa perustietämys, mutta näiden taitojen kohentaminen ja tekniikoiden yhteiskäyttö on omalla mielenkiintoisuudellaan suorastaan ajanut opiskelua eteenpäin.

2 Pelitiedon analysointi

2.1 Miksi analysoidaan?

Vuosien saatossa videopeleistä on tullut koko kansan viihdettä. Enää näitä eivät pelaa ainoastaan pikkulapset ja parikymppiset syrjäytymisuhan alla olevat nuoret miehet vaan myös pullan tuoksuiset kotirouvat sekä rahaa tahkoavat liikemiehet. Alati kehittyvän teknologian ansiosta pelejä pelataan missä ja milloin vain mitä eriskummallisimmilla laitteilla.

Peliteollisuudessa taistellaan pelaajista ja kehitellään uusia ja ehkä hiotaan vanhempiakin peli-ideoita koukuttamaan kansalaisia unohtamatta tietenkään kokonaisvaltaista pelikokemuksen luomista äänillä, grafiikoilla ja toiminnallisuuksilla. Luodakseen kaikin puolin ikimuistoisen pelielämyksen, yhdeksi pelin teon tärkeimmistä vaiheista voidaan luokitella käyttökokemusten selvittäminen. Yksi oikein hyvä keino selvittää pelaajien

liikehdintää ja elämää pelissä on kerätä tietoa pelaamisen aikana ja säilöä tätä myöhempää analysointia varten. Pelistudiot ovat aikojen saatossa siirtyneet ulkoistamaan pelien analysointiin tarvittavat välineet niillä elantonsa hankkiville yrityksille, mikä antaa lisää aikaa panostaa itse peliin.

Pelit eivät kuitenkaan luonnollisesti ole ainut asia, jota voidaan analysoida. Erityisesti teknologian osa-alueella pelien analysointia lähellä on verkkosivujen ja -palveluiden analysointi, jotka molemmat myöskin paneutuvat kävijäseurantaan. Nk. web-analytiikalla mitataan verkkosivuston tärkeyttä erilaisten tavoitteiden ja tarpeiden täyttämisen avulla. Tavoitteena voi verkkokaupassa toimia esimerkiksi ostoksen tekeminen. Sitä, kuinka tämä käyttäjä pääsee tähän tavoitteeseen, on erityisen oleellista seurata palveluiden kehittämistä ajatellen. Eksyykö käyttäjä muille sivuille välillä? Kuinka kauan käyttäjä viipyy sivuilla? Mitä tuotteita käyttäjä katsoo? Muiden muassa näihin kysymyksiin haetaan vastauksia tietojenkeruun ja diagnosoinnin avulla. (Mitä on web-analytiikka? n.d.)

Monia web-analytiikkaan päteviä ohjenuoria ja ajatusmaailmoja voidaan hyödyntää myös pelidataa analysoitaessa. Poikkeuksetta on erityisen tärkeää tietää, kuinka usein ja miten pitkään käyttäjä viettää aikaa pelin ääressä. Näin saadaan tietoa mm. siitä, sijoittuuko pelaaminen johonkin tiettyyn vuorokauden aikaan tai viikonpäivään, minkä perusteella esimerkiksi mainoksia voisi kohdentaa tarkemmin. Luonnollisesti pelien yksilöllisiä tapahtumia ja tavoitteita pitää myös pystyä seuraamaan ja taltioimaan, että pelaamisen seurannasta saa kaiken mahdollisen hyödyn irti. Muutamat pelien analysointityökalut tarjoavatkin myös heatmap- eli lämpökartta-ominaisuuden, jonka avulla voidaan selvittää esimerkiksi, missä kohdassa kenttää pelaajat kuolevat tai liikkuvat useimmiten. (Drachen 2013.)

2.2 Olemassa olevat järjestelmät

2.2.1 Yleisesti analysointityökaluista

Luonnollisesti markkinoilta löytyy jo vastaavanlaisia tuotteita, kuin mitä tämän opinnäytetyön tuloksena syntyi. Näitä on lukuisia erilaisia, mutta kaikissa on kuitenkin sama periaate: integroida peliin erilaisten tietojen keräämiseen tarkoitettu järjestelmä, jonka

kautta tietoa säilötään tietokantaan. Saaduista tiedoista muodostetaan tarpeiden mukaan erilaisia yhteenvetoja ja vertailuja.

Tiedon keräämistä ja analysointia ei harjoiteta kuitenkaan pelkästään pelien saralla. Tunnetuin tällainen palvelu Internet-maailmassa lienee Googlen tarjoama Google Analytics -palvelu, jolla pystytään erittäin hyvin seuraamaan www-sivujen liikennettä. Palvelusta on saatu todella helppokäyttöinen ja perusominaisuuksiltaan monipuolinen. Käyttääkseen sitä ei tarvitse olla tietotekniikan ammattilainen, vaan jokainen palvelua halajava voi opastusta seuraamalla luoda tilin ja lisätä sivuilleen koodinpätkän, joka hoitaa käytännössä kaiken muun. Palvelusta löytyy asioiden analysoimiseen perusominaisuuksia, ja luonnollisesti maksua vastaan käyttöön saa enemmän ominaisuuksia ja rajoitukset löysenevät. Google Analytics on myös integroitavissa mobiiliin Android- ja iOS-sovelluksiin. (Google Analytics 2014.)

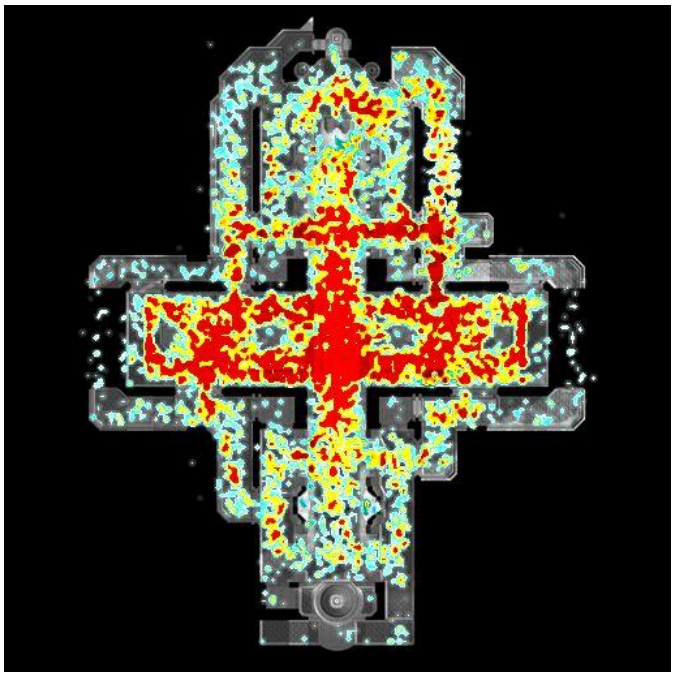
Hieman lähemmäs pelien analysointityökaluja osuvat Yahoo!:n omistama Flurry ja Applen iTunes Connect. Flurrystä löytyy tuki mm. iOS-, Android- ja Windows Phone-sovelluksille kun taas iTunes Connect on luotu pelkästään Applen App Storeen lisättyjen iOS-käyttöjärjestelmässä toimivien sovellusten seurantaan ja analysointiin. Molemmat ovat kuitenkin tarkoitettu vain ja ainoastaan mobiilisovellusten tarkkailuun, vaikkakin samoja käyttäjien tekemisiä niissäkin seurataan kuin vaikka www-sivujen analysoinnissa. Mobiilisovellusten seurannassa mukaan astuukin rahan liikkuminen ja mahdollisten virheraporttien kirjaaminen. (Flurry 2014; App Store (iOS) 2014.)

Yleisimmin analysoitavan tuotteen tilastot voidaan jakaa karkeasti kolmeen luokkaan: käyttäjän hankinta (acquisition), käyttäjän sitoutuminen (engagement) ja ns. rahoiksi lyömiseen (monetization). Opinnäytetyössä keskitytään käyttäjien sitoutumiseen paneutuviin tilastoihin, sillä ne ovat kriittisimmät tiedot, joita jokainen pelinjulkaisija haluaa tietää. Luvussa 6 perehdytään tarkemmin näihin osa-alueisiin sekä siihen, kuinka saatuja tilastoja voidaan konkreettisesti hyödyntää niin yritystoiminnassa kuin käyttäjien kokemuksien osalta.

2.2.2 Pelien analysointityökalut

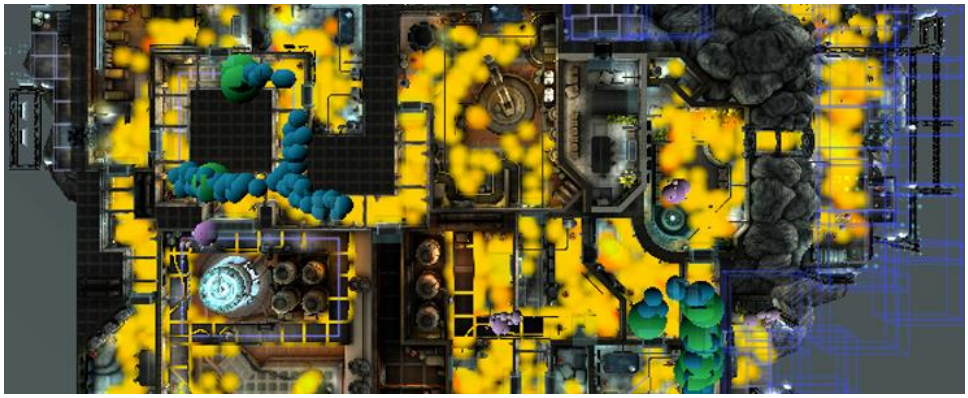
Kovin radikaalisti näistä edellä mainituista mobiilisovellusten analysointijärjestelmistä eivät varta vasten pelien analysointiin tehdyt ohjelmistot enää poikkeakaan. Näitäkin tuotteita löytyy lukuisia, joista valtaosa kokonaan maksullisia ja vain muutama ilmainen, mutta maksullisen premium-jäsenyyden omaava. Vähäisestä joukosta löytyy silti Unity Technologiesin GameAnalytics, joka on oikein vakuuttava, monipuolinen ja täysin ilmainen järjestelmä pelien analysoimiseen. Kuten voikin arvata, tältä Unityn omistamalta yhtiöltä löytyy varta vasten Unitylle räätälöity versio. Muita geneerisempiä alustavaihtoehtoja ovat mm. iOS, Android ja Flash.

Unity-versiossa on saatavilla luvussa 2.1 mainittu heatmap-ominaisuus, jota käytetään pääasiassa selvittämään, missä pelaajat kuolevat eniten. Tämä ei kuitenkaan ole ainoa käyttötarkoitus, sillä mm. GameAnalyticsin heatmapia voi mukauttaa loputtomasti vastaamaan omia tarpeita. Heatmapien avulla voidaan myös optimoida kenttiä vastaamaan haluttua vaatimustasoa. Voidaan selvittää, mihin pelaajat kiinnittävät kentissä huomiota ja kuinka helposti pääsevät etenemään pelissä. Kuviosta 1 nähdään yhden pelin kentän kuolleisuutta kuvaava heatmap, jossa tummemmaksi muuttuva väri tarkoittaa enemmän kuolemia. (Drachen 2013.)



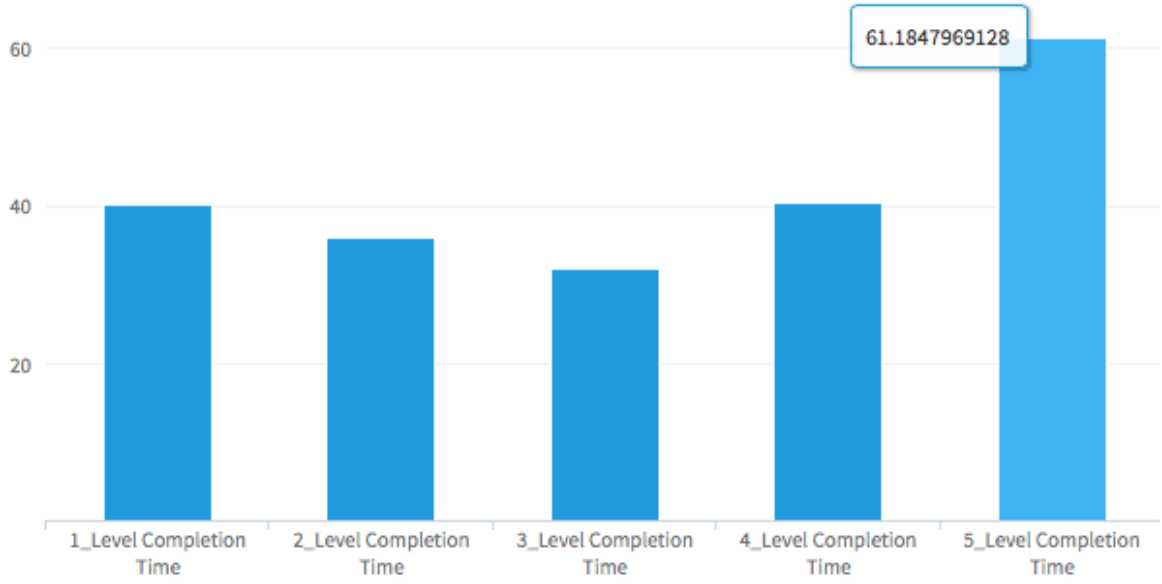
Kuvio 1. Kuolemien heatmap Halo 3 -pelistä (Drachen 2013)

Kuviosta 2 käy ilmi Unityssä käytössä ollut heatmap, joka kuvastaa pelaajan liikehdintää kentässä.



Kuvio 2. Liikkumista kuvaava heatmap (Drachen 2013)

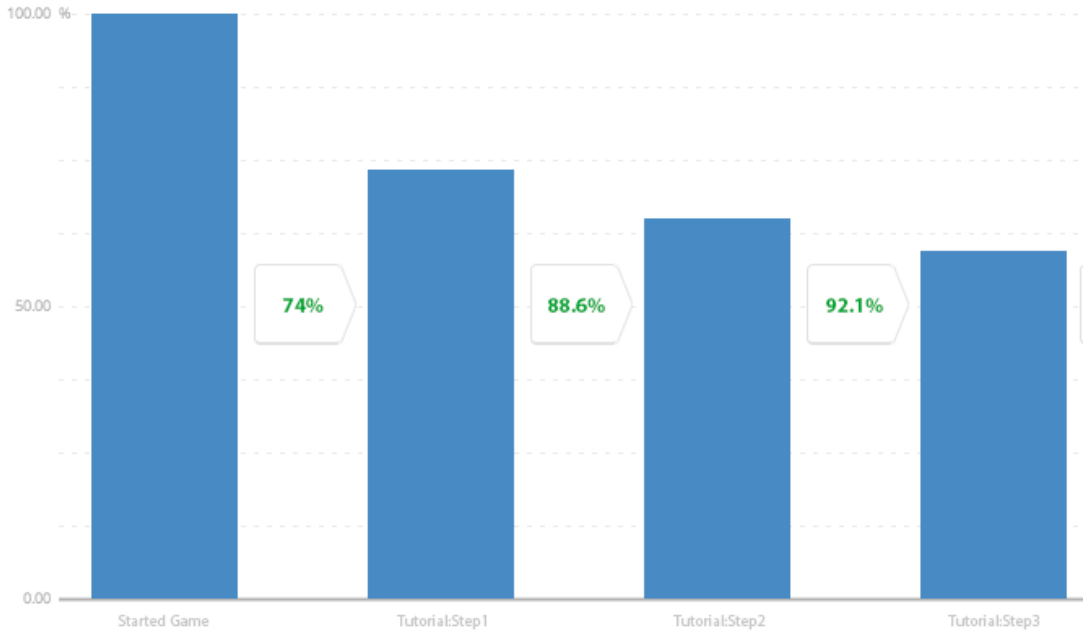
Toinen ilmainen ja helposti testattavaksi saatava pelien analysointityökalu on Rebel Hippo -yrityksen Lumos, joka on kuitenkin melko pelkistetyn oloinen GameAnalyticsin käyttämisen jälkeen. Lumos on ilmainen vain 100 pelaajaan asti, ja siitä puuttuu heatmap-ominaisuus. Järjestelmä on kuitenkin helppokäyttöinen ja tarjoaa GameAnalyticsin tapaan mahdollisuuden luoda mukautettuja tapahtumien seurantakaavioita. Kuviosta 3 nähdään esimerkki, kuinka voidaan kerätä ja visualisoida pelin kenttien läpäisyyn käytetty keskimääräinen aika pelaajien kesken. (Overview 2014.)



Kuvio 3. Keskimääräinen kenttien läpipeluuaika Lumoksella seurattuna (Overview 2014)

Pelinkehittäjiä luultavasti kaikkein eniten kiinnostava asia on uusien käyttäjien päivittäinen saanti. Tämä ja DAU (daily active users, päivittäiset aktiiviset käyttäjät) ovat melkeinpä jopa pakollisia ominaisuuksia, jotka tulisi jokaisesta analysointityökalusta löytyä. Kun DAU:ssa tarkkaillaan päivittäisiä käyttäjiä, niin vastaavasti MAU:ssa (monthly active users) summataan kuukauden aikana olleet aktiiviset käyttäjät yhteen. Luonnollisesti myös peleissä tehtyjä ostotapahtumia halutaan seurata, jos tällaiselle on tarvetta. Näiden lisäksi erittäin haluttua on kerätä pelissä tapahtuneita virheitä ja muita varoituksia.

Yksi erityisen hyödyllinen ominaisuus missä tahansa analysointijärjestelmässä ovat funnelit. Ne ovat käytännössä eräänlaisia suppiloita tai suodattimia, joilla luodaan ns. polkuja pelin tai muun palvelun läpi. Jos esimerkiksi halutaan tietää, kuinka moni pelaaja avaa pelin tutoriaalin ja lukee sen loppuun saakka, voidaan tutoriaalin eri vaiheet syöttää järjestelmään ja näin seurata käyttäjien liikehdintää, kuten kuviosta 4 nähdään. Funneleiden avulla voidaan selvittää myös, missä vaiheessa pelaajat jättävät tutoriaalin kesken tämän avulla koittaa päätellä, missä vika on. Onko käyttöliittymä liian vaikeasti ymmärrettävä vai ovatko tutoriaalin ensimmäisen kohdan opastukset niin perinpohjaisia, että pelaaja päättelee osaavansa kaiken oleellisen ja tuomitsee tutoriaalin turhaksi? (Taras 2014.)



Kuvio 4. Tutoriaalin funnelin tulokset visualisoituna

Retentionilla seurataan, kuinka hyvin asiakkaat ovat pysyneet uskollisina ja jatkaneet esimerkiksi pelin pelaamista. Tämän laskemiseksi on olemassa lukuisia kaavoja eri tuote- tai palvelutyypeille. Pelin käyttäjien koukuttamislukemia saadaan selville varsin yksinkertaisella kaavalla, mutta erilaisia laskutapoja on useita. Tyypillisesti halutaan tietää 1:n, 7:n, 14:n ja 28:n päivän retentionit. Ensin tarvitaan tieto siitä, ketkä käyttäjät ovat asentaneet pelin. Tämän jälkeen seuraavasta päivästä eteenpäin aina haluttuun päivämäärään asti lasketaan kaikki käyttäjät, jotka olivat rekisteröityneet sinä tiettynä päivänä. Seuraavaksi tämä uudelleen kirjautuneiden määrä jaetaan silloisella rekisteröityneiden määrällä ja kertomalla saadun luvun saadaan selville se prosentuaalinen luku käyttäjistä, ketkä palasivat pelin pariin. Termiä käytetään muuallakin työelämässä liittyen työntekijöiden tyytyväisenä pitämiseen. Työnantajalle tulee selvästi kalliimmaksi menettää tyytymätön työntekijä ja kouluttaa tilalle uusi, kuin yrittää korjata epäkohdat ja vääryydet ja näin ollen pitää työntekijä tyytyväisenä. Loppujen lopuksi tämä ei poikkea paljoa pelaajien koukuttamisesta, pyritäänhän molemmissa estämään kohteen lähtö. Luvussa 4.4.3 käydään läpi, kuinka työssä toteutettiin retentionin laskeminen. (Sommer 2014; Retention Analysis n.d.)

GameAnalytics ja Lumos ovat molemmat oikein monipuolisia ja niihin tutustumalla ja käyttämällä pääsee hyvin alkuun ja selville, mitä kaikkea käyttäjistä voikaan saada irti. Muita maksullisia järjestelmiä on useita, joista mm. HoneyTracks näyttää oikein lupaavalta, ja siitä on tarjolla myös karsittu ilmaisversio.

Tuotteet eivät kuitenkaan vastanneet yrityksen tarpeita tai jopa jossain määrin ylittivät ne monimutkaisuudellaan. Työn toteutuksessa lähdettiin erityisesti tavoittelemaan tuotetta, jota olisi suhteellisen helppo muokata aivan omanlaiseksi vastaamaan omia tarpeita. Mainitut tuotteet tarjoavatkin toki laajat mahdollisuudet mm. mukautettujen hakujen luomiseen ja kaavioiden muodostamiseen, mutta tuotteet eivät ole reaaliaikaisia. Tämä olikin suurin syy alkaa työstämään omaa järjestelmää, jossa tilastot muodostettaisiin reaaliaikaisena eikä päivää myöhemmin.

3 Käytetyt teknologiat

3.1 MongoDB

3.1.1 Yleistä

Järjestelmän tietokannaksi valittiin NoSQL-tietokanta nimeltään MongoDB (”Mongo”) tämän joustavuuden ja skaalautuvuuden takia. Mongo ei seuraa mitään määrättyä taulukkoskeemaa toisin kuin SQL-pohjaiset relaatiotietokannat, kuten MySQL. Aivan upouusi Mongo ei ole, sillä tämän avoimen lähdekoodin tietokanta juontaa juurensa vuoteen 2007. Mongo on tällä hetkellä suosituin NoSQL-tietokantajärjestelmä, eli siinä tieto säilötään nk. avain-arvo-pareina poiketen relaatiomallia noudattavista tietokannoista. (Hows, Plugge, Membrey & Hawkings 2013, 3.)

Aikaisemmin muiden tietokantojen kanssa tekemisissä olleet voivatkin vähän säikähtää kun kuulevat, mitä ”puutteita” ja muita eroavaisuuksia Mongossa on verrattuna vaikkapa tuttuun ja turvalliseen MySQL:ään. Tämän ollessa se kaikista tunnetuin tietokantavaihtoehto viitataan ja verrataan Mongoakin jatkossa suoraan tai epäsuorasti siihen. Jo terminologiassa tulee vastaan eroavaisuuksia, sillä MySQL:n taulu (table) tarkoittaa käytännössä samaa kuin Mongon kokoelma (collection) ja rivi (row) on Mongossa dokumentti (document).

3.1.2 Perusteet

Kuten mainittu, Mongossa tieto säilötään ihmissilmälle suhteellisen helppolukuisina avain-arvo-pareina aivan kuin laajalti käytössä olevien JSON-tiedostojen tapaan. Mongoon siis käytännössä talletetaan JSON-objekteja. Ihan näin yksinkertaisesti tietoja ei kuitenkaan kantaan tallenneta, vaan niistä muodostetaan BSON-objekteja, eli binäärimuotoisia JSON-objekteja. Tämä tarkoittaa käytännössä sitä, että määritellyt arvot voivat olla tyypiltänsä jotain muutakin kuin perinteiset merkkijono (string), totuusarvomuuttuja (boolean) ja numero (number). Näistä esimerkkeinä päivämäärä (Date) ja ObjectId. Yhden BSON-dokumentin eli ns. tietueen enimmäiskoko on 16 MB, millä estetään keskusmuistin ylikuormitus. (Documents 2014.)

MySQL:ssä taulut määritellään tarkasti niitä luotaessa, mutta Mongossa kenttiä ei luoda etukäteen, mikä tarjoaa paitsi joustavan ja selkeän myös helpon tavan täydentää dokumentteja lennossa. Jokaisessa dokumentissa täytyy olla yksilöllinen _id-kenttä. Tämän puuttuessa tietokanta generoi itse ObjectId-tyyppisen merkkisarjan. (Mt.)

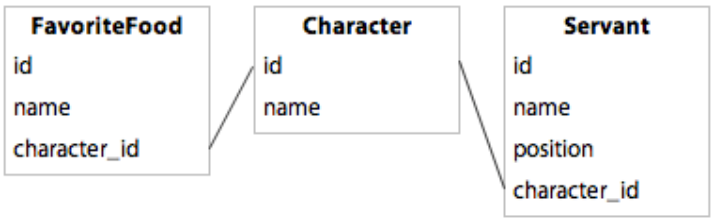
Suurin eroavaisuus relaatiokantoihin Mongossa ovat sisäkkäiset dokumentit. Tämä käytännössä tarkoittaa sitä, että yhden dokumentin sisällä voidaan säilöä nk. lapsidokumentteja, jotka voivat sisältää mitä tahansa tietoa kuten vanhempansakin. Näiden lisäksi myös taulukoiden sisällyttäminen yhden kentän tiedoiksi on mahdollista. Relaatiokannoissa vastaavanlainen rakenne ratkaistaisiin luomalla jokaista lapsidokumenttityyppiä tai taulukon solua vastaava taulu, josta viitataan jollain avaimella vanhempaan. (Mt.)

Seuraavaksi esimerkki sisäkkäisiä dokumentin ja taulukon omaavasta dokumentista Mongossa:

```
{
    _id: 1337,
    name: "Count Duckula",
    favorite_food: [
        "broccoli", "sandwiches"
    ],
    servants: [
        { name: "Nanny", position: "nanny"  },
        { name: "Igor",  position: "butler" }
    ]
}
```

Kuviosta 5 nähdään, miltä ylemmän esimerkin rakenne lapsidokumentteineen näyttää visualisoituna ja kuviossa 6 sama tietokanta relaatiotietokantojen muodossa. Tässä Characterin tiedot koostuvat kolmesta taulusta, jotka yhdistetään toisiinsa Character-taulun id-kentän avulla. Jokainen lapsidokumentti on omana tietueenaan omassa taulussaan, johon on myös lisätty kenttä, josta löytää oikean Character-dokumentin.

Kuvio 5. Esimerkkitietokannan rakenne



Kuvio 6. Esimerkkitietokannan rakenne relaatiomallilla

Mongossa ei tarvitse tehdä mitään erikoista päästäkseen käsiksi `favorite_food`-kohdan tietoihin, kun taas relaatiomallisessa kannassa täytyisi käyttää hyödyksi erilaisia JOIN-operaatioita hakeakseen eri taulusta tarvittavia tietoja. Ei kuitenkaan saa sokaistua sisäkkäisten dokumenttien upeudelle, sillä rakennetta suunniteltaessa täytyy ottaa huomioon dokumentin 16 MB:n kokorajoitus. Jos dokumentista tulee valtavan iso, täytyy rakennetta pilkkoa useampaan kokoelmaan, jotka sitten yhdistetään relaatiomallin tavalla jollain avaimella. Nämä haut on kuitenkin toteutettava ohjelmakoodillisesti, eli Mongo sysää vastuuta ohjelmoijalle enemmän toisin kuin relaatiomallia noudattavat serkkunsa. (MongoDB Limits and Tresholds 2014.)

3.1.3 Operaatiot

Insert

Mongossa operaatioiden käyttö on suhteellisen yksinkertaista. Käsky muodostuu kokoelman nimestä sekä toiminnosta ja dokumentista, jota muokataan tai lisätään. Seuraavassa esimerkissä character-kokoelmaan lisätään dokumentti, jolla on `_id` ja `name`.

```
db.character.insert({
    _id: 1337,
    name: "Count Duckula"
})
```

Update ja upsert

Olemassa olevaa dokumenttia voidaan päivittää `update`-operaatiolla, jolloin annetaan ensin dokumentin määrittelevä ja tavallaan hakuehtona toimiva dokumentti jota päivitetään. Seuraavassa esimerkissä haetaan muokattava dokumentti `_id`:n perusteella ja käytetään `$push`-operaattoria lisäämään `favorite_food`-kenttään uusi alkio. Vaikka kenttää ei alun perin ole olemassa, Mongo luo sen automaattisesti, koska se on joustava.

Ilman operaattorin käyttöä dokumentti korvattaisiin annetulla dokumentilla, mutta $push-operaattorilla sekä monilla muilla ($set, $pop, $pull, $addToSet …) voidaan muokata vain osaa dokumentista ilman vanhan poispyyhkimistä.

```
db.character.update({
    _id: 1337
},
{
    $push: {
        favorite_food: "broccoli"
    }
})
```

Update siis päivittää dokumentin vain kun se on olemassa, mutta upsert-operaatiolla voidaan lisätä haluttu dokumentti annetuilla tiedoilla, jos sitä ei ennalta ole olemassa. Tämä on oikein elegantti tapa päivittää ja lisätä dokumentteja. Aiemman operaation update-sana vain korvataan upsert-sanalla ja muita muutoksia ei tarvita.

Remove

Myöskin dokumenttien poistaminen käy yksinkertaisesti. Operaatioon sisällytetään hakudokumentti, jonka avulla löytyvä dokumentti poistetaan. Seuraava operaatio poistaa kaikki dokumentit, joiden favorite_food-kentässä olevasta taulusta löytyy ”broccoli”. Lausekkeen voi myös määritellä poistamaan kaikki muut, joiden favorite_food-taulussa ei ole listattuna broccolia.

```
db.character.remove({
    favorite_food: {
        $in : ["broccoli"]
    }
})
```

Aggregaatio

Aggregaatioiden avulla voidaan muodostaa entistä monimutkaisempia hakuja kantaan. Tässä operaatiot järjestetään peräkkäin pipelineen eli eräänlaiseen komentoputkeen. Operaattoreilla voi mm. muokata kenttien nimiä, purkaa tauluja, järjestää kentän mukaan, suodattaa dokumentteja pois jonkin kentän arvon mukaan ja ryhmitellä. Myös aggregaatioissa voidaan ja toisaalta pitääkin käyttää aikaisemmin mainittuja operaattoreita, sillä nämä helpottavat todella paljon tiedon suodattamista ja uudelleen muotoilua.

Seuraavassa esimerkissä puretaan ensin favorite_food-kentän tiedot ja ryhmitellään ottamalla jokaiselta characterilta $addToSet-operaattorin avulla vain yhden kerran kyseinen taulukossa oleva ruoka. Sitten taulukko puretaan jälleen ja kaikki dokumentit ryhmitellään näiden ruokien nimien mukaan. Count-kenttään luodaan $sum-operaattorilla laskuri, jota nostetaan aina sellaisen dokumentin löytyessä, jonka favorite_food-taulukosta löytyy kyseinen ruoka. $project-operaattoriin päästyä _id-kentässä on ruoan nimi, mutta tämä sijoitetaan uuteen food-nimiseen ”kenttään” ja alkuperäinen _id piilotetaan. Count näytettään ilmoittamalla tämän arvoksi 1, vaikka tämä näytettäisiin muutenkin, mutta tämä selkeyttää aggregaatioden muodostamista.

```
db.character.aggregate([
{ $unwind: "$favorite_food" },
{
    $group: {
        "_id": "$_id",
        "foods": { $addToSet: "$favorite_food" }
    }
},
{ $unwind: "$foods" },
{
    $group: {
        "_id": "$foods",
        "count": { $sum: 1 }
    }
},
{
    $project: {
        "_id": 0,
        "food": "_id",
        "count": 1
    }
}])
```

Kuvitteellisilla dokumenteilla varustettuun kokoelmaan tehdyn haun tulostuksesta pystyy nyt näkemään kuinka monella hahmolla oli lempiruoaksi merkitty mitäkin. Tulostus voisi olla esimerkiksi seuraavanlainen:

```
{ "count" : 1, "food" : "apple" }
{ "count" : 2, "food" : "broccoli" }
{ "count" : 2, "food" : "bacon" }
{ "count" : 3, "food" : "banana" }
```

3.1.4 Ylläpito

Mongo-tietokannan ylläpitoon liittyvät kiemurat eivät kuitenkaan ole aivan niin yksinkertaiset kuin mitä seuraavaksi käydään. Luonnollisesti jokainen seuraavaksi mainituista osa-alueista on paljon laajempi kuin mitä käydyt perusasiat antavat ymmärtää. Näin asioiden perusteellinen läpikäyminen vaatisi huomattavan määrän lisäsivuja ja rautaista ammattitaitoa ymmärtääkseen jokaisen ominaisuuden toiminnan. Opinnäytetyön toteutuksessa ei panostettu lainkaan tietokannan ylläpitoon liittyviin asioihin eli seuraavaksi mainittaviin osa-alueisiin. Nämä ovat kuitenkin oleellisia huomioitavia asioita Mongo-tietokannan ylläpitosuunnitelmaa laadittaessa, joten olisi todellinen vääryys jättää näiden mainitseminen.

Transaktiot

Mongossa ei ole muista SQL-tietokannoista tutuksi tulleita transaktioneita, joilla tarkoitetaan useamman taulun muokkaamista yhden komennon avulla. Syy näiden puuttumiseen on dokumenttien rakenne, sillä lapsidokumenteilla yleensä korvataan muiden taulujen kanssa tehtävät liitokset. Mongo ei siis noudata ACID-periaatetta (atomicity, consistency, isolation, durability), jonka avulla turvataan järjestelmän tietojen eheys kaikissa tilanteissa. (Perform Two Phase Commits 2014.)

Jokainen Mongossa yksittäiseen dokumenttiin tehtävä operaatio on kuitenkin atominen, eli muutokset suoritetaan joko kokonaan tai ei lainkaan. Sama pätee eheyteen, eli jos jokin vakava asia (esim. Internetkatko) keskeyttää operaation suorittamisen, tietokanta palautuu edelliseen vakaaseen tilaan. Useamman dokumentin muokkaaminen transaktoimaisesti (multi-document transactions) onnistuu, mutta tässäkin tapauksessa ohjelmoijalle sysätään vastuu. (Mt.)

Indeksointi

Indeksoimalla Mongo-tietokannasta saa tehokkaamman ja nopeamman. Ilman tällaista toimenpidettä Mongon täytyisi käydä läpi joka ikinen dokumentti kokoelman sisältä etsiessään hakua vastaavaa/vastaavia dokumentteja. Yleensä joka dokumentista löytyy ainakin 1 kenttä, jonka käyttö eri operaatioissa toistuu useammin kuin muiden. Seuraavassa esimerkissä indeksoidaan character-kokoelmasta löytyvien dokumenttien `_id`-

kenttä nousevaan järjestykseen, ja se onkin yksi yleisimmistä indeksoinnin kohteista. `-1` tarkoittaisi laskevaa järjestystä. (Hows ym. 2013, 249 - 253.)

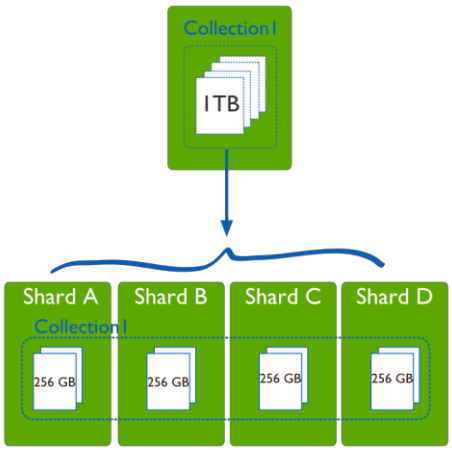
```
db.character.ensureIndex({
    "_id" : 1
})
```

Jatkossa jokainen haku, jossa käytetään `_id`-kenttää hakudokumentissa, toimii nopeammin. Indeksoinnin poistaminen käy myös helposti korvaamalla edellisen komennon `ensureIndex` operaattorilla `dropIndex`. (Mts. 254.)

Indeksoimalla voidaan siis laskea merkittävästi hakuihin käytettyä aikaa, joskus puhutaan jopa sadasosiin pääsystä. Indeksoiminen vie kuitenkin myös suhteellisen paljon aikaa, mutta on usein kannattavaa. (Mts. 248.)

Sharding

Kuten järjestäen kaikissa monimutkaisissa järjestelmistä, löytyy Mongostakin osa-alueita, joita ei pidetä jokaisen peruskäyttäjän osaavan. Kun tiedon määrä tietokannassa kasvaa isoihin mittoihin eivätkä yhden palvelimen laskentatehot ja muisti riitä käsittelemään kaikkea tulevaa liikennettä, joudutaan turvautumaan joko shardingiin tai palvelimen päivittämiseen. Shardingilla ja shardauksella tarkoitetaan tietokannan kokoelmien jakamista useammalle laitteelle. Kuviossa 7 on esimerkki, kuinka 1 TB kokoinen kokoelma voitaisiin jakaa osiin. (Sharding Introduction 2014.)



Kuvio 7. Kokoelman shardaus (Sharding Introduction 2014)

Jotta kokoelman pystyisi shardaamaan parhaalla mahdollisella tavalla, täytyisi siitä löytyä tarpeeksi yksilöllinen kenttä shard keyksi, jonka perusteella dokumentit jaettaisiin, mutta joka varmasti löytyy jokaisesta dokumentista. Esimerkiksi henkilötietoja sisältävän kokoelman dokumentit voisi lajitella vaikka syntymäkuukauden tai jopa puhelinnumeron mukaan. Huono shard key voisi olla vaikka lempiväri, sillä silloin eroavaisuuksia ei oisi kovin paljoa. Shard keytä ei pysty muuttamaan noin vain kun sen on kerran valinnut, joten täytyy miettiä tarkkaan, mikä olisi kaikista paras ratkaisu. (Mt.)

Replication

Estääkseen tietojen katoamisen vaikkapa sähkökatkoksen aika, joudutaan käyttämään tietokantojen replikoimista eli toisin sanoen kopioimaan tietokantaa eri palvelimelle muodostaen näin replica set -nimisen tietokantapalvelinperheen. Tyypillisimmin replica setteja on kolme, ja ne koostuvat yhdestä primary-tason kannasta, johon kohdistetaan kaikki haut ja lisäykset, joita kantaan tehdään. Tämän lisäksi joukkoon kuuluu myös korkeintaan secondary-tasoisia kantoja, joista yksi ”äänestetään” primaryksi. Replica settiin voi myös liittää yhden arbinter-palvelimen, jossa ei säilötä tietokantaa. Tämä osallistuu vain uuden primaryn päättämiseen. Tällä tavalla voidaan estää kahden primaryn syntymistä. (Hows ym. 2013, 259 - 262.)

Primaryyn tehdyt muutokset talletetaan myös oplog-nimiseen (operation log) kokoelmaan, josta secondary-arvoiset tietokannat kopioivat muutokset itseensä. Jos secondaryt kopioisivat tietonsa suoraan primarystä ja primary menisi syystä tai toisesta alas, kuinka kävisi tietojen, joita secondaryt eivät saaneet kopioitua ja jostain secondarystä tulisi nyt primary? Tiedon häviämistähän siitä seuraisi. Secondaryt kuitenkin pyrkivät pitämään itsensä melkein reaaliajassa päivitettyinä. (Mts. 263.)

3.2 Go

3.2.1 Yleistä

Go, hakukoneystävällisemmin golang, on Googlen kehittämä nuori ohjelmointikieli. Tämä vuonna 2009 päivän valon nähnyt kieli sai alkunsa kolmen kehittäjäkonkarin tyyty-

mättömyydestä nykyiseen tarjontaan ja näin ollen lähtivät kehittämään nykyaikaista ja kaikin puolin täydellistä kieltä. Go:ta on kuvattu Pythonin ja C:n sekoitukseksi, sillä tästä löytyy yhtäläisyyksiä molempiin vanhoihin kieliin. Go:ta kehittäessä on pyritty panostamaan nopeaan ohjelmien kääntämiseen, suorituskyvyn tehokkuuteen ja ohjelmoinnin helppouden yhdistämiseen, joihin ei tekijöiden mukaan mikään kilpailevista kielistä ole täysin pystynyt. (Kuosmanen 2009.)

Opinnäytetyön toteutusteknologioita suunniteltaessa Go:n kilpailijaksi loppusuoralle selvisi varta vasten palvelinpuolen ohjelmointiin suunniteltu Javascript-pohjainen Node.js (”Node”). Tämä olisi ollut hyvinkin luonnollinen valinta täydentämään MongoDB-tietokantaa sekä niinikään Googlen kehittämää AngularJS-ohjelmistokehystä, sillä näiden teknologioiden yhteiskäytöstä löytyy lukuisia esimerkkejä ja tutoriaaleja. Noden heikkous on kuitenkin itse Javascript, koska tämä on heikosti tyypitetty ja sillä on helppo tehdä huonoa koodia ja sortua ratkaisuihin, jotka eivät pitkällä aikavälillä ole kantavia. Erääksi Noden suureksi heikkoudeksi kuuleekin useiden mainittavan callbackit, eli eräänlaiset metodit, joita kutsutaan kun joku toinen metodi on suoritettu. Näitä käytettäessä joutuu helposti ikävään koodisotkuun, eikä pakotietä näy. Tällaista toimintaa kuitenkin tarvitaan välillä, esimerkiksi ladatessa tiedostoa tai otettaessa yhteyttä tietokantaan. Go:sta löytyy kuitenkin `goroutine`-niminen toiminto, joka kertoo kutsuttavalle funktiolle, että tämän täytyy toimia asynkronisesti eli muun koodin suorittaminen jatkuu normaalisti eikä tätä jäädä odottelemaan. (Go vs Node.js for servers 2014.)

Go:n valintaan vaikutti tietenkin myöskin se, että tämän laskentanopeus ja yleinen tehokkuus on kielten kärkipäätä. Nuoresta iästään johtuen Go ei ole vielä voittanut vannoutuneita Python-koodareita puolelleen, eikä tästä löydy yhtä paljon opiskelumateriaalia kuin vanhemmista kielistä, mikä hankaloittaa aloittelevaa Go-koodaria. (Mt.)

3.2.2 Perusteet

Jokainen Go:lla tehty ohjelma omaa `main`-nimellä kulkevan paketin (package), josta aloitetaan ohjelman suorittaminen. Kuten monissa muissakin ohjelmointikielissä, luetellaan Go:ssakin tiedoston alussa ohjelmassa paketit `import`-komennolla. Go on vahvasti tyypi-

tetty staattinen kieli, mikä tarkoittaa sitä, ettei muuttujien tyyppejä pysty ohjelman ajon aikana enää muuttamaan toisin kuin dynaamisilla kielillä ohjelmoitaessa.

Yksi Go:n parhaimmista puolista on sisäänrakennettu gofmt-työkalu, joka automaattisesti muotoilee ohjelmakoodin standardin mukaiseksi. Tämä tekee koodista paitsi helpommin luettavaa, kirjoitettavaa ja hallinnoitavaa niin myös poistaa turhan väittelyn ohjelmoijien välillä liittyen sisennysten ja sulkeiden määriin ja sijainteihin. Go:ssa ei myöskään tarvita puolipisteitä rivien lopussa tai sulkuja ehto- ja toistolausekkeissa. Monista muista ohjelmointikielistä poiketen Go:sta puuttuu `while`-toistolauseke, sillä tämän korvaa oikein muodostettu `for`-lauseke. Go:ssa pystyy myös palauttamaan yhden kuin useamman muuttujan metodista. Monissa muissa kielissä tämä on ratkaistu sijoittamalla tarvittavat muuttujat taulukkoon ja palauttamalla sen, lisäten näin ylimääräisiä vaiheita muuttujiin käsiksi pääsemiseen.

Seuraava esimerkki on hyvin yksinkertainen versio toimivasta Go-ohjelmasta. Vastoin monien muiden ohjelmointikielien syntaksia Go:ssa metodeihin vietävien muuttujien tyypit ilmoitetaan muuttujan nimen jälkeen. Esimerkistä nähdään myös, kuinka muuttujia pystytään ”ketjuttamaan” ja luomaan eri tavoin.

```
package main
import "fmt"

func split(sum int) (x, y int) {
	x = sum * 6 / 9
	y = sum - x
	return
}

func main() {
	var str string
	str = "Good night out there, whatever you are."
	a, b := 21, 2
	fmt.Println(str)
	fmt.Println(split(a * b))
}
```

Ajamalla komentokehotteessa komennon `go run [tiedosto]` ohjelma ajetaan heti. `go build [tiedosto]` loisi sovelluksesta ajettavan tiedoston. Ohjelmassa on määritelty tulostettavaksi `fmt`-pakettia hyödyntäen merkkijono sekä metodista palautettavat numeromuuttujat.

```
# komentokehote
$ go run test.go
Good night out there, whatever you are.
28 14
```

Ennen kuin ohjelma voidaan suorittaa, ajetaan se automaattisesti kääntäjän läpi. Kääntäjä käy koodin läpi ja tarkastaa, löytyykö koodista esimerkiksi tyyppivirheitä, puuttuvia sulkeita tai mahdollisesti käyttämättömiä tai määrittelemättömiä muuttujia tai paketteja. Jos ohjelmaa buildatessa tai ajettaessa kääntäjä löytää virheitä, niistä kerrotaan komentokehotteella eikä ohjelmaa suoriteta.

Muiden ohjelmointikielten tavoin Go:stakin löytyy paljon hienouksia eri osa-alueilta, joihin voisi perehtyä loputtomiin. Opinnäytetyön kannalta tärkeää osaa näytteli kuitenkin structit, jotka ajavat saman asian kuin muissa kielissä käytetyt luokat ja on sama tärkeä osa Go:n oliomaisuutta. Siinä missä luokkien sisälle määritellään tähän luokkaan liittyvät metodit, määritellään ne Go:ssa ulkopuolelle. Metodien kylkeen vain määritellään, että tämä on vain tietyn structin käytössä. (Sathish VJ. 2011.)

Seuraavassa esimerkkikoodissa luodaan ja tulostetaan eri tavoin struct-tyyppisiä muuttujia. Alussa määritellään uusi struct tyyppi nimeltään VHS ja tälle kaksi ominaisuutta. Tämän jälkeen koodista löytyy molemmille ominaisuuksille yksinkertaiset getterit eli arvon palauttajat. Huomioitavaa on, kuinka `func`-sanan jälkeen on määritelty, mille structille metodi on tarkoitettu. Pääohjelmassa luodaan oliot kolmella hyväksyttävällä tavalla. Jos jotakin arvo ei anneta, tulee sen kohdalle oletusarvo eli numerokenttiin 0 jne. Tulostus vaiheessa ensimmäinen olio tulostetaan viittaamalla suoraan muuttujan ominaisuuksiin pistenotaatiolla, toisessa vaiheessa käytetään hyödyksi structia varten luotuja metodeja ja kolmas tulostus tulostaa olion raakana.

```
package main

import "fmt"

type VHS struct {
	year int
	Name string
}

func (tape VHS) getYear() int {
	return tape.year
}
```

```
func (tape VHS) getName() string {
	return tape.Name
}

func main() {
	a := VHS{}
	a.year = 1989
	a.Name = "The Vampire Strikes Back!"

	b := VHS{
		year: 1990,
		Name: "A Fright at the Opera",
	}

	c := VHS{1990, "The Great Ducktective"}

	fmt.Printf("%s was released in %d\n", a.Name, a.year)
	fmt.Printf("%s was released in %d\n", b.getName(), b.getYear())
	fmt.Println(c)
}

# komentokehote
$ go run struct.go
The Vampire Strikes Back! was released in 1989
A Fright at the Opera was released in 1990
{1990 The Great Ducktective}
```

Perinteisessä olio-ohjelmoinnissa on totuttu paitsi määrittelemään olioita luokkienomaisesti niin myös rajoittamaan luokkien ja näiden ominaisuuksien näkymistä ulospäin erilaisin ennalta määrätyin sanoin. Tällaisia sanoja ovat muun muassa private (yksityinen), public (julkinen) ja protected (suojattu). Go:ssa ei näitä perinteisen luokkamallin lisäksi löydy, vaan struct-tyyppisen muuttujan ja tämän ominaisuuksien näkyvyyttä paketin ulkopuolelle säädellään pelkästään nimen alkukirjaimen perusteella. Iso alkukirjan tarkoittaa julkista ja pieni taas puolestaan yksityistä, joka on vain siis käytössä nykyisessä paketissa. (Mt.)

3.3 AngularJS

3.3.1 Yleistä

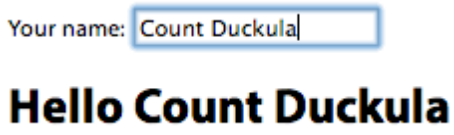
Järjestelmän asiakaspuolen toteutuksessa käytettiin Googlen kehittämää Javascript-pohjaista AngularJS-ohjelmistokehystä (”Angular”). Valinta oli helppo tehdä, sillä Angularilta löytyy laajat taustajoukot jo pelkästään Googlen puolesta, puhumattakaan haltioituneista puolueettomista alan ammattilaisista ympäri maailman. Angularin vahvuuksiin

lukeutuvat MVC-arkkitehtuurin (Model-View-Controller) noudattamisen helppous. Tällä tarkoitetaan käyttöliittymän, tietokannan ja käsittelijän erottamista toisistaan, jotta näiden eri osa-alueiden kehittäminen olisi selkeämpää. Näin esimerkiksi yhdellä sovelluksen toiminnallisuudella voi olla useampia käyttöliittymiä. (AngularJS 2014.)

Suosion salat piilevät myös kahdensuuntaisessa tiedon sitomisessa (two-way data binding), joka tarkoittaa käytännössä sitä, että kun muuttujaan sidottu tieto muuttuu, päivittyy se automaattisesti myös näkymään (Mt.). Alla erittäin yksinkertainen esimerkki Angularin toiminnasta kahdensuuntaisen tiedon sidonnan kanssa.

```
Your name: <input type="text" ng-model="name"/>
<h1>Hello {{name}}</h1>
```

Kuviosta 8 nähdään, mitä ylempi koodin pätkä saa aikaan. Kaikessa yksinkertaisuudessaan sovellukseen syötetään nimi sille varattuun tekstikenttään. Nimi päivittyy automaattisesti kirjain kirjaimelta HTML-elementtiin sidottuun muuttujaan. Tässä esimerkissä kriittisessä osassa toimii `ngModel`-direktiivi, jonka puuttuminen luonnollisesti tekisi ohjelmasta hyödyttömän. Sama toimii myös toisin päin: tekstikenttään päivittyy uusi arvo, jos muuttujaa muutetaan koodillisesti. (Directives 2014.)



Kuvio 8. Yksinkertaisen AngularJS-esimerkin lopputulos

Angular voidaan siis raa'asti jakaa muutamaan osa-alueeseen: näkymät, kontrollerit, direktiivit ja palvelut. Näistä kaksi ensimmäistä ovat käytännössä välttämättömät sovelluksen luomisessa Angularin avulla. Kuviossa 8 nähtyyn lopputulokseenkin tarvittiin nimenomaan vain näkymää ja kontrolleria.

jQuery-kirjastoa aiemmin käyttäneiden on luultavasti hankala hahmottaa, kuinka ohjelmointi toimii Angularilla. Monet sortuvatkin käyttämään jQueryä projekteissaan mm. DOM:n manipulointiin, koska tämä on helppoa ja lähestyy asiaa paljon yksinkertaisemmalta kantilta kuin Angular. Direktiiveihin perehtymällä voikin pian huomata, ettei se DOM:n manipuloiminen olekaan niin vaikeaa ja monimutkaista. Yleisenä ohjeena

jQuerystä Angulariin siirryttäessä onkin unohtaa kaikki, mitä on jQuerystä oppinut. (FAQ 2014.)

Kaiken kaikkiaan AngularJS on rakenteeltaan, toiminnallisuuksiltaan ja laajennettavuudeltaan erittäin kelpo ohjelmistokehys WWW-pohjaisten sovellusten luomisessa. Tätä päivitetään ja kehitetään jatkuvasti, mikä lupailee vielä pitkää ikää ja loisteliasta tulevaisuutta.

Seuraavissa luvuissa käydään läpi Angularista löytyviä eri osa-alueita, joita käyttämällä sovelluksesta saa monipuolisen, tehokkaan sekä selkeän kokonaisuuden ohjelmakoodillisesti.

3.3.2 Näkymät

Angularilla luodaan nk. yhden sivun sivustoja tai sovelluksia (single-page application), joiden toiminta perustuu erilaisten näkymien reitittämiseen. Näkymät ovat käytännössä HTML-sivuja, joihin sijoitetaan tuttujen ja turvallisten HTML-tagien ja muotoilujen lisäksi erilaisia muuttujia ym. viittauksia Javascriptin puolella luotuihin muuttujiin ja metodeihin sekä erilaisiin direktiiveihin.

3.3.3 Direktiivit

Direktiiveillä hallinnoidaan ja muokataan DOMia. Angular pitää sisällään oletuksena useita erilaisiin käyttötarkoituksiin soveltuvia direktiivejä, kuten aikaisemmin mainittu `ngModel`. Muita usein käytettäviä ovat mm. `ngShow` ja `ngHide`, joita käytettään elementtien näyttämiseen ja piilottamiseen, sekä esimerkiksi taulukon tai listan tietojen tulostamiseen käytetty `ngRepeat`. Direktiivien nimet voidaan kirjoittaa monella tapaa elementteihin, sillä ne käännetään kuitenkin samalla tavalla. Alla esimerkki `ngRepeat`-direktiivin toiminnasta. (Mt.)

```
// Javascript
$scope.enemies = ["Dr. Von Goosewing", "Gaston and Pierre", "Pirate Penguins"];

// HTML
<ul>
       <li ng-repeat="enemy in enemies">{{enemy}}</li>
</ul>
```

Taulukon kenttien sisältö määritellään enemies-nimiseen taulukkomuuttujaan, joka on määritelty osaksi $scope-muuttujaa, joka on eräänlainen liima kontrollereiden ja näkymien välillä (Scopes 2014). Tällaiseen muuttujaan lisättyä tietoa voidaan käyttää HTML:n seassa kuten aikaisemmassa ngModel-esimerkissä. Kuviosta 9 nähdään lista, jonka ngRepeat tulostaa.

- Dr. Von Goosewing
- Gaston and Pierre
- Pirate Penguins

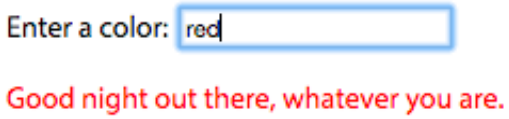
Kuvio 9. NgRepeat-esimerkin tulostus

Direktiivejä voi myös luonnollisesti luoda itse ja tämä onkin erittäin suotavaa DOM:n manipuloinnin helpottamiseksi. Seuraavassa esimerkissä luodaan colorChange-direktiivi, jota käytetään sijoittamalla näkymään color-change-elementti. Kuviosta 10 nähdään lopputulos, kun tekstikenttään on kirjoitettu punaista väriä tarkoittava sana. Värin voi syöttää niinikään kaikissa CSS:ssä tuetuissa värimuodoissa, eli esimerkiksi heksadesimaali- ja RGB-muodoissa.

```
// Javascript
app.directive("colorChange", function() {
    return {
        restrict: "E",
        replace: true,
        template: "<p style='color:{{color}}'>Good night out there,
whatever you are.</p>",
        link: function(scope, element, attrs) {
            element.bind("click", function() {
                scope.$apply(function() {
                    scope.color = "blue";
                });
            });

            element.bind("mouseover", function() {
                scope.$apply(function() {
                    scope.color = "orange";
                });
            });
        }
    };
});

// HTML
Enter a color: <input type="text" ng-model="color"/>
<color-change></color-change>
```



Kuvio 10. Itsetehty direktiivi tositoimissa

Direktiivi korvaa elementin `template`-kohtaan määritellyllä näkymällä. Elementtiin myös sidotaan kuuntelijat klikkaukselle ja hiiren päälle viemiselle. Elementtiin on määritetty värjäämään teksti `color`-muuttujan mukaan, jota pystyy muuttamaan tekstikentän avulla. Klikkaamalla tekstiä väri muutetaan siniseksi ja hiiren ollessa päällä väri muuttuu oranssiksi. Angularin omalla `$apply`-metodilla tiedotetaan sovellukselle värimuutoksesta, että tämä osataan päivittää näkymään. Yleensä tämä suoritetaan automaattisesti, mutta tässä tapauksessa `$scope`-muuttujan arvoa muutetaan direktiivissä, jolloin muutosta ei huomata. (Mt.)

3.3.4 Kontrollerit

Tavallisesti jokaisella näkymällä on oma kontrollerinsa (controller), joka sisältää aiemmin mainitun `$scope`n lisäksi metodeja näiden muuttujien hallinnointiin. Kontrollereita ei pitäisi käyttää DOM:n manipulointiin, sillä tätä varten ovat direktiivit. Kontrollereihin ei pitäisi myöskään sijoittaa paljoa logiikkaa, laskemista tai esimerkiksi yhteydenottoja palvelimeen, vaan nämä pitäisi ulkoistaa palveluihin sovelluksen rakenteen selkeyttämiseksi. Kontrollerin on tarkoitus toimia ns. portinvartijana näkymien ja palveluiden välillä, jonne sijoitetaan näkymissä tarvittavat muuttujat ja metodit. (Controllers 2014.)

3.3.5 Palvelut

Angularista löytyy monenlaisia palveluita, jotka poikkeavuuksia toisistaan voi olla vaikea ymmärtää, mutta kaikkien käyttötarkoitus ja periaate on kuitenkin sama. Palveluita käytetään mm. jakamaan metodeja sovelluksen eri kontrollereiden ja direktiivien välillä. Palveluihin on myös hyvä sijoittaa laskemista ja muita tehtäviä pitääkseen kontrollerit siisteinä ja yksinkertaisina. Kaikki palvelumuodot ovat singleton-tyyppisiä, eli näistä an-

netaan vain yksi instanssi yhtä kontrolleria kohti, mutta palvelun voi liittää moneen kontrolleriin. (Services 2014.)

`Service` ja `factory` muistuttavat erittäin paljon toisiaan, mutta ajavat silti käytännössä saman asian. Nämä poikkeavat toisistaan kirjoitusasun ja käyttötavan perusteella. Yleisenä nyrkkisääntönä palvelutyyppiä valittaessa toimii se, että jos et tiedä näiden eroja, valitse `factory`. `Provider`-palvelu on ainut näistä muutamasta, jota voidaan käyttää sovelluksen moduuleja konfiguroitaessa, jos halutaan kaikkien moduulissa kiinni olevien saada muutokset. Angularista löytyy luonnollisesti oletuksena palveluita, joista suosituimpia ovat `$http`- ja `$resource`-palvelut, joita molempia käytetään kommunikointiin palvelimen kanssa REST-menetelmiä hyödyntäen. (Services 2014.)

Seuraava esimerkki on hyvin yksinkertainen ja pelkistetty versio siitä, kuinka mm. `factory`-palvelua voidaan käyttää pienessäkin sovelluksessa. Kontrolleria määritettäessä tähän tuodaan `enemies`-factory, johon viitataan tällä nimellä jatkossa.

```
app.controller("ExampleCtrl", function($scope, enemies) {
    $scope.enemies = enemies.getEnemies();

    $scope.addEnemy = function(enemy) {
        enemies.addEnemy(enemy);
        $scope.enemies = enemies.getEnemies();
    };
});

app.factory("enemies", function() {
    var enemies = ["Dr. Von Goosewing", "Gaston & Pierre", "Pirate Pen-
guins"];

    function getEnemies() {
        return enemies;
    }

    function addEnemy(enemy) {
        enemies.push(enemy);
    }

    return {
        getEnemies: getEnemies,
        addEnemy: addEnemy
    };
});
```

Painiketta klikkaamalla sovellus lisää palveluun määriteltyyn taulukkoon alkion ja päivittää näkymässä näytettävän listan. Listaa tulostetaan aikaisemman ngRepeat-esimerkin tavoin.

```
Add enemy: <input type="text" ng-model="enemy"/>
<button ng-click="addEnemy(enemy)">Add</button>
```

3.3.6 Filtterit

Filttereillä (filter) eli suodattimilla voidaan muokata esimerkiksi taulukossa olevia tietoja lennossa pelkästään näkymää varten. Angularissa on lukuisia sisäänrakennettuja filttereitä, joista mm. currency muuttaa syötteen oletuksena dollarimuotoon ja date:lla voidaan suodattaa syöte päivämääräksi mihin muotoon tahansa käyttäen ennalta sovittuja merkkejä kuvaamaan päivää, kuukautta jne. Filttereitä voi myös tehdä itse. Seuraavassa esimerkissä luodun filtterin läpi mennessään tekstikenttään syötetty tieto tulostetaan lopusta alkuun sekä isoilla kirjaimilla tai alkuperäisessä kirjoitusasussa riippuen siitä, ruksataanko kohdan valintaruutu. (Filters 2014.)

```
// Javascript
app.filter("reverse", function() {
    return function(input, uppercase) {
        input = input || "";
        var out = "";

        for (var i = 0; i < input.length; i++) {
            out = input.charAt(i) + out;
        }

        if (uppercase) {
            out = out.toUpperCase();
        }
        return out;
    };
});

// HTML
Your name: <input type="text" ng-model="name"/>
Uppercase: <input type="checkbox" ng-model="uppercase"/>
<h1>Hello {{ name | reverse:true }}</h1>
```

Kuviossa 11 nähdään filtteriä hyödyntäen saatu tulostus.

Kuvio 11. Itse tehdyn filtterin lopputulos

3.4 Kaaviot

On sanomattakin selvää, että järjestelmän kaavioita muodostavan osan täytyi myös olla joku olemassa oleva Javascript-pohjainen kirjasto, sillä tämä tulisi olemaan hyvin tiukasti yhteydessä AngularJS:llä tehtyyn asiakaspuoleen. Tuotteen täytyi myös ennen kaikkea olla ilmainen, sillä opinnäytetyötä varten ei ollut resursseja ostaa mitään testin vuoksi.

Ihanteellisin valinta kaavioiden piirtäjäksi olisi ollut interaktiivinen ja hyvin pitkällä kehitetty Highcharts JS. Tästä löytyy kaikki mahdollinen, mitä voisi kaavioilta haluta. Kaavioiden ulkonääksi on lukuisia eri vaihtoehtoja, joista palkki-, donitsi-, tutka- ja aaltokaaviot ovat tavallisimpia ja näin ollen välttämättömiä. Highcharts muodostamat kaaviot ovat SVG-formaatissa, joka takaa näiden hyvän skaalautuvuuden eri kokoisille näytöille ja ikkunoille. Highcharts on ilmainen henkilökohtaisille ja voittoa tavoittelemattomille projekteille. (Comparison of JavaScript charting frameworks 2014.)

Toinen varteenotettava vaihtoehto oli Chart.js. Tämän kirjaston hyviin puoliin kuuluu ehdottomasti täysin ilmainen käyttö, oli sitten kyseessä miljoonan kävijän sivusto tai henkilökohtainen salainen projekti. Tällä kaavioiden muodostaminen on helppoa ja siistiä, sillä vaikka kirjastosta löytyy vain kuusi erilaista kaaviotyyppiä, joita luoda, on tämäkin määrä tarpeeksi. Chart.js:n takaa löytyy laaja vapaaehtoisten kehittäjien joukko, ja ominaisuuksia kehitetään jatkuvasti. Toisin kuin Highcharts, muodostaa Chart.js kaavionsa HTML5:n canvas-pohjaisiksi. Viimeisimpänä lisänä erittäin paljon toivottu interaktiivisuutta lisäävä tooltip, joka näyttää lukuja kun hiirtä vie kaavion yli. (Mt.)

Oman lisänsä Chart.js:ään tuo niinikään vapaaehtoisten projekti Angles.js, joka on luotu helpottamaan Chart.js:n ja AngularJS:n yhteiskäyttöä. Tämä pieni kirjasto koostuu kaikessa yksinkertaisuudessaan yhdestä direktiivistä, joka helpottaa huomattavan paljon kaavioiden luontia ja vie websivujen rakennetta kohti ”angularimaisempaa” rakennetta. (Silver 2014.)

Vaikka Highchartsin ominaisuudet ovat aivan omaa luokkaansa monipuolisuuden ja loppuun hiotun ulkokuoren perusteella, valittiin silti kaavioiden toteutustavaksi Chart.js

höystettynä Angles.js:llä. Chart.js saattaa olla hyvinkin riisuttu ja pelkistetty versio Highchartsista, mutta tämä ajaa asiansa oikein hyvin.

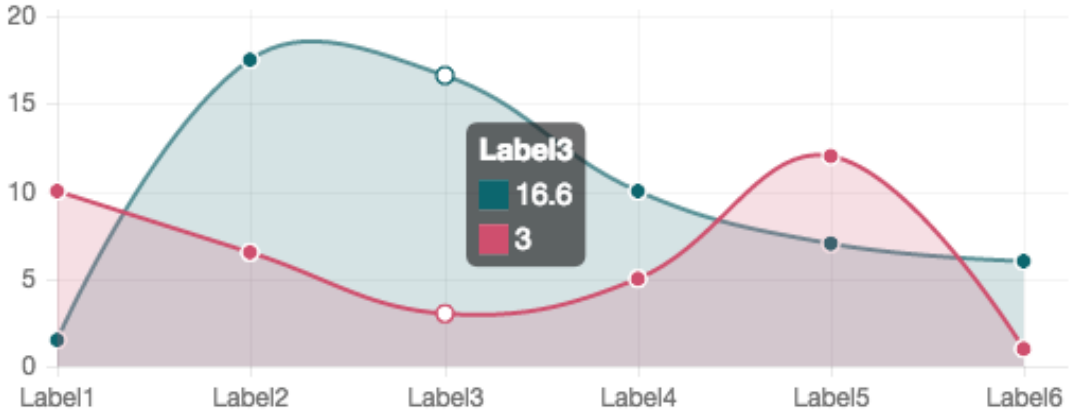
Seuraavassa esimerkkikoodissa määritellään AngularJS:ssä olevaan kontrollerin `$scope`-muuttujaan kaksi muuttujaa, kuten luvussa 3.3.3 niihin jo tutustuttiinkin. Tässä käytännössä annetaan kirjastolle tietoja siitä, minkä värisiä piirrettävät kaaviot ovat, ja millaista tietoa sen täytyy näyttää ja millä nimikkeillä.

```
$scope.chart = {
    labels: ["Label1", "Label2", "Label3", "Label4", "Label5", "La-
bel6"],
    datasets: [
        {
            label: "Dataset label 1",
            fillColor: "rgba(16,91,99,0.2)",
            strokeColor: "rgba(16,91,99,0.7)",
            pointColor: "rgba(16,91,99,1)",
            pointStrokeColor: "#fff",
            pointHighlightFill: "#fff",
            pointHighlightStroke: "rgba(16,91,99,1)",
            data: [1.5, 17.5, 16.6, 10, 7, 6]
        },
        {
            label: "Dataset label 2",
            fillColor : "rgba(200,70,99,0.2)",
            strokeColor : "#C84663",
            pointColor : "rgba(200,70,99,1)",
            pointStrokeColor: "#fff",
            pointHighlightFill: "#fff",
            pointHighlightStroke: "rgba(200,70,99,1)",
            data: [10, 6.5, 3, 5, 12, 1]
        }
    ]
};

$scope.options = {
    tooltipFillColor: "rgba(30,30,32,0.7)",
}
```

HTML:n puolella määritellään canvas-elementtiin halutun kaaviotyypin nimi. Tässä tapauksessa käytettiin linechart eli viivakaaviota. Tämän nimen perusteella Angles.js sijoittaa canvakseen oikean kaavio-direktiivin, joka käyttää canvakseen määriteltyjä `options`- ja `data`-kenttien muuttujia, joista löytyy edellisessä koodissa esitellyt tiedot. Näillä tiedoilla tulostetaan kuvion 12 mukainen kaavio, jonka tietoja pystyy selaamaan tarkemmin viemällä hiirtä kaavion yli.

```
<canvas linechart options="options" data="chart" width="500"
height="200"></canvas>
```



Kuvio 12. Chart.js:n luoma kaavio

4 Järjestelmän toteutus

4.1 Kerättävien tietojen määrittäminen

Markkinoilta löytyvien pelianalysointijärjestelmien tietojen keräys ja tilastojen muodostaminen noudattavat perusominaisuuksiltaan samaa kaavaa. Luonnollisestikin yritys haluaisi ensimmäisenä tietää, kuinka moni pelaa peliä ja kuinka usein. Näistä muodostuukin selkäranka pelitietojen keräykselle. Opinnäytetyössä tahdottiin keskittyä olennaisen ja välttämättömän tiedon keruuseen, jotta saataisiin muodostettua kaavioita kaikista hyödyllisimmistä ja kriittisimmistä pelin analysointiin tarvittavista osa-alueista. (McCalmont 2013; Todor 2014.)

Seuraavaksi on listattu työhön sisällytettyjä analysointijärjestelmien välttämättömimpiä ominaisuuksia, joiden puuttuminen kyseenalaistaisi järjestelmän varteenotettavuuden. Näiden lisäksi markkinoiden analysointijärjestelmät tarjoavat paljon muitakin tilastoja muun muassa rahan liikkumisen suhteen. Tästä voidaankin muodostaa monenlaisia kaavioita, kuten pelin tuotto jokaista päivittäistä käyttää kohden (ARPDAU). Pelin tuottoa kuvaavat tilastot ovat kuitenkin jääneet opinnäytetyön ulkopuolelle näiden monimutkaisuuden ja vuoksi. (McCalmont 2013.)

Uudet käyttäjät

Luonnollisesti halutuin tieto on se, kuinka moni uniikki käyttäjä on löytänyt tuotteen eli tässä tapauksessa pelin. (Mt.)

DAU, Daily Active Users

Päivittäisten käyttäjien tietäminen on myös yksi halutuimmista tiedoista, sillä tästä nähdään, mille päiville sijoittuu pelaajapiikit ja kuinka paljon peli koukuttaa pelaajia. (Mt.)

MAU, Monthly Active Users

Kuukausittaiset uniikkien käyttäjien määriä seuraamalla saa hyvin kuvan, kuinka pelin pelaajakanta on kehittynyt ajan saatossa. (Mt.)

Istuntojen pituus ja määrä

Tutkimalla istuntojen määriä sekä kestoja pelaajien keskuudessa voidaan selvittää, kuinka kauan pelaaja keskimäärin jaksaa pelata peliä ja kuinka monesti. Istuntojen määriä tutkaillessa pystyy esimerkiksi näkemään, kuinka moni käyttäjä on avannut pelin ja lopettanut käyttämisen siihen. (Mt.)

Virheet, varoitukset, lisätieto

Vaikka luonnollisesti peliä tehdessä tähdätään siihen, ettei virheitä tapahdu ja näin ollen pelaajalle aiheudu päänvaivaa ja turhautuneisuutta, välillä sattuu virhetilanteita. Näihin virhetilanteisiin tulisi peliä tehdessä varautua edes ottamalla virheet talteen ja listaamalla niitä analysointityökaluilla. Näin saataisiin esimerkiksi tietää, tapahtuuko pelissä kovin usein Internetin katkeamista tai pelin kaatumista. Tarvittaessa peliin voi myös upottaa tiedonlähetyksiä sellaisiin kohtiin, joista halutaan syystä tai toisesta pitää kirjaa. Tällaisia kohtia voisi olla muun muassa pelaajan nykyisen käyttöjärjestelmän tai prosessorin selvittäminen. (Mt.)

Retention

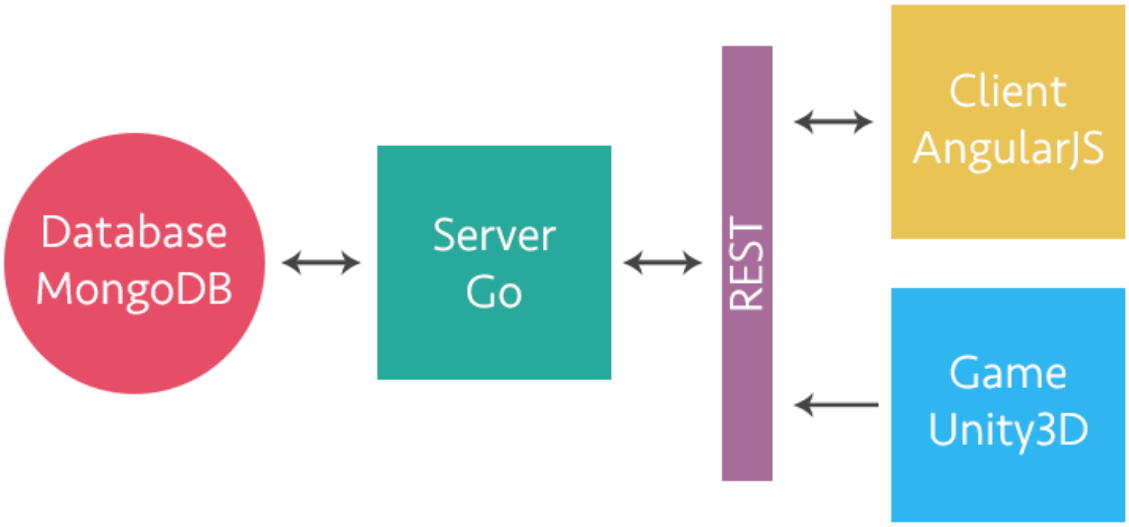
Niin kuin luvussa 2.2.2 jo todettiinkin, on retentionin tarkkailu yksi hyvin tärkeä osa pelin analysointia. Retention kertoo siis kuinka hyvin peli on saanut pidettyä vanhat pelaajat pelin parissa huolimatta uusien pelaajien rekisteröitymisestä.

Virheiden ynnä muiden tapahtumien listausta lukuun ottamatta muut tilastot saadaan muodostettua ottamalla talteen käyttäjäkohtaisesti rekisteröintiajankohdan sekä talti-oimalla käyttäjien jokaisen istunnon ajankohdan sekä pituuden. Nämä ovat perustiedot, joilla saadaan luotua jo kattavat kaaviot pelin käyttäjistä ja näiden aktiivisuudesta.

Näiden yleisten tilastojen selvittämisen lisäksi peleihin yleensä lisätään pelityypistä liittyen esimerkiksi seuranta ostotapahtumille, tasojen läpäisemisille, tavaroiden keräämiselle, kuolemille ja jaoille sosiaalisessa mediassa. Näiden ominaisuuksien toteuttaminen on kuitenkin sen verran vaativaa ja aikaa vievää, ettei opinnäytetyön toteutukseen otettu mukaan mitään yksilöityä tapahtuman tilastointia. Myöskään luonnollisesti luvussa 2.2.2 kuvattujen funneleiden toteuttamiseen ei tahdottu käyttää resursseja, sillä nämä ovat jo huomattavasti korkeammalla tasolla eivätkä olisi mahtuneet toteutusajankohtaan.

4.2 Järjestelmäarkkitehtuuri

Toteutukseen valitut teknologiat ovat suhteiltaan kuvion 13 mukaiset. Järjestelmää käytetään clientistä eli www-selaimesta käsin, jossa hyrrää taustalla AngularJS:llä tehty sovellus. Sovellus tekee käyttäjän pyyntöjen perusteella joko hakuja tai tallennuksia kantaan REST-rajapinnan avulla. Palvelimeen on määritelty osoitteet, joihin saapuessa tietynlainen pyyntö suoritetaan tietty metodi palvelimen sisällä. Opinnäytetyön tapauksessa pyynnöt koskevat aina tietokantaa jollain tavalla. Kaavioita muodostettaessa palvelimelle saapuvan HTTP-pyynnön perusteella kantaan suoritetaan haku, jonka tuloksia pyynnössä kaipaillaan. Tietokanta palauttaa parhaansa mukaan tulokset, jotka palvelin palauttaa takaisin asiakaspuolelle AngularJS:ään. Siellä tuloksille tehdään tarvittavat siistimiset ja muokkaukset, jotta tieto voidaan lisäkirjastojen avulla muodostaa kaavioiksi näytölle. Samanlainen kiertokulku on myös muilla operaatioilla, eli esimerkiksi tietojen päivittämisellä.



Kuvio 13. Järjestelmäarkkitehtuuri

Myöskin varsinainen peli lähettää siihen integroidun pluginin avulla tietoja pelaamisesta REST:n välityksellä palvelimelle, josta ne löytävät tiensä tietokantaan. Peliin ei tuoda missään vaiheessa palvelimelta mitään tietoja, joten tästä syystä yhdensuuntainen liikenne.

4.3 Tietokanta

4.3.1 Rakenne

Kannassa ”user” tarkoittaa järjestelmän käyttäjää, joka voi luoda pelejä ja seurata niiden tilastoja ja ”player” tarkoittaa loppukäyttäjää eli pelaajaa, joka asentaa pelin laitteelleen ja pelaa sitä.

Ennen kuin tietokannan suunnittelu voitiin aloittaa, piti tehdä muutamia kriittisiä valintoja rakenteen suhteen. Ensimmäinen näistä koski pelaajien sijoittamista kantaan. Vaihtoehtoja oli kolme:

1. Pelaajat lajitellaan omiksi dokumenteiksi Players-kokoelmaan. Näiden alta löytyy pelit, joihin pelaaja on rekisteröitynyt, sekä näihin tehdyt istunnot. Huono puoli tässä on se, että todennäköisesti pelaajalla on vain yksi peli tiedoissaan, johon istuntoja kerätään. Aiheuttaa vain ylimääräisiä taulukoita ilman suurempaa hyötyä.

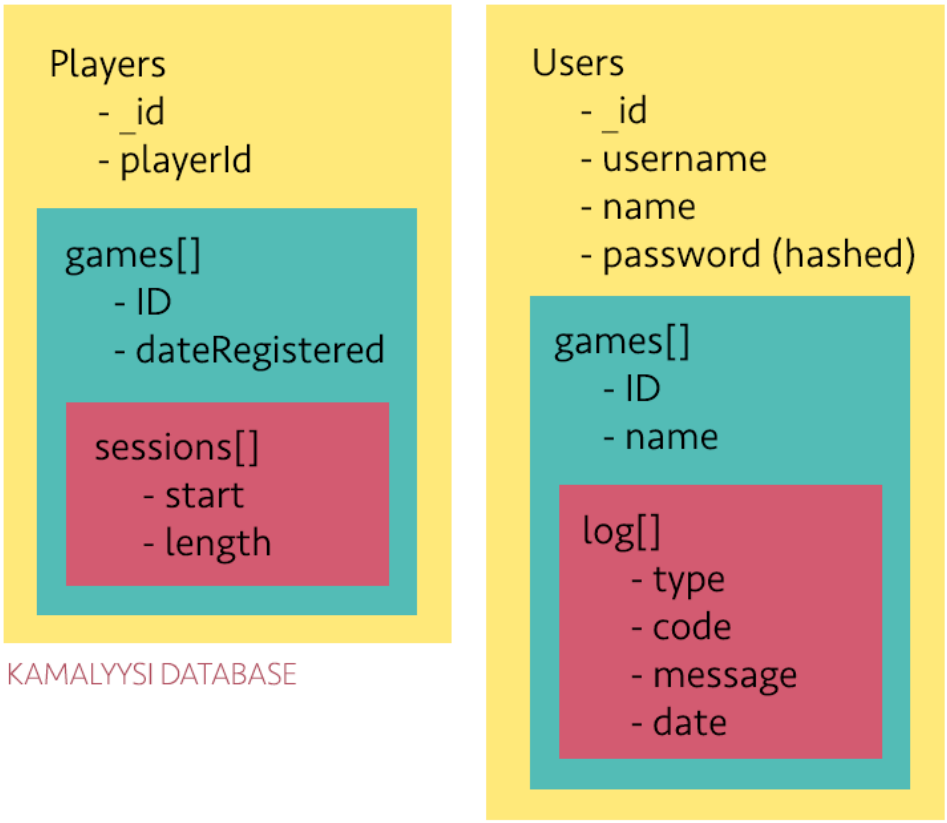
2. Pelaajat sijoitettaisiin pelattavan pelin dokumenttiin taulukkoon, jossa säilöttäisiin myös istunnot ja muut pelaajan tiedot. ”Mongomaisin” ratkaisu, jos peli vielä sijoitettaisiin tämän omistavan käyttäjän tietoihin. Ilmeisimpänä vaarana ja kompastuskivenä dokumenttien paisuminen hillittömiin kokoluokkiin, sillä yhden pelin jokaisen pelaajan jokainen istunto talletettaisiin samaan dokumenttiin.
3. Players-kokoelmaan sijoitetaan aina dokumentti jokaista uutta pelaaja + peli kombinaatiota varten. Dokumentissa olisi vielä talletettu peliin tehdyt istunnot kyseiseltä käyttäjältä. Relaatiomaisin ratkaisu, vaikkakin selkein.

Toteutustavaksi valittiin ensimmäinen vaihtoehto. Toisaalta ei olisi ollut väliä, valitaanko ensimmäinen vai kolmas, sillä todennäköisesti dokumenttimäärä tulisi pysymään samana. Ensimmäinen on kuitenkin mongomaisin ratkaisu sitten toisen vaihtoehdon jälkeen, joka on huono dokumenttien koon massiivisuuden vuoksi.

Seuraavaksi täytyi päättää, pystyykö luotuun peliin liittämään useampia käyttäjätunnuksia, joilla pääsee tutkimaan pelin tilastoja. Järjestelmän tavoitellessa yksinkertaisuutta ja helppokäyttöisyyttä päätettiin rajoittaa pelin näkyminen vain yhdelle käyttäjälle.

Kuviossa 14 nähdään tietokannan rakenne kuvattuna laatikkomuodossa. Sisäkkäiset laatikot tarkoittavat lapsidokumentteja, kuten luvussa 3.1.2 niihin jo tutustuttiinkin. Mongomaisin tapahan olisi ollut sijoittaa Users-kokoelman sisälle vielä yksi taulukko, joka olisi ollut rakenteeltaan melkein sama kuin Players-kokoelma. Poikkeuksena se, ettei pelejä olisi tarvinnut relaatiomaisesti linkittää toisiinsa viittaamalla ID-kenttiin. Kuten kuitenkin jo todettiin, olisi tämä ollut varsin huono ratkaisu, joten pelaajat pysyvät erillään käyttäjän luomista peleistä, ja näihin viitataan vain ID:llä. Periaatteessa _id-kenttä on turha, sillä playerId tulee olemaan yksilöity pelilaitteesta saatava tunnus, mutta selkeyden vuoksi sen annettiin generoitua Mongon toimesta. Jokainen peli, johon nimenomainen pelaaja rekisteröityy lataamalla pelin, talletetaan pelaajan games-taulukkoon objektina, josta löytyy pelin ID ja päivämäärä, milloin pelaaja rekisteröityi. Näiden lisäksi löytyy myös sessions-taulukko, johon pelaajan tekemistä istunnoista säilötään ajankohta sekä kesto.

Users-kokoelmasta löytyy _id-kenttä niinikään samasta syystä kuin Players-kokoelmasta. Tämän lisäksi käyttäjällä täytyy olla tunnus sekä salasana, joka salataan ennen tietokantaan laittoa tietoturvasyistä. Nimi on vapaaehtoinen. Käyttäjän luomat pelit sijoitetaan games-lapsidokumenttiin, jonne säilötään pelien generoidut ID:t, nimet sekä tapahtumalokit. Lokeihin tallennetaan kaikki pelin puolelta lähetetyt viestit ja tapahtumat, kuten virheet. Olisi ollut turha sijoittaa peleissä tapahtuneet virhetilanteet pelaajien alle Players-kokoelmaan, sillä se ei ensinnäkään ole oleellista, kenelle tämä tapahtui ja toisekseen niiden säilömisestä siellä ei olisi mitään hyötyäkään. Tapahtumista säilötään tyyppi (virhe, info …), viesti ja ajankohta, sekä mahdollisesti myös koodi.

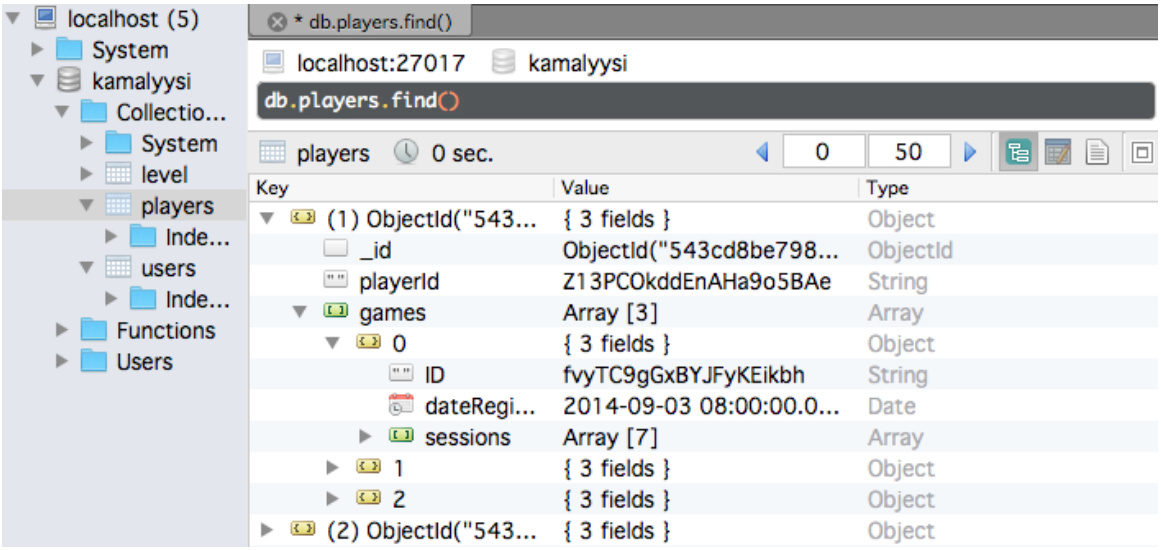


Kuvio 14. Tietokannan rakenne

4.3.2 Menetelmät

Tietokannan suunnittelussa ja testauksessa käytettiin hyvin paljon graafista Robomongo-nimistä ilmaisohjelmaa, joka listaa selkeästi palvelimelta löytyvät tietokannat kokoelmi-

neen päivineen. Ohjelmalla on helppo tehdä hakuja ja muutoksia kantaan. Kuviossa 15 näkymä siitä, kun Players-kokoelmasta on haettu kaikki dokumentit.



Kuvio 15. Kuvakaappaus Robomongo-ohjelmasta

Aivan kaikkeen Robomongoa ei voinut kuitenkaan käyttää, sillä tämä ei esimerkiksi tunnista päivämääriä niiden ollessa Date-objektimuodossa, eli `new Date("2014-02-18 13:37:10")`. Tämä oli suuri haittapuoli, sillä monet kantaan tehdyt aggregaatiot tarvitsivat ajankohdan rajausta. Osaksi testausta täytyi siis tehdä perinteisin menetelmin, eli komentokehotteen kautta. Sinänsä eroa ei ollut sen kummemmin, sillä Robomongoon syötetään täsmälleen samanlaisia hakulausekkeita ja muita komentoja kuin mitä komentokehotteelle. Erona vain graafisen käyttöliittymän puuttuminen.

4.3.3 Käyttö

Kaikki tilastojen saamiseksi tehtävät käskyt täytyi muodostaa käyttämällä luvussa 3.1.3 esiteltyä aggregaatiota. Seuraavaksi esimerkki siitä, kuinka uusien käyttäjien päiväkohtaiset lukumäärät saadaan selville.

```
db.players.aggregate(
{ "$unwind" : "$games" },
{ "$match" : {
"games.ID" : "pelinID",
       "games.dateRegistered" : {
              "$gte" : new Date("2014-10-01T00:00:00.000Z"),
              "$lte" : new Date("2014-10-14T23:59:59.000Z")
}
```

```
}},
{ "$group" : {
              "_id" : {
                    "year"  : { "$year" : "$games.dateRegistered" },
                    "month" : { "$month" : "$games.dateRegistered" },
                    "day"   : { "$dayOfMonth" : "$games.dateRegistered"}
              },
              "date"  : { "$first" : "$games.dateRegistered" },
"count" : { "$sum": 1 }
}},
{ "$project" : {
"_id" : 0,
"date" : 1,
       "count": 1
}},
{ "$sort" : { "date": 1 } }
);
```

1. Ensimmäisenä tehtävän `$unwind`-operaation tarkoitus on lajitella jokainen games-taulukossa oleva dokumentti omaksi dokumentikseen vanhempi mukanaan.
2. Seuraavaksi suodatetaan mukaan ne, joissa on tutkittavan pelin ID ja pelaajan rekisteröitymispäivämäärä on jälkeen ajankohdan alun, mutta ennen ajankohdan loppua.
3. Ryhmittelyvaiheessa `_id`-kenttä muodostetaan päivämäärän eri osista. Tämän avulla tunnistetaan, millä dokumenteilla on sama päivämäärä. Jokaisen saman päivämäärän omaava dokumentti nostaa `count`-kenttää yhdellä, ilmoittaen näin tuona päivänä rekisteröityneet pelaajat. `Date`-kenttään sijoitetaan vain yksinkertaisesti päivämäärä kokonaisuudessaan ottamalla tämän ensimmäinen ilmentymä `$first`-operaatiolla.
4. Tulostuksen siisteyden vuoksi `$project`-operaattorilla määritellään, että `_id`-kenttä piilotetaan ja `date`- sekä `count`-kentät näytetään.
5. Lopuksi vielä tuloslistaus järjestetään `date`-kentän mukaan nousevaan järjestykseen.

Edellinen aggregaatio tulostaa esimerkiksi seuraavanlaisen tulostuksen.

```
{ "date" : ISODate("2014-09-01T13:37:00Z"), "count" : 1 }
{ "date" : ISODate("2014-09-02T11:40:30Z"), "count" : 7 }
{ "date" : ISODate("2014-09-03T07:03:05Z"), "count" : 3 }
```

Loput tilastoja varten muodostetut haut noudattavat kutakuinkin samaa kaavaa. Eroja on lähinnä operaatioiden järjestyksessä `pipelinessä`, eli putkessa, ja ryhmittelyssä `$group`-operaation avulla.

4.4 Palvelin

4.4.1 REST

Palvelimen tehtäviin kuului luonnollisesti näyttää sovelluksen käyttäjälle oikea sivusto sekä käsitellä siihen tehtyjä REST-pyyntöjä. Palvelin on yhteydessä tietokantaan käyttäjien tekemien pyyntöjen mukaisesti, sekä palauttaa näille tarvittavia tietoja, jos näitä halutaan. Go-palvelimen pystyttäminen on helppoa, nopeaa ja lopputulos on ennen kaikkea kevyt ja yksinkertainen, mutta toimiva.

Apuna palvelimen luomisessa käytettiin Go:lle löytyvää mux-nimistä kirjastoa. Kirjastolla voi monipuolisesti määritellä mm. alidomaineihin suunnattuja pyyntöjä. Seuraavassa esimerkkikoodissa esitellään järjestelmää varten luotu palvelin pelkistetysti perehtymättä liikaa yksityiskohtiin. Alussa määritellään kaksi uutta tyyppiä parantamaan koodissa tapahtuvien virheiden käsittelyä. `ServeHTTP`-metodia käytetään niinikään virheiden käsittelyssä, mutta on jätetty pois tämän pituuden ja kryptisyyden vuoksi. (mux n.d.)

```
type (
	handlerError struct {
		Error   error
		Message string
		Code    int
	}

	handler func(w http.ResponseWriter, r *http.Request) (inter-
face{}, *handlerError)
)

func (fn handler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
	// ...
}
```

Esimerkki jatkuu pääohjelmalla, joka aloitetaan määrittämällä kansio, jonka sisältämät HTML-dokumentit näytetään selaimessa käyttäjälle. Tämän jälkeen määritellään mux-tyyppiselle router-muuttujalle URL-polkuja, joihin suunnatut erilaiset pyynnöt toteutetaan määritellyllä metodilla riippuen lähetystavasta (GET, POST, PUT tai DELETE). Esi-

merkkiin on kerätty muutama erilainen polku eri tarkoituksiin. Lopussa määritellään palvelimen toimivan portissa 1337, eli tässä tapauksessa paikallisella koneella pyörivän palvelimen web-käyttöliittymään pääsee osoitteella `localhost:1337`.

```
func main() {
	fs := http.Dir("web/")
	fileHandler := http.FileServer(fs)

	router := mux.NewRouter()
	router.Handle("/api/stats/{id}/{type}/{from}/{to}",
handler(getGameStats)).Methods("GET")
	router.Handle("/api/players",
handler(playerAction)).Methods("POST")
	router.Handle("/api/user", handler(updateUser)).Methods("PUT")
	router.Handle("/auth/login", handler(login)).Methods("POST")

	router.PathPrefix("/")
.Handler(http.StripPrefix("/", fileHandler))
	http.Handle("/", router)

	log.Printf("Running on port :1337")
	http.ListenAndServe(":1337", nil)
}
```

Pelitilastojen saamiseksi käytettyyn getGameStats-metodiin pääsee siis käyttämällä URL-osoitetta, johon on määritelty pelin ID, tilastojen tyyppi sekä ajankohdan alku ja loppu. GET-tyyppisiä pyyntöjä voidaan myös toteuttaa ihan tavalliselta selaimen osoiteriviltä, eikä pelkästään AngularJS:n uumenista. Annettuihin muuttujiin päästään helposti käsiksi `getGameStats`-metodin sisällä mux-kirjaston avulla seuraavan koodinpätkän mukaisesti.

```
vars := mux.Vars(r)
gameId := vars["id"]
```

Muuttujia käytetään oikeiden tietojen hakemiseen tietokannasta. Tietokantaan suoritetut haut olivatkin suurimmaksi osaksi luvussa 4.3.3 läpikäydyn MongoDB:n aggregaation mukaisia. Näiden ja muiden lisäysten ja päivitysten tekemiseen käytettiin Go:lle saatavaa mgo-kirjastoa, joka on todella monipuolinen väline MongoDB:n käyttöön Go:lla.

4.4.2 Yhteys tietokantaan

Seuraavassa esimerkissä toteutetaan uusien pelaajien määriä kyselevä GET-pyyntö. Go ohjaa tämän edellisessä luvussa esiteltyyn `getGameStats`-metodiin, josta tämä ohjataan `newPlayers`-metodiin URL:ssa tuodun tyypin perusteella. Alussa luodaan yhteys meto-

din avulla, jonne on määritelty tietokannan osoite ja nimi, käyttäjätunnus sekä salasana. Koska aggregaatio on käytännössä identtinen luvussa 4.3.3 olevan esimerkin kanssa, on se jätetty pois koodista.

Aggregaatio suoritetaan mgo:n Pipe-metodilla, jonka avulla löytyvät tiedot, eli päivämäärä ja lukumäärä, sijoitetaan ensin luotuun `UsersCount`-tyyppiseen struct-muuttujaan, joka vielä lisätään myös saman tyyppisestä structista muodostettuun taulukkomuuttujaan. Tämä muuttuja palautetaan sinne, mistä GET-pyyntö alun perin annettiin, eli todennäköisesti web-sovellukseen, jossa vastausta käsitellään JSON-muuttujana.

```
UsersCount struct {
	Date  time.Time `bson:"date" json:"date"`
	Count int       `bson:"count" json:"count"`
}

func newPlayers(game string, from time.Time, to time.Time) []UsersCount
{
	c, s := getConnection("players")
	defer s.Close()

	pipeline := []bson.M{
		// Katso aggregaatio luvusta 4.3.3
	}

	var (
		result  UsersCount
		results []UsersCount
	)

	iter := c.Pipe(pipeline).Iter()
	for {
		if iter.Next(&result) {
			results = append(results, result)
		} else {
			break
		}
	}

	err := iter.Err()
	if err != nil {
		// Virheiden käsittely
	}

	return results
}
```

Istuntojen lisääminen pelaajien sammutettua pelin suoritetaan seuraavanlaisesti. Ensin määritellään, kenen pelaajan istuntoja lisätään sekä kyseessä oleva peli näiden ID-kenttien perusteella. doc-muuttujassa kerrotaan, kuinka kyseistä löydettyä dokumenttia

muokataan. Tässä tapauksessa dokumentin games-taulukon lapseen, josta annettu pelin ID löytyy, sessions-nimiseen taulukkoon lisätään uusi objekti, jolla on istunnon ajankohta sekä pituus. Dokumentti päivitetään mgo:n Update-metodilla.

```
query = bson.M{"playerId": id, "games.ID": gameId}
doc = bson.M{
	"$push": bson.M{
		"games.$.sessions": bson.M{
			"start":  session.Date,
			"length": session.Length,
		},
	},
}

err = c.Update(query, doc)
if err != nil {
	// Handle error
}
```

4.4.3 Retention

Siinä missä muut tilastot saadaan suoraan oikeanlaisina tietokannasta, täytyy retention laskemiseen tehdä toistolause jos toinenkin. Perusta retentionin laskemiselle saadaan hakemalla tietokannasta niin päiväkohtaiset uudet käyttäjät kuin DAU:t eli istuntoja omaavat käyttäjät. Näille molemmille tietojen hauille tehtiin omat aggregaatiolausekkeet edellisen luvun mallin mukaan. Alusta loppuun vaiheet ovat seuraavat:

1. Järjestelmän käyttäjä on valinnut kaavioon päivämääriksi 1. - 7.10. Tietokannasta haetaan siis tiedot 28 päivää ennen valittua ajankohtaa eli 3.9. - 7.10.
2. Haetut tiedot käydään läpi jokaista retentionpituutta (1, 7, 14, 28 päivää) kohden. Esimerkiksi Day 7 retentionia laskettaessa päivälle 3.10. uudet käyttäjät haetaan päivältä 26.9. Tämän jälkeen ajalta 27.9. - 3.10. otetaan talteen käyttäjät, joilla on istuntoja tuona aikana.
3. Vertaillaan, löytyykö rekisteröityneiden listan käyttäjiä tältä istuntoja omaavien listalta. Jos löytyy, käyttäjä merkitään säilytetyksi.
4. Retention saadaan selville, kun näiden säilytettyjen käyttäjien määrä jaetaan uusien käyttäjien määrällä. Esimerkiksi käyttäjiä rekisteröityi 26.9. yhteensä 22 kappaletta ja 3.10. mennessä näistä 15 oli avannut pelin jonain päivänä. Kaavaan syötettynä saadaan seuraavanlainen lauseke:

```
(15 / 22) * 100 ≈ 68 %
```

5. Vastaus kertoo, että 68 % viikko sitten rekisteröityneistä käyttäjistä on avannut pelin tämän jälkeen. Tämä lasketaan vallan hyväksi tulokseksi. (McCalmont 2013.)

Ohjelmakoodillisesti toteutus ei ole rakettitiedettä, vaan erilaisia ehto- ja toistolausekkeita milloin missäkin muodossa. Tästä syystä olisi turhaa avata koodia, sillä sitä on paljon ja se on paikka paikoin varsin vaikeasti ymmärrettävää. Sovellukseen palautetaan jokaisen retentionin pituuden tulokset omissa taulukoissaan, joista ne voidaan helposti ja kätevästi vain sijoittaa kaavioon sovelluspuolella.

Retentionin laskemisen olisi voinut suorittaa suoraan sovelluksessa toimivalla Javascript-kielellä, mutta Go on merkittävästi nopeampi tässä tarvittavien vertailujen suhteen. Palvelimella laskeminen on muutenkin paljon järkevämpää, sillä silloin ei rasiteta asiakaspäätteen tietokonetta suuria määriä.

Retentionin tarkkailuun valittiin lukuisista kaavoista versio, jossa käyttäjä katsotaan säilytetyksi, jos tämä kirjautuu peliin edes kerran retentionin pituuden aikana. Tätä kutsutaan nimellä ”return retention”, eli paluu retention. Tämä valittiin siksi, että se kertoo tarpeeksi kattavasti, kuinka hyvin on kutsunut pelaajia takaisin. Retentionin voi myös laskea esimerkiksi katsomalla jokaiselta päivältä, onko käyttäjä kirjautunut peliin. Jos välistä jää yksikin päivä pois, ei käyttäjää lasketa säilytetyksi. Näin ankaraa linjausta pitää versio, jota kutsutaan nimellä ”full retention”, eli täysi retention. Jokaisella versiolla on omat prosentuaaliset rajansa, joita tulisi pelin saavuttaa ollakseen ”terve”. Jokainen retentionin laskutapa on yhtä oikea, kunhan vain tietää, millaisia tuloksia sieltä pitäisi odottaa. (Sommer 2014.)

4.5 Sovellus

4.5.1 Yleistä

Sovelluksen asiakaspuolta toteutettaessa pyrittiin seuraamaan jotain tiettyä linjausta rakenteen kasaamisen suhteen ja muutenkin yleisen ohjelmakoodin rakenteen puolesta.

Tässä käytettiin hyödyksi GitHub-kehitysyhteisöstä löytyvää ohjekirjasta, joka on enemmän tai vähemmän ammattilaisen kasaama vinkkinivaska, kuinka AngularJS-sovellus kannattaa koostaa. Opas ei suoranaisesti opeta, kuinka AngularJS:n eri ominaisuuksia käytetään, vaan ennemminkin kuinka käyttää niitä oikein. Javascript-ohjelmointikieli kun on kuitenkin surullisen kuuluisa siitä, että sillä saa helposti ohjelmoitua, mutta vielä helpommin ohjelmoitua huonosti. (Papa 2014.)

Muiden AngularJS:llä tehtyjen sovellusten tapaan myös tässä työssä on käytetty tuiki tavallisia luvussa 3.3 esiteltyjä osa-alueita toimivan sovelluksen kasaamiseksi. Sovelluksesta löytyy luonnollisesti näkymät ja näille osoitettuja kontrollereita, sekä myös kaavioiden luomista helpottamiseksi luotuja direktiivejä ja palveluita. Seuraavissa luvuissa on käyty läpi sovelluksen toiminnan kannalta kriittisimpiä kohtia.

4.5.2 Tiedon hakeminen ja sen näyttäminen

Tässä luvussa keskitytään avaamaan kaavion elämänkaarta aina sovellukseen saapuneesta pyynnöstä diagrammin muodostamiseen. Kaikki alkaa siitä, kun käyttäjä saapuu tilastosivulle. Kontrollerissa suoritetaan metodi, joka käskyttää palvelua antamaan tilastotietoja, joita sijoittaa `$scope`-muuttujaan kaavioiden näyttämistä varten. Seuraavassa koodissa on määritelty `$resource`-palvelu, joka on yksi keino HTTP-pyyntöjen tekemiseen AngularJS:ssä. Tässä määritellään URL, jota pitkin HTTP-pyyntö lähetetään sisältäen mahdolliset muuttujat, joita vaaditaan onnistuneen kyselyn saavuttamiseen.

```
function StatFactory($resource, $http) {
    return $resource("api/stats/:id/:type/:from/:to",
        {id: "@id", type: "@type", from: "@from", to: "@to"},
        {"query": {
            method: "GET",
            isArray: true
        }}
    );
}
```

Tähän muuttujaan pääsee käsiksi tuomalla tämän attribuuttina mukana palvelumetodiin, jolloin se toimii suoraan muiden muuttujien tapaan. Seuraavassa koodissa Stat-muuttuja pitää sisällään edellä esitellyt `$resource`-ominaisuudet. Oletuksena tästä löytyy metodit GET-, POST- ja DELETE-pyynnöille. Muuttujaan suunnattu query-metodi ottaa sisäänsä objektin, johon sijoitetaan URL:n muodostamiseen tarvittavat parametrit.

Tämän lisäksi metodia kutsuttaessa annetaan myös metodi, joka suoritetaan kun vastaus HTTP-pyyntöön on saatu palvelimelta. Esimerkistä on poistettu epäoleellinen osuus metodin sisältä, jossa tulosta käsiteltiin ja palautettiin hämmentävissä muodoissa.

```
Stat.query({id: activeGame.ID, type: type, from: dateFrom, to: dateTo},
    function (data) {
        // Return data
    }
);
```

Tässä vaiheessa pyyntö lähetetään palvelimelle. Luvussa 4.4.1 esitellyn palvelinrakenteen mukaan pyyntö ohjautuu osoitteen mukaan oikealle käsittelijälle, josta se jatkaa toimintojaan samaisessa luvussa käytyjen askelien mukaisesti. Kun palvelin on saanut haluamansa tiedot tietokannasta, palautetaan nämä takaisin AngularJS:ään, jonka jälkeen suoritetaan aikaisemmassa esimerkissä määritelty metodi.

Kun kaikki tarvittava tieto on saatu palvelimelta, aloitetaan kaavioiden luominen. Olennaisena osana tätä prosessia on saadun tiedon muokkaaminen halutunlaiseksi ja sijoittaminen oikeisiin kohtiin kaaviossa. Tämä vaihe ei sinänsä ole mitään rakettitiedettä, vaan pitkä litania erilaisia ehtolauseita ja tiedon suodattamista Chart.js:lle sopivaksi.

Tietokannasta saadut tiedot talletetaan palvelun sisään, josta nämä kaiken päätteeksi sitten haetaan kontrollerissa olevaan `charts`-muuttujaan. Tämä taulukko-muuttujan täyttyessä näkymään määritelty `ngRepeat`-direktiivi herää, ja tulostaa jokaista `charts`-taulukossa olevaa objektia kohden div-elementin, jonka sisältä löytyy kaavion tyyppiä ja värimaailmaa muokkaavien valikkojen lisäksi kaavion muodostamiseen tarvittava direktiivi.

```
<div ng-repeat="chart in charts">
    <select ng-options="t as t for t in types"
            ng-model="chart.type"
            ng-change="modifyChart()">
    </select>
    <select ng-options="c as c for c in colors"
            ng-model="chart.color"
            ng-change="modifyChart()">
    </select>

    <div stat-chart info="chart" type="{{chart.type}}"></div>
</div>
```

Käytännössä `statChart`-direktiivi palauttaa luvussa 3.4 nähdyn esimerkin mukaisesti `canvas`-elementin, johon on sijoitettu `options`- ja `data`-attribuutteihin kaavion nimikkeiden ja tietojen lisäksi ulkoasutyylitykset. Elementtiin on myös määritelty kaavion tyyppi, joka toimii direktiivin nimenä, sillä työssä käytettävä Angles.js hoitaa lopullisen kommunikoinnin luomamme kaaviodirektiivin ja Chart.js:n välillä. Luvussa 5.2 käsitellään tämän vaiheen toteuttamisessa ilmennyttä ongelmaa.

4.5.3 Käyttäjätietojen ylläpito

AngularJS:n `$resource`-palvelua käytetään myös käyttäjän tietojen ja pelien ylläpitoon. Siinä missä tilastotietoihin kohdistetaan vain hakuja, käyttäjätietoihin kohdistuu myös lisäämistä, päivittämistä ja poistamista. Periaatteeltaan näiden toimintojen suorittaminen toimii täysin samoin kuin edellisessä luvussa käsiteltiin. Uutta `$resource`-pohjaista palvelua luotaessa määritellään kuitenkin myös toiminnot muillekin kuin `query`-metodille.

Kuten tietokantarakenteesta (luku 4.3.1) käy ilmi, sijaitsevat käyttäjän pelit käyttäjän dokumentin sisällä. Näin ollen olisi oletettavaa, että päivittäminen kohdistettaisiin joka kerta käyttäjään kokonaisuudessaan. Työssä kuitenkin päädyttiin luomaan molemmille, käyttäjätiedoille ja peleille, oma `$resource`. Näihin kohdistettavat muutokset menisivät pitkin omia HTTP-pyyntöjä. Seuraavaksi esitellään molempien palvelut. Käyttäjätietoihin on määritelty näiden hakemisen lisäksi päivittäminen. Haun aikana haetaan myös käyttäjän pelit, ja tästä syystä pelien palvelussa ei ole hakua näille. Pelien palvelussa on kuitenkin määritelty päivittäminen, joka päivittää pelkästään kyseisen pelin tietoja eikä mitään muuta. Tästä löytyy myös uuden pelin lisäykseen tehty POST-pyyntö sekä pelin poistamiseen DELETE-pyyntö. Näille kaikille on määritelty omat käsittelijänsä palvelimella, kuten luvussa 4.4 on käyty läpi.

```
function UserFactory($resource) {
    return $resource("api/user/:username",
        {email: "@username"},
        {
            "get":      { method: "GET" },
            "update":   { method: "PUT" }
        });
}
```

```
function GameFactory($resource) {
    return $resource("api/game/:id/:user",
        {id: "@id", user: "@user"},
        {
            "update":   { method: "PUT" },
            "add":      { method: "POST" },
            "remove":   { method: "DELETE" }
        });
}
```

Näiden palvelujen käyttäminen käy yhtä yksinkertaisesti kuin edellisessäkin luvussa. Seuraavassa esimerkissä tietokannasta haetaan käyttäjätiedot halutulle käyttäjätunnukselle. Saadut tiedot sijoitetaan muuttujaan. Tähän muuttujaan kohdistetut HTTP-metodit suoritetaan suorastaan taian omaisesti oikein. Kutsuttaessa päivitystä eli `$update`-metodia AngularJS lähettää `accountInfo`-muuttujan sisällön palvelimelle, jossa kyseisen käyttäjätunnuksen tiedot päivitetään annettuihin. Muidenkin toimintojen suorittaminen noudattaa samaa yksinkertaista linjaa.

```
User.get({username: user},
    function (data) {
        accountInfo = data;
});

accountInfo.$update();
```

4.5.4 LocalStorage

Kaavioiden selailun ja ulkonäön muokkaamisen helpottamiseksi tietoja kaavioista talletetaan myös selaimesta löytyvään localStorageen. Tämä on helppo tapa säilöä harmitonta tietoa merkkijonoina, jota tarvitaan vähän väliä. Tähän selaimessa säilöttyyn domainkohtaiseen varastoon pääsee helposti käsittelemään seuraavan koodiesimerkin tavoin. Huomaa, kuinka JSON-objektit täytyy muuttaa merkkijonoiksi ennen varastoon tallentamista ja takaisin objektiksi, kun tämän ottaa sieltä käyttöön.

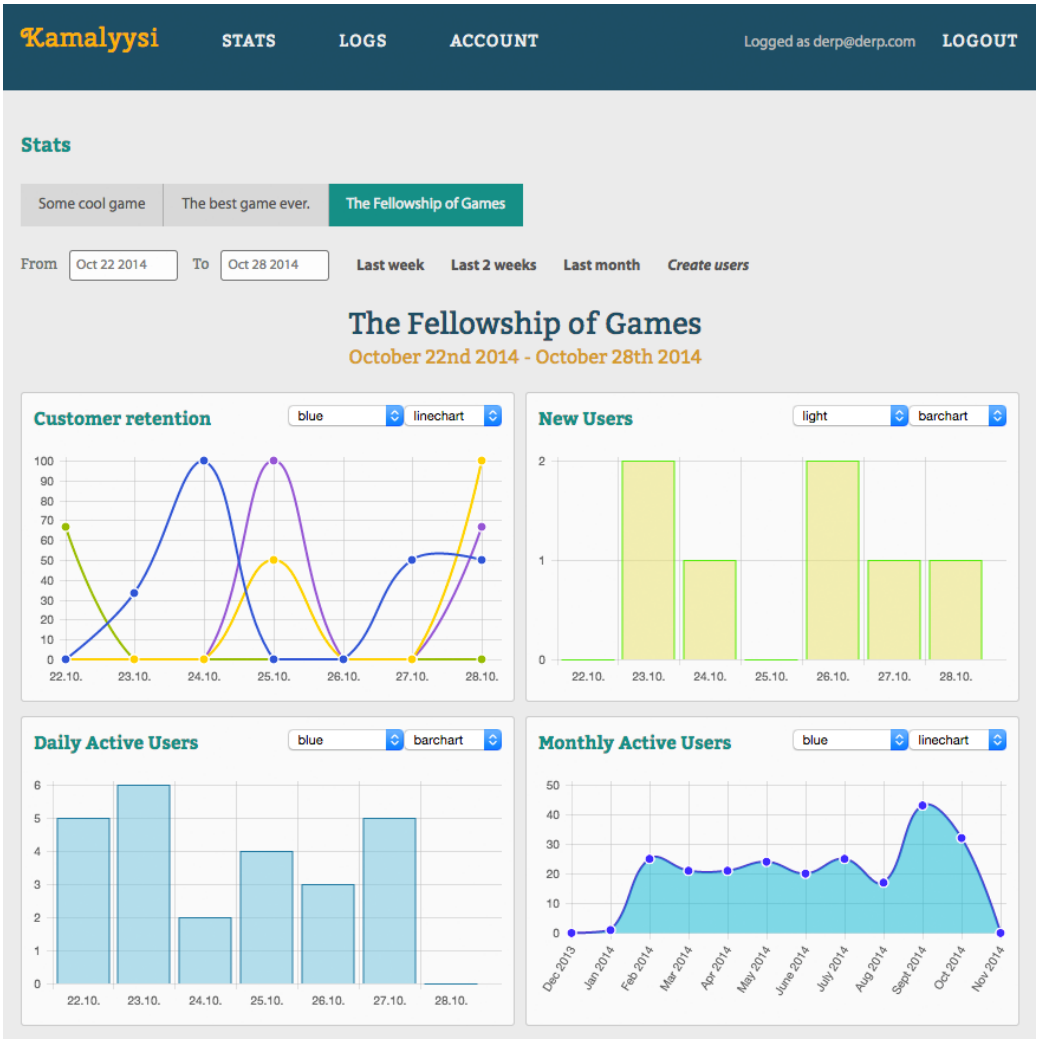
```
localStorage["charts"] = JSON.stringify({ charts: [] });
var charts = JSON.parse(localStorage["charts"]);
```

4.6 Ulkoasu

Nykyisen trendin mukaisesti järjestelmän web-käyttöliittymästä haluttiin tehdä yksinkertaisen, ilmavan ja värimaailmallisesti harmonisen kokonaisuuden omaava paketti. Käyttöliittymän suunnittelun pääpaino on käytettävyydessä, mutta siihen ei kuitenkaan käy-

tettäisi paljoa resursseja. Ulkoasusta ei haluttu tehdä graafisesti raskasta, joten vastuu viihtyvyyden luomiselle oli todellakin väri- ja kirjasinvalinnoilla.

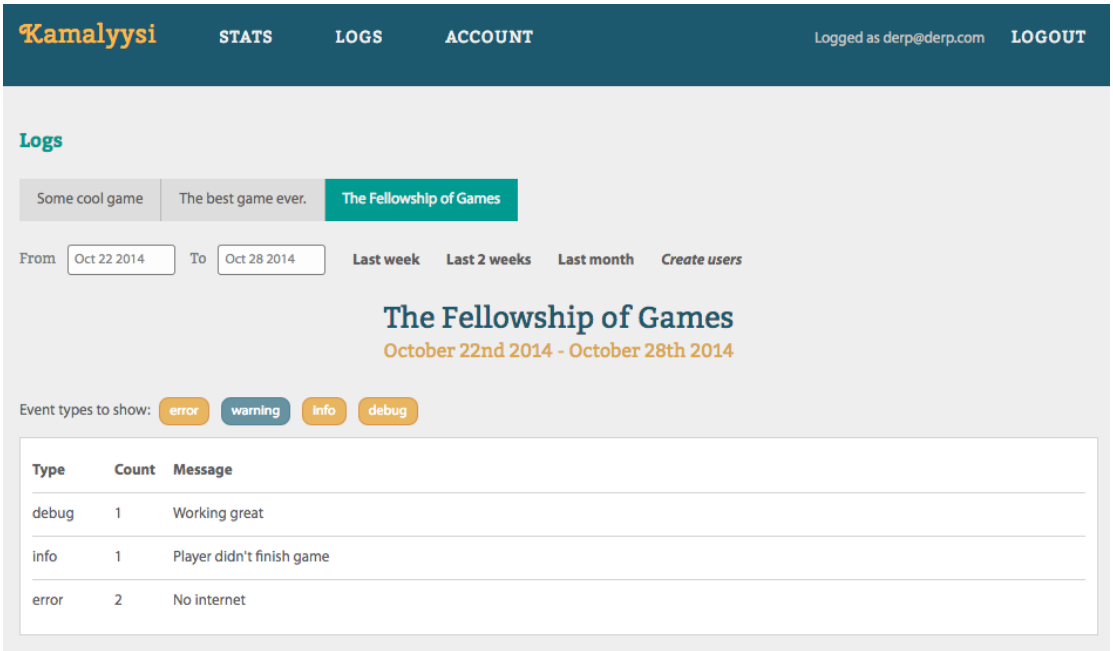
Kuviosta 16 nähdään kuvakaappaus kaavioita sisältävästä sivusta. Kaaviot skaalautuvat ikkunan koon mukaan ja lopulta ovat koko sivun leveitä.



Kuvio 16. Sovelluksen muodostamia kaavioita

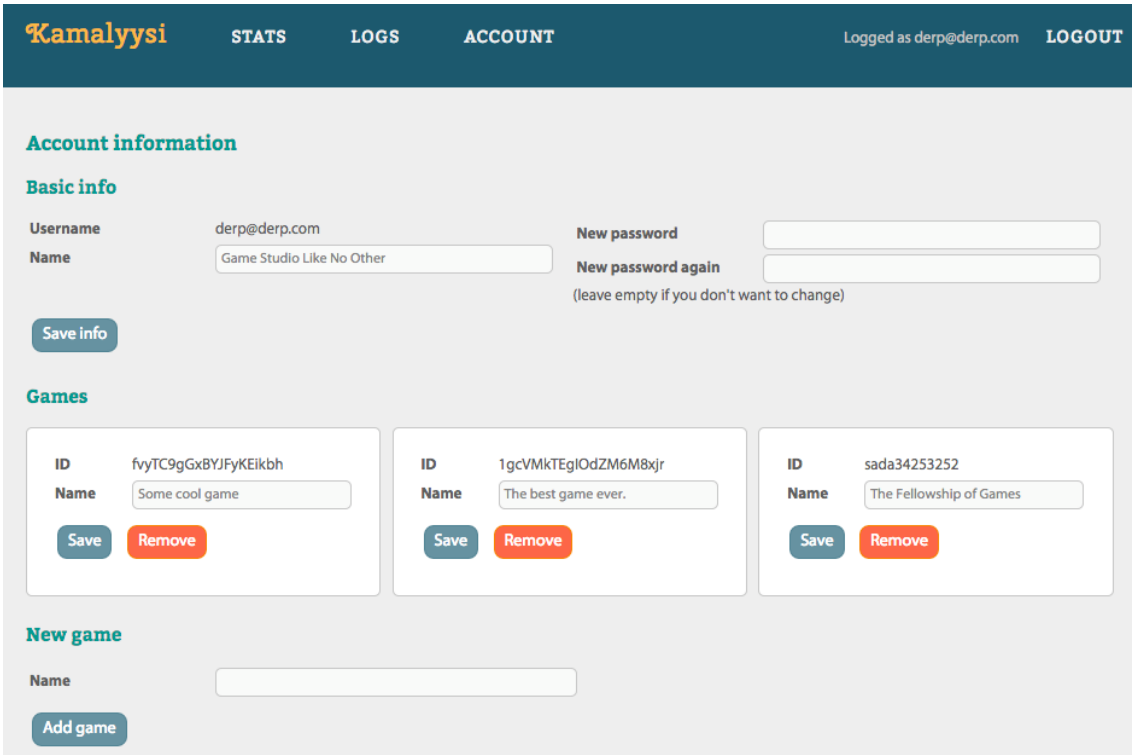
Kaaviosivun ulkoasun suunnittelua helpotti kaaviokirjastoon tullut päivitys, joka mahdollisti kaavioiden määrittämisen responsiivisiksi, jolloin ne muuttavat kokoaan ikkunan pienentyessä. Tämä mahdollistaa kaavioiden tutkimisen niin pienemmilläkin laitteilla, kuten puhelimilla.

Pelin lokeja pystyy selaamaan Logs-näkymässä, ja tapahtumat listataan suodatusten mukaisesti kuten kuviosta 17 huomataan.



Kuvio 17. Sovelluksen luoma tapahtumalistaus

Käyttäjän ja pelien tietoja pystyy muokkaamaan Account-näkymässä (kuvio 18). Koska peleille ei koettu tarpeelliseksi laittaa kuvaa tai muuta tietoa kuin nimi, näyttää pelilistaus hieman karulta kaikessa simppeliydessään.



Kuvio 18. Sovelluksen käyttäjätietosivu

Aluksi ulkoasun toteutuksessa käytettiin Semantic UI -nimistä CSS-kehystä, joka tarjoaa muidenkin samankaltaisten tuotteiden tapaan responsiivisen eli eri kokoisille päätelaitteille skaalautuvan ulkoasuruudukon (grid). Kehys sisältää myös muun muassa valmiiksi muotoillut fontit, painikkeet, lomakkeet, taulukot ja erilaisia kuvakkeita nykypäivän Internetissä käytettäviin palveluihin.

Semantic UI:sta kuitenkin luovuttiin tämän nuoresta iästä johtuvan suppeuden vuoksi. Tilalle otettiin Bootstrap, josta löytyy niinikään kaikki Semantic UI:n ominaisuudet sekä paljon muuta. Osasyy Bootstrapin valintaan oli kaavioiden ajankohdan valintaan tarvittava lisäosa, joka oli tehty toimivaksi AngularJS-Bootstrap-kombinaatiolla.

Ajan saatossa täytyi kuitenkin todeta, ettei Bootstrapin ylikirjoittamat rakenteen muotoilut sovi näin isoja kaavioita omaavaan järjestelmään. Niin Bootstrapista kuin ajankohdan valitsimesta luovuttiin, joista jälkimmäiselle löytyi onneksi korvaaja jQuery-UI:n sisältämästä käyttöliittymäkomponenttivalikoimasta. Lopullisessa toteutuksessa ei ole käytetty mitään valmista CSS-kehystä, sillä sivuston rakenteen ollessa yksinkertainen ja kaavioiden vaatiessa tietynlaista kohtelua tämä olisi ollut aivan turha.

CSS:n luomiseen otettiin kuitenkin mukaan Sass-niminen esiprosessointiohjelma, joka helpottaa CSS:n kirjoittamista huomattavasti. Siinä missä perinteisessä CSS:ssä tyylityksiä ei voi kirjoittaa sisäkkäin porrastaen tai sisällyttää muuttujia kuvaamaan jotain tiettyä väriä, onnistuu tämä kaikki ja moni muukin asia Sass:lla. Kokemusta Sass:sta ei ole aikaisemmin karttunut, joten nyt oli oiva hetki opetella sekin kaiken muun ohessa.

4.7 Unity-plugin

Vaikka sinänsä järjestelmä ei ole mitenkään riippuvainen tietystä pelimoottorista, toteutettiin tietojen keräysjärjestelmä Unityyn asennettavaan pluginin avulla, johon sisällytettiin kaikki välttämättömimmät ominaisuudet. GameAnalyticsin ja Lumosin tuotteet ovat aivan omaa luokkaansa, sillä näistä molemmista löytyy lukuisia tiedostoja, joihin tutustuminen oli hankalaa. Tuotteista löytyi totta kai graafiset editorit tehtäville muutoksille sekä ohjeita.

Lopulta päädyttiin varsin yksinkertaiseen, mutta toimivaan ratkaisuun, joka koostuu yhdestä ainoasta tiedostosta. Tämä tiedosto täytyy liittää johonkin pelissä olevaan komponenttiin, useimmiten pelin pääkameraan. Web-sivujen kautta lisätään peli, jolle järjestelmä generoi yksilöllisen ID:n. Tämä saatu merkkijono syötetään komponenttiin liitettyyn tiedostoon ja lähetetään kerättävien tietojen mukana palvelimelle jatkossa, jotta tilastot kirjataan oikealle pelille.

Järjestelmään on oletuksena sisällytetty istuntojen lähetys, joten pelinkehittäjän ei tarvitse erikseen alkaa sijoittamaan minnekään istuntojen tallettamista. Joka kerta, kun peli sammutetaan, muodostetaan JSON-muotoinen merkkijono istunnon ajankohdasta ja pituudesta, sekä pelaajan ja pelin ID:istä. Merkkijono lähetetään sitten käyttämällä Unityn omaa WWW-luokkaa, joka mahdollistaa HTTP-pyynnöt. Istunnon lähetys suoritetaan seuraavanlaisesti:

```
Kamalyysi.instance.SendSession();
```

Pelinkehittäjän harteille jää metodien sijoittelu oikeisiin paikkoihin sovelluksessa. Muun muassa seuraavanlaisilla metodeilla voidaan lähettää merkintöjä pelin tapahtumalokiin. Oleellinen asia tapahtuman tallentamisessa on tietenkin tapahtumaa kuvaava viesti. Tapahtuman voikin tallentaa pelkästään joko tämän viestin perusteella tai sitten lisäämällä tapahtumaan mukaan jonkun ennalta määrätyn koodin, jos näin tahtoo.

```
Kamalyysi.instance.SendError("No internet");
Kamalyysi.instance.SendWarning("No highscores", "1337");
Kamalyysi.instance.SendInfo("Player didn't finish game");
```

Metodit päätyvät järjestelmän sisälle, josta nämä lajitellaan eteenpäin kohti palvelimelle lähetystä. Istuntojen tapaan näistä muodostetaan JSON-muotoinen merkkijono, johon sijoitetaan tapahtuman tyypin, viestin ja koodin lisäksi myös ajankohta sekä pelaajan ja pelin ID:t.

Jokainen laite, jolla Unityä voidaan käyttää, omaa uniikin laitetunnuksen. Tämä yksilöi laitteet, ja sitä käytetäänkin istuntojen ym. tallentamisessa oikeille pelaajille.

Järjestelmään on määritelty osoite, jossa palvelin vastaanottaa HTTP-paketteja ja aloittaa näiden jatkokäsittelyn, kuten luvussa 4.4.1 on kuvattu.

4.8 Testaus

Kuten luvussa 4.7 jo käytiinkin läpi, testattiin järjestelmän tietojen keräystä pelaajilta Unity-pohjaisen pelin avulla. Alkuperäisistä suunnitelmista poiketen järjestelmän toimivuutta ei testattukaan aivan oikeilla mobiililaitteilla, sillä tietokoneella pyörivä Unity-peli käyttäytyy samalla tavalla kuin mobiililaitteessa. Pelin kääntämisestä erilliselle laitteelle olisi ollut paitsi työlästä lisenssimaksujen takia niin myös varsin turhaa, sillä mitään uutta ja olennaista ”oikeat” mobiilipelit eivät tuo.

Järjestelmää testattiin aluksi generoimalla AngularJS:n puolella testidataa arpomalla käyttäjille tunnuksia, istuntojen pituuksia ja ajankohtia. Mitään sen suurempia tarkastuksia ei tietojen oikeellisuuksista tai päällekkäisyyksistä ei tehty, sillä tärkeintä oli saada tietoja, joista suodattaa oikeat käyttöön ja muodostaa kaavioita. Kun kaavioiden muodostaminen oli saatu kuntoon, toteutettiin Unityyn plugin, joka noudatti samaa pelaajien lisäyksen suhteen kuin aikaisempi massagenerointi. Pluginin myötä täytyi kuitenkin luoda jo olemassa olevien pelaajien päivittäminen, sillä tälle ei ollut aikaisemmin ollut tarvetta.

Järjestelmän käyttöliittymän testaukseen käytettiin yrityksen työntekijöitä. Näiltä saikin hyvää tietoa siitä, onko käyttöliittymä selkeä ja helppokäyttöinen. Kaavioihin toivottiinkin enemmän muokkausmahdollisuuksia kaaviotyypin ja värimaailman suhteen, mikä oli helppo toteuttaa.

5 Työn tulokset

5.1 Yhteenveto

Opinnäytetyön aikana tehtyyn järjestelmään ei voi mielestäni olla muuta kuin tyytyväinen. Tie GameAnalyticsin kokoiseksi vaikuttavaksi pelien analysointityökaluksi olisi hyvin pitkä eikä suinkaan vain yhden opiskelijan kuljettavissa. Tämän kokoluokan järjestelmää ei opinnäytetyössä kuitenkaan lähdetty toteuttamaan, vaan enemmänkin perusominaisuuksiltaan helppokäyttöistä ja yksinkertaista järjestelmää.

Koska lähestulkoon jokainen teknologia ja työväline tuli uutena ja tunnettujenkin käyttöä piti syventää, ovat tulokset yllättävänkin paljon odotusten mukaiset. Järjestelmällä on selkeä web-käyttöliittymä, jonka käyttö on helppoa ja selkeää ilman ylimääräisiä hämmentäviä tilastotietoja. Mihinkään suureen peliprojektiin tai enemmän asiaan perehtyvälle pelistudiolle en luonnollisesti tätä ensimmäisenä suosittelisi. Järjestelmässä on vielä paljolti kehitettävää ja kohennettavaa, kuten luvusta 5.3 tulee selviämään.

AngularJS:n osaamista oli karttunut jo aikaisemmin, mutta vasta opinnäytetyötä tehdessä asioihin tuli perehdyttyä kunnolla ja oikein, kuten myös MongoDB:n kanssa. Go:lla työskenteleminen oli paikka paikoin tuskaista, mutta muuten oikein opettavaista ja ensi kerralla osaa jo varoa tiettyjä kuoppia. Opinnäytetyö tarjosi siis oikein loistavat puitteet uusien teknologioiden opiskeluun.

5.2 Ongelmakohdat

Järjestelmän kokoamisen aikana vastaan tuli todella paljon ongelmia, sillä käytössä olevat teknologiat olivat täysin uusia tai vähemmän tuttuja. Näinpä ollen tekovaiheessa joutuivat erinäiset oppaat, hakukoneet sekä keskustelufoorumit kovalle käytölle.

Eniten ongelmia oli Go:n kanssa. Tämän omalaatuinen syntaksi poikkeaa kaikista aiemmin käyttämistäni kielistä, joten tämän ymmärtäminen vei oman aikansa. Eniten hidastuksia tuli structien kanssa. Kantapään kautta tuli muun muassa opittua structien käyttö perin pohjin, sekä varsinkin näiden käyttö asiakaspuolelta tulleiden JSON-objektien rinnalla. Go myös vie tyypittämisen ja syntaksin aivan omalle tasolleen niiden tiukkuudessa, sekä tämä taitaa myös mystisten virheilmoitusten antamisen. Teon aikana taakkana oli myös paljon muita pieniä ongelmia, jotka johtuivat vain uuden kielen aiheuttamasta tietämättömyydestä.

MongoDB:n käytöstä oli kokemusta aikaisemmin Node.js:n rinnalla käytettynä, mutta Go:n mgo-kirjasto oli kuitenkin sen verran helppo opiskella, ettei ongelmia liiemmin ollut. Oikean ajankohdan tietojen hakeminen tietokannasta tuotti ongelmia, sillä Go:hon saapuvien päivämäärien ja kellonaikojen muuntaminen tapahtui jostain syystä väärin. Paljon aikaa tuhlautui siihen, kun joutui miettiä, miksi tulokset ovat järjestään edelliseltä

päivältä. Syy löytyi kuitenkin aikavyöhykkeistä, Go nimittäin muuttaa tulevat kellonajat UTC:hen eli 3 tuntia taaksepäin, tarkoittaen että keskiyöllä tehdyt istunnot merkattiin Go:n oma-aloitteisuuden vuoksi edelliselle päivälle. Tämä korjattiin lisäämällä tuotuihin ajankohtiin aikavyöhykkeet, jotta Go ei tekisi omia johtopäätöksiään.

MongoDB:n aggregaatioiden muodostaminenkaan ei ole kovin helppo työ, mutta se on kuitenkin niin loogista ja samaa kaavaa noudattavaa, että siihen löytyi paljon neuvoja joita soveltaa.

Chart.js:n avulla tehdyissä kaavioissa ei ollut aluksi mukana nk. tooltip-ominaisuutta, jolla saa sarakkeiden arvot näkymään pienessä laatikossa luvussa 3.4 nähdyn esimerkin tavoin. Ominaisuus lisättiin vasta kirjaston käyttöönoton jälkeen, kun projekti oli jo hyvässä vauhdissa kasaantumassa. Kirjaston päivitys ei kuitenkaan sujunut ongelmitta, sillä tooltippien tulon myötä tilastojen päivittäminen vaikeutui. Enää uuden ajankohdan myötä saaduilla tiedoilla ei voinut korvata vanhoja tietoja, sillä kirjaston luomat kuuntelijat hiirelle jäivät elämään ja näin ollen näyttivät vanhoja tietoja kun hiirtä vei kaavion päällä. Tämän korjaamiseksi koko kaavioiden rakennustapaa piti muuttaa dynaamiseksi, eli hyödynnettiin AngularJS:n `ngRepeat`-direktiiviä. Tällä kertaa kirjaston luoma `canvas` luodaan aina uusiksi, kun sovelluksen `$scope`-muuttujaan sidottuja tietoja muutetaan. Sinänsä muutos oli hyvä, vaikka vaatikin paljon koodin uudelleen kirjoittamista.

Direktiivejä luotaessa tuli myös todettua se, että direktiiviin määriteltyyn templateen, eli direktiivin pohjaan, ei voi määritellä toisen direktiivin nimeä muuttujan avulla. Edellisessä kappaleessa kuvatun `ngRepeat`-direktiivin hyödyntäminen nimittäin tarkoitti sitä, ettei Angles.js-kirjaston vaatimia `canvas`-elementtejä voinut muodostaa lennosta ilman välissä olevaan direktiiviä. Ratkaisuksi luotiin direktiivi `ngRepeat:`n sisään, joka palauttaa määritellyn tyypin mukaisen oikean pohjan, jonka avulla Angles.js voi muodostaa Charts.js-kaavioita.

5.3 Jatkokehitys

Kaikkien projektien tapaan myös tästäkin järjestelmästä löytyy kosolti kehitettävää. Suurin puutos järjestelmässä on kuitenkin kustomoitavien hakujen suhteen. Alun perin näi-

den tekeminen piti mahdollistaa, mutta koko järjestelmän kasaamisen vaatima työ tuli yllätyksenä. Tämän tyyppisten hakujen tulisi kuitenkin olla olemassa, sillä näillä pelien kehittäjät saavat kuitenkin arvokasta tietoa juuri siitä omasta pelistä, eli vaikkapa siitä, kuinka paljon ja mitä aseita pelaajat käyttävät peleissä.

Alun perin oli myös tarkoituksena toteuttaa tietojen lähetys Unity-pluginista websocket-tien kautta. Näin olisi voinut kätevästi vain avata websocket-portti, johon plugin ottaisi kerran pelin käynnistämisen alussa yhteyttä, ja sen jälkeen vain helposti ja nopeasti lähettäisi pelitietoja palvelimelle. Ominaisuus todettiin kuitenkin turhaksi näin suppeaan järjestelmään, ja tämän toteuttaminen Unityssä ei olisi ollut niin helppoa. Nyt kaikki tieto kulkee REST:n läpi HTTP-pyyntöinä. Websocket toisi kuitenkin omat hienot nopeutensa mukanaan.

Jos järjestelmällä on tulevaisuutta, vaihdettaisiin Chart.js-kirjaston tilalle Highcharts-kirjasto, jota käsiteltiin luvussa 3.4. Vaikka tämä on maksullinen suuremmalla käytöllä, on tämän käyttö ja ominaisuudet kuitenkin paljon monipuolisempia ja vakaampia kuin Chart.js:n.

Luonnollisesti osa järjestelmän tulevaisuutta olisi kirjastojen ja teknologioiden versioiden päivittäminen.

6 Kerätyn tiedon hyödyntäminen

6.1 Pelaajien hankinta

Kiinnostavin asia pelaajien tutkimisessa lienee luonnollisesti se, kuinka moni pelaa ja kuinka usein. Varmasti jokainen pelinkehittäjä haluaa luoda pelejä, jotka saavuttavat mahdollisimman monia ihmisiä eri ikäluokissa ja elämäntilanteissa.

Selvittämällä pelaajien asuinmaan voidaan pohtia, miksi peli on suosittu juuri kyseisessä tilastojen kärjessä olevassa maassa, mutta jossain muussa maassa tätä on hädin tuskin asennettu kertaakaan? Näihin maihin voidaan kohdistaa kulttuuriin sopivampaa mainontaa. Lisäksi voidaan myös miettiä, onko pelin sisällössä jotain maan kansalaisia pois työntävää. Uskonnollisia viittauksia, väkivaltaa, eläimiä tai huumoria? (Benefits 2014.)

Monet suositutkin pelit joutuvat kokemaan tietynlaisia ”siistimisiä” sopiakseen muiden maiden kulttuureihin. Tällaista sensuroimista harvoin näkee mobiilipeleissä, sillä nämä ovat luonteeltaan come-and-go-tyylisiä, eli nämä on helppo asentaa laitteeseen, mutta vielä helpompi poistaa. Konsoli- ja PC-peleissä eri maiden versioihin tehdään luonnollisesti muutoksia jo kielenkin takia, joten tiettyjen hahmojen ja tapahtumien korvaaminen eri kulttuureille sopivimmiksi on luonnollista. (Custer 2014.)

Pelaajien kotimaiden lisäksi on myös erittäin kiintoisaa tietää, missä vaiheessa pelin elämänkaarta pelaajia saadaan kaikista eniten. Herättävätkö peliin tehdyt pienetkin päivitykset ja korjaukset huomiota pelaajayhteisössä? Opinnäytetyöhön ei sisällytetty maanosien saatikka versionumeroiden tarkkailua, sillä näitä ei koettu tarpeellisiksi. (Benefits 2014.)

Varsinkin mobiilipelien kohdalla on hyvä punnita pelin julkaisemista eri käyttöjärjestelmille. Vaikka peli ei saisi paljoa huomiota jossain toisessa marketissa, voi se toisessa olla oikea hitti. Menekki riippuu paljolti tarjonnan määrästä ja laadusta, sekä tietenkin mainostamisesta.

6.2 Pelaajien sitoutuminen

Oleellinen osa pelaamista on tietenkin sen toistaminen. Peleistä pyritäänkin tekemään mahdollisimman koukuttavia ja mielenkiintoisia, jotta pelaajat eivät hyppäisi kilpailevien pelien kelkkoihin. Tuskin on olemassa montaa peliä, jotka olisi tarkoitettu vain yhdellä istunnolla pelattavaksi. Tämän takia sitoutumista voidaan tarkkailla mm. retentionin avulla, jonka laskutapa työn osalta käytiin läpi luvussa 4.4.3. Peleihin pätee samat säännöt kuin muihinkin tuotteisiin, joille pyritään löytämään kuluttajia: tuotteen tarjoajalle tulee kalliimmaksi hankkia uusia käyttäjiä kuin pitää vanhat tyytyväisinä. (Benefits 2014.)

Oletettavasti jokaisen tuotteen käyttäjien kohdalla pystytään näkemään tietynlainen alkuinnostus. Hyvässä lykyssä tämä kestää pitkäänkin, mutta valitettavasti tämä voi päättyä jo ensimmäisen pelikerran jälkeen kun pelaaja hylkää tuotteen. Tällaisen toiminnan seuraamista voidaan tutkia istuntojen määriä kuvaavalla kaaviolla, joka erittelee käyttäjien lukumääriä eri istuntolukumäärille. Kukaan pelinkehittäjä tuskin haluaa nähdä

valtavaa piikkiä yhden istuntokerran kohdalla, sillä tämä herättää eritoten kysymyksen ”miksi?”. Oliko peli liian vaikea, helppo, rasittava, ruma, tökkivä, tylsä, yksipuolinen tai vain yksinkertaisesti huono? Valitettavasti tällaisiin kysymyksiin on vaikea saada vastausta ilman erillisen palautelomakkeen työntämistä pelaajille. (Mt.)

6.3 Taloudellinen hyöty

Lähtökohtaisesti pelinkehittäjä lisää peliinsä analysointijärjestelmän rahan kiilto silmissä. Harvassa ovat pelit, joilla ei tähdätä rahoiksi lyömiseen ja maineen kartuttamiseen. Yleensä myös ilmaispeleihin lisätään mainoksia, joista saa muutamia senttejä näyttökertaa kohden. Nämä mainokset voidaankin melkein aina ostaa pois sopusuhtaiseen hintaan.

Vaikka työssä ei toteutettu mitään rahan käyttöön viittaavia kaavioita, ovat ne silti olennainen osa etenkin mikromaksuilla toimivan mobiilipelin seurantaa. Kaavioidaan perusteella voisi päätellä muun muassa sen, kuinka kauan pelaajalta menee rekisteröitymisestä ennen kuin tämä käyttää rahaa pelissä. Kiinnostavaa olisi myös tietää, saako joku tietty kenttä pelissä pelaajia käyttämään enemmän rahaa. Tuollaisessa tilanteessa voisi alkaa punnita, onko kenttä liian haastava vai muut liian helppoja? (Mt.)

On myös ikävä tosiasia, että välillä peleihin pääsee luikahtamaan sellaisia virheitä, jotka mahdollistavat vakavankin väärinkäytön. Taannoin eräs pelinkehittäjä oli unohtanut poistaa mobiilipelistään testitarkoitukseen käytetyn ominaisuuden, jolla avattiin pelin kaikki kentät ilman, että tästä maksettiin mitään (Reynolds 2014). Tällaisten virheiden korjaaminen on tietenkin erittäin tärkeää, ja muiden vastaavien madonreikien löytämisen voi tehdä kaavioita tutkimalla. Jossakin pelin kentässä voisi olla mahdollista huijata ja läpäistä kenttä reilusti alle oletetun ajan, jolloin tämä näkyisi selvästi kenttien läpäisyaikoja kuvaavassa kaaviossa. (Mt.)

Mahdollisien virheiden varalta näihin on hyvä varautua edes pitämällä näistä lokia, kuten työssäkin toteutettiin. Odottamattomia virheitä voi ilmetä, kun pelaajakunnasta löytyy paljon erikokoisia, -mallisia ja käyttöjärjestelmiltään erilaisia älylaitteita.

7 Pohdinta

Kaiken kaikkiaan opinnäytetyön tekeminen oli oikein opettavainen kokemus, josta jäi paljon käteen tulevaisuutta varten, kuten luvussa 5.1 jo todettiinkin työn toteutuksen suhteen. Teknologiavalinnat olivat nykyaikaisia ja hyvin tehtyjä, sillä näen käyttäväni niitä jatkossakin. Tavoitteet täyttyivät niin teknisten taitojen opiskelussa kuin tietämyksen kartuttamisessa.

Analysoimisen tärkeys korostui työtä varten tehdyn taustatutkimuksen aikana. Mielestäni millään pelillä tai muullakaan digitaalisella palvelulla, kuten www-sivulla, ei ole mitään hyvää syytä jättää analysointijärjestelmää pois. Tällaisia palveluja käyttämällä saadaan helposti tietoa siitä, onko sovellus terve ja onko tällä tulevaisuutta nykytahdilla. Järjestelmien käyttöönotto on erittäin helppoa ja näistä saatava hyöty tarjoaa todella hyvän hinta-laatusuhteen, sillä useat analysointityökalut ovat ilmaisia tavalliselle käyttäjälle.

Eri asioiden analysointia on harrastettu jo pitkään paljon ennen mobiili- tai edes konsolipelien saapumista. Vaikkakin esimerkiksi tekstien analysoimisella tarkoitetaan eriä, kuin mitä pelaajien analysoimisella, on molemmissa periaate sama: ymmärtää, mitä kohde tekee ja miksi. Pelaajien analysoimisella tähdätään luonnollisesti pelin pidemmän elinkaaren saavuttamiseen, mutta myös pelikokemuksen parantamiseen käyttäjän näkökulmasta. Pelien analysointia tullaan harrastamaan niin pitkään kuin pelejä tekee useampi kuin yksi taho.

Lähteet

AngularJS. 2014. AngularJS-sivusto. Viitattu 16.7.2014. https://angularjs.org/

App Store (iOS). 2014. Wikipedia-sivusto. Viitattu 3.11.2014. http://en.wikipedia.org/wiki/App_Store_(iOS)

Benefits. 2014. GameAnalytics-sivusto. Viitattu 31.10.2014. http://www.gameanalytics.com/benefits.html

Comparison of JavaScript charting frameworks. 2014. Wikipedia-sivusto. Viitattu 15.10.2014. http://en.wikipedia.org/wiki/Comparison_of_JavaScript_charting_frameworks

Controllers. 2014. AngularJS Developer Guide -sivusto. Viitattu 16.7.2014. https://docs.angularjs.org/guide/controller

Custer, C. 2014. Here's how Diablo 3 will be censored in China. Viitattu 31.10.2014. http://www.gamesinasia.com/diablo-3-china-censorship/

Directives. 2014. AngularJS Developer Guide -sivusto. Viitattu 16.7.2014. https://docs.angularjs.org/guide/directive

Documents. 2014. MongoDB Manual -sivusto. Viitattu 13.6.2014. http://docs.mongodb.org/manual/core/document/

Drachen, A. 2012. Third Party Analytics: what are the options? Blogikirjoitus GameAnalytics-sivustolla. Viitattu 26.8.2014. http://blog.gameanalytics.com/blog/third-party-analytics-what-are-the-options.html

Drachen, A. 2013. Heatmapping. Blogikirjoitus GameAnalytics-sivustolla. Viitattu 13.6.2014. http://blog.gameanalytics.com/blog/heatmapping.html

FAQ. 2014. AngularJS-sivusto. Viitattu 16.7.2014. https://docs.angularjs.org/misc/faq

Filters. 2014. AngularJS Developer Guide -sivusto. Viitattu 6.10.2014. https://docs.angularjs.org/guide/filter

Flurry Analytics. 2014. Flurry-sivusto. Viitattu 26.8.2014. http://www.flurry.com/solutions/analytics

Google Analytics. 2014. Wikipedia-sivusto. Viitattu 25.8.2014. http://en.wikipedia.org/wiki/Google_Analytics

Go vs Node.js for servers. 2014. Reddit-palvelussa käyty keskustelu. Viitattu 16.5.2014. http://www.reddit.com/r/golang/comments/1ye3z6/go_vs_nodejs_for_servers/

Hows, D., Plugge, E., Membrey, P. & Hawkins, T. 2013. Definitive Guide to MongoDB. 2. p. New York: Apress.

Kuosmanen, H. 2009. Go on Googlen uusi ohjelmointikieli. Artikkeli mBnet-sivustolla. Viitattu 8.10.2014. http://www.mbnet.fi/artikkeli/ajankohtaiset/ajassa/go_on_googlen_uusi_ohjelmointikieli

McCalmont, T. 2013. How Do I Know I Have a Healthy Game? Blogikirjoitus Gamasutra-sivustolla. Viitattu 12.10.2014. http://www.gamasutra.com/blogs/TrevorMcCalmont/20130228/187460/

Mitä on web-analytiikka? N.d. Analytics-sivusto. Viitattu 12.6.2014. http://www.analytics.fi/mita-on-web-analytiikka/

MongoDB Limits and Tresholds. 2014. MongoDB Manual -sivusto. Viitattu 28.8.2014. http://docs.mongodb.org/manual/reference/limits/

mux. N.d. Gorilla web toolkit -sivusto. Viitattu 20.10.2014. http://www.gorillatoolkit.org/pkg/mux

Overview. 2014. Lumos-sivusto. Viitattu 26.8.2014. https://www.lumospowered.com/powerups/analytics/overview

Papa, J. 2014. AngularJS Style Guide. GitHub-sivusto. Viitattu 24.10.2014. https://github.com/johnpapa/angularjs-styleguide

Perform Two Phase Commits. 2014. MongoDB Manual-sivusto. Viitattu 28.8.2014. http://docs.mongodb.org/manual/tutorial/perform-two-phase-commits/

Production Deployments. 2014. MongoDB-sivusto. Viitattu 12.6.2014. http://www.mongodb.org/about/production-deployments/

Retention Analysis. N.d. MixPanel-sivusto. Viitattu 27.8.2014. https://mixpanel.com/retention/

Reynolds, B. 2014. How I accidentally gave away my game for free. Blogikirjoitus Gamasutra-sivustolla. Viitattu 3.11.2014. http://gamasutra.com/blogs/BenReynolds/20140809/222987/How_I_accidentally_gave_away_my_game_for_free.php

Sathish VJ. 2011. Blogikirjoitus GoLang Tutorials -sivustolla. Viitattu 10.10.2014. http://golangtutorials.blogspot.fi/2011/06/structs-in-go-instead-of-classes-in.html

Scopes. 2014. AngularJS Developer Guide -sivusto. Viitattu 16.7.2014. https://docs.angularjs.org/guide/scope

Sharding Introduction. 2014. MongoDB Manual -sivusto. Viitattu 28.8.2014. http://docs.mongodb.org/manual/core/sharding-introduction/

Silver, L. 2014. Angles.js. GitHub -sivusto. Viitattu 28.10.2014. https://github.com/gonewandering/angles

Sommer, T. 2014. User Retention: Yes, But Which One? AppLift-verkkosivut. Viitattu 31.10.2014. http://www.applift.com/blog/user-retention.html

Taras, G. 2014. Funnels use cases. GameAnalytics-sivusto. Viitattu 27.8.2014. http://support.gameanalytics.com/hc/en-us/articles/200840856-Funnels-use-cases

Todor, A. 2014. Core GameAnalytics metrics. GameAnalytics-sivusto. Viitattu 26.8.2014. http://support.gameanalytics.com/hc/en-us/articles/200841006-Core-GameAnalytics-metrics