Haaga-Helia
ammattikorkeakoulu Oy

Lyömäsoitinten nuotinnus AngularJS ja D3 -ohjelmilla

Mikko Luukkonen

Opinnäytetyö

Tietojenkäsittelyn koulutusohjelma

2016

Tekijä(t)
Mikko Luukkonen

Koulutusohjelma
Tietojenkäsittelyn koulutusohjelma

Opinnäytetyön otsikko
Lyömäsoitinten nuotinnus AngularJS ja D3 -ohjelmilla

Sivu- ja liitesivumäärä
72 + 19

Opinnäytetyön otsikko englanniksi
Notation app for percussions with AngularJS and D3

Tässä opinnäytetyössä toteutettiin länsiafrikkalaisten dundun- ja djemberumpujen selainpohjaisen nuotinnussovelluksen prototyyppi. Työ tehtiin käyttämällä kahta teknologiaa: AngularJS ja D3.js. Angular on Googlen ylläpitämä sovelluskehitysalusta joka on suunniteltu yksisivuisten sovellusten kehitystyökaluksi. D3 on JavaScript –kirjasto datan visualisointiin.

Työn teoreettinen osa jakaantuu kahteen osioon. Ensimmäisessä osuudessa esitellään sovelluksen tukemat instrumentit ja joitain rytmiikan peruselementtejä. Tarkoituksena on antaa tarvittava pohjatieto nuotinnuksen ymmärtämiselle. Toisessa osuudessa tutkitaan käytettyjä teknologiota tarkemmin, Angularin toimintaa, kuinka sen parhaita ominaisuuksia on tarkoitettu käytettävän, ja sen tärkeimpiä sovelluskomponentteja. Toisekseen havainnollistetaan D3:n kykyä sitoa dataa visuaalisiin HTML –elementteihin ja luoda näiden välille vuorovaikutusta.

Operationaalisessa osuudessa esitellään suunnitelma ja arkkitehtuuri sovelluksen toteuttamiseksi, jonka jälkeen sovelluksen toteutusta havainnollistetaan lukuisin konkreettisin koodiesimerkein. Angularin ja D3:n yhteistoiminnasta annetaan konkreettinen esimerkki.

Opinnäytetyön päätöskappaleessa reflektoidaan toteutuksen teon vaiheita, peilataan tulosta ja opittuja asioita asetettuihin tutkimuskysymyksiin sekä ruoditaan sovelluksen potentiaalia, puutteita ja jatkokehitysnäkymiä.

Asiasanat
AngularJS, D3, Single Page Application, sulkeuma, nuotinnus

Sisällys

1 Johdanto

Opinnäytetyön tavoitteena on toteuttaa verkkoselaimessa toimiva perkussioinstrumenteilla soitettavan rytmin nuotinnusohjelman prototyyppi. Sovelluksella voidaan tallentaa jokaisen rytmissä käytetyn instrumentin soittama rooli, toisin sanoen niiden nuotit. Idea sovelukseen syntyi omasta rumpujensoittoharrastuksesta. Web-pohjaisena nuotteja on myös mahdollistaa jakaa tehokkaammin.

Opinnäytetyön teoriaosuudessa perehdytään angularin perusarkkitehtuuriin jonkin verran niiltäkin osin, mikä ei välittömästi ole projektin toteutuksen kannalta välttämätöntä. Tarkoitus on antaa lukijalle käsitys angularin perustoimintamekaaniikasta ja miten se laajentaa web-selaimen toiminnallisuuksia. Lisäksi D3 –kirjastosta kerrotaan asiat jotka itse opinnäytetyön tuotoksena syntyvän ohjelman ymmärtämisen kannalta on välttämätöntä. Ohjelma toteutetaan JavaScript –sovelluskehitystyökalulla AngularJS ja JavaScript –kirjastolla D3.js. Olennaisimmat sovelluksen osat on toteutettu D3 –kirjastoa apuna käyttäen.

Kyseessä on toiminnallinen opinnäytetyö, tuotoksena on sovelluksen prototyyppi. Mikäli sovellus osoittautuu jatkokehityskelpoiseksi, voi sen saattaa kenen tahansa Länsiafrikkalaisen rumpumusiikin nuotinnuksesta kiinnostuneille.

Toiminnallinen tavoitehakuisuus heijastuu jonkin verran lähteiden käyttöön: monin paikoin lähdeviittausten tarkoitus on vain esittää lähde, josta informaatio jonkin toiminnallisuuden ominaisuuden toteuttamiseksi on haettu. Lähteinä käytetään jonkin verran – sen yleisestä kyseenalaisuudesta huolimatta - myös wikipediaa. Toisin kuin esim. yhden kirjoittajan kirjoittamat artikkelit, on wikipedia kuitenkin useimmiten ainakin jollain tasolla vertaisarvioitu.

1.1 Rajaukset

Rytmien tuottamista äänellisesti ei toteuteta. Ohjelmalla ei ole mahdollista esittää instrumentin ääniä sävelasteikolla, vaan ainoastaan ennalta määrättyä rajoitettua kullekin instrumentille ominaista lyöntivalikoimaa. Instrumenttien määrä rajataan kahteen: djembe ja dundun. Kiinnitetään kuitenkin huomiota sovelluksen skaalautumiseen – uusien instrumenttien lisäämisen tulee olla mahdollisimman helppoa. Sovellus kykenee näyttämään tietynrakenteisessa json –tiedostossa esitetyn rytmin valmiin nuotinnuksen, ja tallentamaan muutetun rytmin. Uuden rytmin luontia ei toteuteta. Sovellus toteutetaan yksinomaan JavaScript –kieltä ja valmiita JavaScript –kirjastoja käyttäen.

1.2 Opinnäytetyön tutkimuskysymykset:

- Rytmin esittäminen selkeästi ja intuitiivisesti? Miten rytmin nuotinnuksen esittäminen toteutetaan?
- Miten luoda sellainen modulaarinen rakenne ohjelmalle, jossa uusia ominaisuuksia pystyy mahdollisimman helposti lisäämään ohjelmaan?
- Miten AngularJS ja D3 soveltuvat yhdessä datan visuaaliseen esittämiseen? Millainen on tarkoituksenmukainen vastuunjako niiden kesken?

1.3 Käsitteet

API (Application Programming Interface)

Objektin julkinen ohjelmointirajapinta, jonka kautta objektin kommunikointi ohjelman muiden palasten kanssa tapahtuu.

DDO (Directive Definition Object)

Direktiivin palauttama objekti, jonka sisälle koodataan direktiivin toiminta.

Deklaratiivinen

Määrittelevä, ei yksityiskohtaista logiikkaa (=ohjelmointia) sisältävä tapa ilmaista mitä halutaan tehdä. Deklaratiiviset käskyt delegoivat toteutuksen jollekin toiselle taholla.

Dependency Injection (DI)

Annettava riippuvuus. Vältetään sen objektin luonti jota käytetään.

Direktiivi

Angular komponentti. Upotetaan HTML koodiin deklaratiivisesti. Itse luoduissa direktiiveissä direktiivin toiminto koodataan omaan elementtiinsä imperatiivisesti.

D3 Datan visualisointiin tarkoitettu JavaScript –kirjasto.

Imperatiivinen

Yksityiskohtaisella ohjelmalogiikalla (koodilla) toteutettu.

JSON

Webissä yleisin tiedonvälitykseen käytettävä yksinkertainen avoimen standardin tiedostomuoto.

MVC (Model-View-Controller) arkkitehtuuri

Yleistermi nykyaikaisten sovellusten modulaarisille ohjelmistoarkkitehtuurimalleille.

Objekti-literaali

Json -objekti, jonka avaimen arvona voi olla myös funktio.

SPA (Single Page Application)

Yksisivuinen web –sovellus, joka pitää sisällään (kaiken) sovelluksen tarvitseman tiedon.

SVG Skaalautuva vektorigrafiikka (Scalable Vector Graphics)

View Käyttäjälle näkyvä osa sovellusta.

Valinta (selection)

D3:lla luotu joukko joka sisältää DOM –elementit ja elementtejä vastaavat data-alkiot.

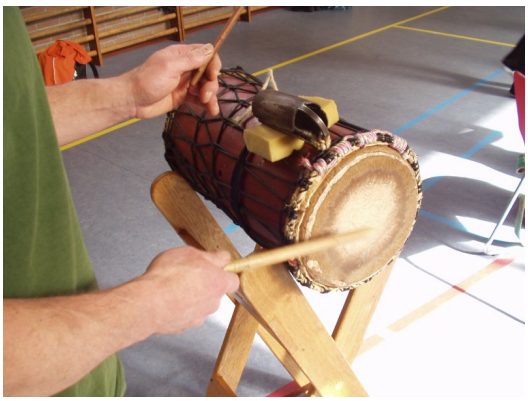
ViewModel Mallin ja näkymän synkronia.

2 Djembe ja dun-dun rumpujen äänien nuotinnus

Rytmimusiikin nuotinnus on yksinkertaisempaa kuin sävelkorkeuden vaihteluun perustuvien soitinten nuotinnus, koska yhdellä perkussio –instrumentilla voi tyypillisesti soittaa vain tietyn rajallisen määrän erilaisia ääniä. Opinnäytetyössä toteutetaan Länsi – Afrikassa suosittujen Djembe ja dun dun –rumpujen nuotinnuksen mahdollistava ohjelma. Molemmilla rummuilla voi tuottaa tietyn joukon perusääniä.

2.1 Dun dunit

Dun dun –rumpuja on kolme, paitsi rakenteeltaan, myös niillä tuotettavan äänivalikoiman kannalta samanlaisia, eroten ainoastaan sävelkorkeuden osalta. Dun duneilla luodaan musiikin rytminen ja melodinen taustapohja jota kiinnekohtana käyttäen kokoonpanon muut soittimet soittavat (Wikipedia dununs 2016). Dun dunit ovat sylinterinmallisia rumpuja joiden päihin on viritetty nahkainen kalvo (Wikipedia dununs 2016). Dun dun -rumpuja saatetaan kutsua Länsi-Afrikan eri kulttuureissa eri tavoin; tässä opinnäytetyössä käytetään Guineassa yleensä käytettyjä nimityksiä, jotka ovat alkaen tuotetun äänen sävelkorkeuden mukaan korkeimmasta matalimpaan ja koon mukaan pienimmästä suurimpaan: *kenkeni*, *sangban* ja *dounoumba*. Kalvon läpimitat rummuilla on yleensä ~30, ~40 ja ~50cm (Wikipedia dununs 2016.). Kullakin näistä on kaksi äänilähdettä joita voidaan käyttää samanaikaisesti – yksi äänilähde kummallekin kädelle. Toinen äänilähde on rumpuun kiinnitettävä kello ja toinen rummun kalvo. Kellosta syntyy vain yksi ääni, kalvosta on mahdollista tuottaa kaksi ääntä: suljettu tai basso. Basso ääni toteutetaan antamalla kepin kärjen kimmota kalvosta, kun taas suljetussa lyönnissä keppi ikään kuin liimataan hetkeksi rummun kalvolle. Traditionaalisessa kokoonpanossa kullakin rummulla on oma soittajansa.



Kuva 1 Pienin dunduneista, kenkeni. Kaikilla dunduneilla on kaksi äänilähdettä: kalvo ja kello. (Wikipedia, dununs 2016.)

Rummut on mahdollista asettaa myös pystyasentoon, jolloin yksi soittaja voi soittaa niitä molemmissa käsissä olevilla kapuloilla. Kelloja ei tällöin käytetä.

2.2 Djembe

Djembe on käsin soitettava, yleensä vuohennahalla päällystetty puinen pikarinmallinen rumpu (Wikipedia, djembe 2016). Djembellä on vain kolme peruslyöntiä: avoin, suljettu ja basso. Bassoääni tuotetaan lyömällä kämmenellä keskelle rummun kalvoa, avoin ja suljettu ääni saadaan erilaisilla lyöntitekniikoilla kalvon reunaan, siten että sormet ja osa kämmenestä osuvat kalvolle.

Kuva 2 Djembe.
(Wikipedia, djembe 2016.)

2.3 Rytmin nuotinnus

Rytmiksi voidaan käsittää se tapa miten sävelteos jaottuu ajallisesti korvalle hahmotettavalla tavalla (Wikipedia, rytmi). Vaikka hahmottaminen voi tuottaa korvalle vaikeuksia, on kaikki teoksen lyönnit ajallisesti tietyssä paikassa. Tyypillisessä traditionaalisesta Guinealaisesta djembe & dun dun -musiikkista on yleensä löydettävissä luonteeltaan hyvin toisteisia piirteitä. Jos kokoonpanossa on djembejä enemmän kuin yksi, soittaa yksi djembeistä yleensä toistuvasti samaa kuviota. Toistuvaa roolia soittavaa kuviota kutsutaan usein kompiksi. Samoin joku dun duneista – yleensä kenkeni – soittaa koko rytmin ajan samaa kuviota. Kuvio koostuu tietystä määrästä mahdollisia kohtia aikajaksolla, joihin lyönti voi osua. Metrisessä kuviossa iskujen kohdat ovat tasavälisiä keskenään, jolloin niiden esitys horisontaalisella akselilla lineaarisesti vastaa itse musiikkia.

Mitään yleisesti käytettyä standardoitua tapaa djembe ja dun dun –rumpujen nuotinnokseen ei ole olemassa. Patrick Hernlyn (Hernly 2010) tekemä tutkimus 53 musiikin opiskelijalle antoi viitteitä siitä millainen nuotinnus –järjestelmä djembe & dun dun -rumpujen nuotinnukseen opiskelijalle tuntuu luontevimmalta ja musiikin hahmottamisen kannalta parhaalta. Opiskelijat, jotka olivat jo perehtyneitä länsimaiseen tapaan

nuotintaa musiikkia, jaettiin kolmeen eri ryhmään joissa opiskelijoiden kokemusta sekä rumpujen soitosta että vaihtoehtoisista tavoista nuotintaa rytmejä kasvatettiin asteittain. Vaihtoehtoisiin nuotinnustapoihin perehdyttäessä ja djembe & dun dun –rumpujen soiton opetuksen kasvaessa opiskelijat enimmäkseen kallistuivat TUBS (Time Unit Box System) nuotinnustavan kannalle. (Hernly 2010, 14-15). Vastaavantyyppinen systeemi on tarkoitus toteuttaa myös opinnäytetyössä, jonkin verran mukautettuna. Tarkastellaan rytmin soli nuotinnusta kyseisellä (TUBS) tavalla.

Rumpujen lyöntien symbolit:

Djembe		Dun dun		Dun dun -kello	
basso:	B	Basso	B	kello	x
avoin	S	suljettu:	⊙		
suljettu	T				

pulssi	1	2	3	4	5	6	7	8	9	10	11	12
tahtikohta	1	2	3	1	2	3	1	2	3	1	2	3
kutsu	TT		T	T		T	T		T	T		
Djembe1	S		T	S			S		T	S		
Djembe2	S			S	T	T	S			S	T	T
1) kenkeni	x		x		x		x		x		x	
	B		B				B		B			
Sangban	x		x		x		x		x	x		x
	B				⊙		⊙			B		
dounoumba	x		x	x		x	x		x	x		x
	B			B					B	B		B

Kaavio 1. Rytmin soli nuotinnus tahtiviivoineen. Rytmin kuvion pituus on kaksitoista.

Nuotinnoksessa on merkitty vertikaalisilla lihavoiduilla viivoilla tahdit. Rytmi koostuu neljästä tahdista, jotka kaikki koostuvat kolmesta atomisesta kohdasta johon lyönti voi

osua. Rytmiä kutsutaan tällöin kolmijakoiseksi. Jos tahdissa olisi neljä mahdollista kohtaa, kutsuttaisiin rytmiä nelijakoiseksi. Yhtä aikajaksolla mahdollista kohtaa rytmissä voidaan kutsua pulssiksi (Hernly 2010, 12) - jokaisella instrumentilla on joko kyseisessä kohdassa jokin instrumentille ominaisen iskuvalikoiman isku, tai sitten iskua ei ole lainkaan. (Hernly 2010, 12) Vertikaalisen visuaalisen luonteensa ansiosta TUBS -nuotinnus valaisee instrumenttien iskujen ajoitusta toisiinsa nähden länsimaista perinteistä nuotinnusta paremmin, mikä auttaa rytmin analysoimisessa (Hernly 2010, 15). Yksittäisen iskun keston kuvaamista TUBS -nuotinnuksessa ei tehdä (Hernly 2010, 12). Samoin tempoa, nopeutta jolla musiikkia soitetaan eli ajanjaksoa joka kuvion toistamiseen kuluu, ei kuvata. Esimerkkirytmin nuotinnus koostuu kahdestatoista alkeiskohdasta, pulssista. Jokaisella instrumentilla on tietty roolinsa tai kuvionsa jota ne enimmäkseen toistavat koko rytmin soiton ajan, välillä mahdollisesti varioiden. Eri instrumenteilla kuvio voi olla samassa rytmissä erimittainen, tällöin tyypillisesti pidempi kuvio on lyhyemmän monikerta. Instrumentin toistamaa kuviota kutsutaan tässä opinnäytetyössä rooliksi. Rytmin kuulijalle hahmottuma toisteisuuden ja jatkuvuuden tunne rakentuu monella tasolla rytmin rakenteen mukaan, mutta pisimmällä kaarella se yleensä muodostuu sen instrumentin mukaan jolla on pisin rooli. Tätä kutsutaan tässä opinnäytetyössä kuvioksi, tavallaan nuotinnuksen ja rytmin pääelementiksi, jota vasten nuotinnuksen instrumenttien nuotit hahmotetaan. Soli –rytmissä kuvion pituus on kaksitoista, sama kuin sarakkeiden määrä.

2.4 Vaatimusmääritys

Ohjelmalla on tarkoitus kyetä näyttämään, muokkaamaan, poistamaan jo olemassa olevia rytmejä sekä luomaan uusia rytmejä. Uutta rytmiä luodessaan ohjelman käyttäjän on kyettävä valitsemaan rytmin kuvion sekä tahdin pituus. Koska rytmi soli on kolmijakoinen, tahdin pituudeksi tulee kolme ja kuvion pituudeksi 12. Kuvion pituuden on oltava jaollinen tahdin pituudella. Uutta rytmiä luotaessa valitaan pohjaksi kuvion ja tahdin pituudet. Tämän jälkeen valitaan kokoonpanon instrumentit käyttöliittymästä, yksi kerrallaan, jolloin kyseiselle instrumentille ilmaantuu nuotinnospohjaan oma rivi (esimerkissä 1). Dun dun –rummuille on oltava mahdollista antaa sekä kellon että basson nuotit.

Rytmi aloitetaan solo –dejembeistin soittamalla kutsulla. Kutsu on yleensä kuvion mittainen, ja kutsun päättyessä musiikki alkaa alkeiskohdasta yksi, ykköseltä, taulukon vasemmanpuoleisimmasta solusta.

Uuden rytmin nuotinnuksen tekeminen aloitetaan valitsemalla käyttöliittymän nuottipaletista tietyn instrumentin tietty lyönti. Nuottipaletissa on kaikkien sovelluksen tukemien instrumenttien kaikki lyönnit. Kun lyönti on valittu paletista, voidaan kyseisen instrumentin äänilähteen rivillä valittuun soluun klikkaamalla asettaa valittu lyönti. Jos instrumentin valikoimaan ei kuulu kyseistä lyöntiä, ei uuden lyönnin asettaminen saa onnistua.

3 AngularJS

Verkko- ja selainteknologian kehitys on mahdollistanut toiminnaltaan yhä monipuolisimpien verkkosovellusten kehittämisen, sellaisten, joita aikaisemmin on totuttu näkemään vain työpöytäkoneilla, natiivisovelluksissa. Verkkosovelluksissa sovelluksen taustalla oleva data ja sovelluslogiikka on sijainnut web-palvelimella, mikä on perinteisesti aiheuttanut haasteita sovelluksen toteuttamisessa. Pelkistetysti ilmaistuna alkuvaiheissaan verkkosovellukset toimivat seuraavasti: Käyttäjä syötti selaimeen osoiteriville URL- osoitteen, osoitteesta löytyvä html-sivu latautui asiakkaan selaimeen, ja selain tulkitsi html -sivun käyttäjän ymmärtämään muotoon. Halutessaan uutta sisältöä käyttäjä joko kirjoitti osoiteriville uuden osoitteen tai napsautti hyperlinkkiä joka etsi osoitetta vastaavaan uuden sivun jonka selain käyttäjälle näytti. Jokaisen merkittävän toiminnon tapahtuessa lähti pyyntö asiakkaan koneesta palvelimelle: dataa liikkui molempiin suuntiin lähes joka kerta käyttäjän suorittaessa jonkin toiminnon joka edellytti uutta dataa. Vain joitain suhteellisen yksinkertaisia toimintoja – kuten lomakkeiden tekstikenttien syötteiden tarkistukset – kyettiin toteuttamaan selaimiin oletuksena sisältyvällä JavaScript –ohjelmointikielellä.

3.1 SPA (Single Page Application)

Ratkaisevaa roolia vastaamisessa haasteisiin joita syntyy datan siirtämisessä verkon ylitse eri sovelluksen osien välillä on esittänyt SPA (Single Page Application) – sovelluskehitystapa. SPA:ssa kaikki tarvittava, tai ainakin oleellinen koodi – HTML, CSS, JavaScript – ladataan tyypillisesti yhdellä kertaa palvelimelta selaimeen. Sovellukselle on vain yksi pääsykohta, yksilöity URL- osoite, joka tarjoaa asiakkaalle kaiken sovelluksen käytön antaman informaation (Sotelo 2012). Käyttäjälle näytetään kulloisellakin hetkellä vain haluttu osa tästä tiedosta jota päivitetään tyypillisesti reaktiona käyttäjään toimenpiteisiin. Tällöin käyttöliittymän näkymä päivitetään esim. uuden datan perusteella (Wikipedia, Single Page Application), mikä tapahtuu nopeasti, koska vaadittava data on jo valmiiksi käyttäjän koneella. Vuorovaikutus muiden selainpuolen tekniikoiden kanssa (HTML, CSS) edesauttaa monipuolisen ja yhtenäisen kokonaisuuden luomisessa.

3.2 AnguarJS ja D3

AngularJS on on Googlen kehittämä JavaScript –sovelluskehitysalusta (application framework) jonka oleellisena päämääränä on ollut valjastaa SPA –kehityksen mukanaan tuomat mahdollisuudet täysimääräisesti. Angularin dokumentaation mukaan ”HTML:sta olisi tullut angularin kaltainen mikäli se olisi suunniteltu sovelluskehitystä varten.” (Angular doc). Angular on suunniteltu nopeasti käyttäjän toimintoihin reagoivien dynaamisten sovellusten kehittämiseen. Viimeisin vakaa versio Angularista on 1.5 (Angular doc).

3.2.1 Sovelluskehysalustat (framework) ja JavaScript kirjastot (library)

Sovelluskehitysalusta tulee erottaa käsitteenä JavaScript –kirjastosta. Sovelluskehitysalusta määrittää etukäteen rakennetta jota ohjelmiston on toteutuksessa käytettävä. Tietyt komponentit (esim. Näkymä, malli, asynkroninen kutsu) toteutetaan aina pitkälti samalla tavalla ja myös niiden kommunikointi on tarkoin säädelty. Tällöin toimiviksi todetut suunnittelumallit tulevat automaattisesti käyttöön, toiminnot toteutetaan standardoiduilla tavoilla ja koodista tulee kehitysalustaan perehtyneille helpommin ymmärrettäviä. AngularJS on monipuolinen ja yksinään sitä käyttäen on mahdollista toteuttaa laajoja sovelluksia. Miinuspuolena on jonkinasteinen joustavuuden väheneminen. (Waikar 2015, loc 283) Muita suosittuja JavaScript –kielellä toteutettuja sovelluskehitysalusalustoje ovat mm. React (Facebook), Backbone ja knockout.

3.2.2 Kaksisuuntainen sidonta

Angularissa erotetaan näkymä (DOM, HTML-tags) deklaratiivisesti kirjoitettavaksi, ja varsinainen sovelluslogiikka (JavaScript) toteutettavaksi tyypillisellä ohjelmoinniksi käsitettävällä imperatiivisella ohjelmoinilla (Wikipedia, AngularJS). Datan ja näkymän kaksisuuntainen sitominen (two-way binding) on ehkä Angularin eniten ohjelmoijan työtä helpottava piirre. Kun näkymän takana oleva malli (data) muuttuu, heijastuu se välittömästi näkymässä vastaavaa dataa käyttävään elementtiin. Samaten jos näkymän ele-

mentti on sellainen johon käyttäjä voi antaa syötteen, päivittyy syötteen arvo välittömästi malliin.

Angular -sovellus sisällytetään html –sivulle attribuutilla ng-app. Sovellus on käytettävissä sen elementin alueella johon attribuutti on upotettu, useimmiten tämä on <html>. Attribuutin nimen tulee erota html-kielessä käytettävien attribuuttien nimistä, jotta selain kykenee tulkitsemaan sen nimenomaan angulariin kuuluvaksi. Angular sisältää ison joukon valmiiksi määriteltyjä attribuutteja. Näitä attribuutteja Angularissa kutsutaan direktiiveiksi: Angular osaa tulkita näitä direktiivejä merkityksellisellä tavalla. Esimerkki:

```
<div ng-app>      <!--  angular toimii tämän elementin sisällä  -->
Nimi: </label >
<input type =" text" ng-model ="nimi">  <!—luo implisiittisesti muuttujan 'nimi' -- >
<h4> Huomenta {{nimi}}! </h4>  <!—päivittyy automaattisesti muuttujan muuttuessa -- >
</div>
```

Yllä olevassa koodipätkässä tekstikentän direktiivi ng-model luo malliin muuttujan nimi. JavaScript luo sekä mallin että muuttujan siihen implisiittisesti, vaikka sen paremmin mallia kuin nimenomaista muuttujan esittelyä ei koodissa oltaisi tehtykään. Tuplaviiksien {{ }} sisältö tulkitaan angularin mallia vasten: prosessia kutsutaan interpolaatioksi (interpolation). Koska mallista löytyy muuttuja 'nimi' päivittyy kyseiseen kohtaan (<h4>) välittömästi se arvo, mikä mallin muuttujalla kulloisellakin hetkellä on. Kun käyttäjä syöttää tekstikenttään kirjaimia, muuttuu arvo joka kerta.

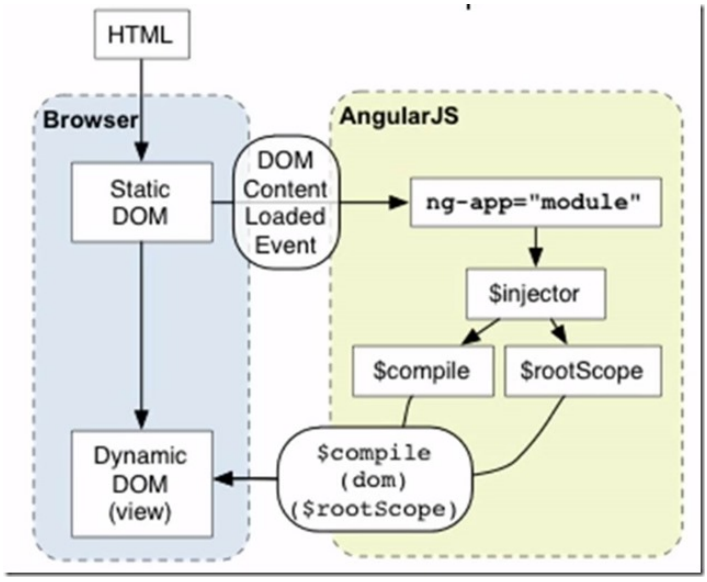
3.2.3 Angular ja selain

Mitchel Marxin mukaan selain tekee elääkseen kahta asiaa: Parsii HTML ja css -koodeista DOM-puun, sekä luo tälle tapahtumankäsittelijät. Angular laajentaa selaimen kykyä molemmista näkökulmista. (Marx 2015)

Saadessaan palvelimelta html –koodia sisältävän sivun, parsii selain sen läpi ja muodostaa siitä ja mahdollisesta css -tyylitiedostosta puurakennemallia vastaavan DOM –

objektin. Modernit selaimet jättävät spesifikaatioon kuulumattomat tunnistamattomat elementit HTML –koodissa huomiotta, kuten angularin direktiivit, joten tässä vaiheessa DOM on rakentunut aivan kuin angularia ei olisi olemassakaan. (Marx 2015.)

Angular rekisteröi itsensä kuuntelemaan selaimen DOMContentLoaded –tapahtumaa, minkä selain laukaisee selaimen rakentaman DOM –puun ollessa valmis. DOM –puusta tulee syöte angularin $compile –prosessille (Dag-Igne 2014), joka etsii DOM –puusta sen sisältämät angular –direktiivit alkaen siitä elementistä jossa direktiivi ng-app sijaitsee, rakentaa direktiiveistä logiikkansa mukaan mahdolliset omat html –elementtinsä ja tapahtumankäsittelijänsä sekä sulauttaa lisäyksensä ja muutoksensa selaimen parsimaan DOM –puuhun. Samassa prosessissa rakentuu myös kaksisuuntaisen sidonnan mahdollistava JavaScript ohjelmakoodi (Dag-Inge 2014). Lopputulemaa voi kutsua dynaamiseksi DOM:iksi (Marx 2015). Clayton Rabenda (Rabenda 2014) ilmaisee angularin antavan DOM:ille lisää persoonallisuutta. Prosessia voi esittää kaaviolla:



Kaavio 2. Angular - dynaamisen DOM -puun rakentaminen. Developer guide, bootstrap 2016.

Jokaisella angular –sovelluksella on yksi $rootScope, angularin globaali malli, johon kaikki muut mallit lisätään (Developer guide, $rootScope 2016). $compile -prosessi suorittaa kääntämisprosessinsa $rootScopea vasten ja lisää siihen ohjelman suorituksen aikana vaadittavat angular –komponentit (Marx 2015). Joillain angular –

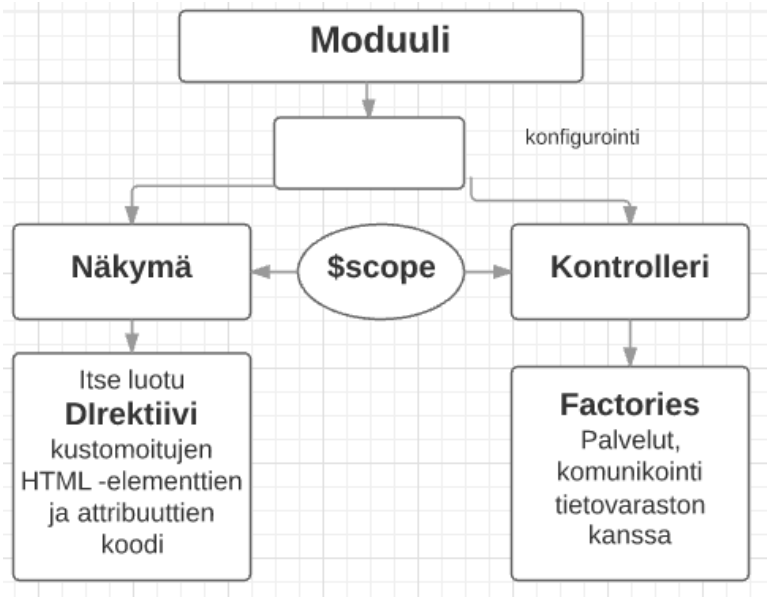
peruskomponenteilla on oma mallinsa (Controller, usein Directive), jota $angular –maailmassa kutsutaan nimellä $scope. Kaikki $scopet lisätään puurakennemaisesti $rootScopeen(ModusCreate).

Koodia kirjoitettaessa staattisen ja dynaamisen DOM:in erosta on hyvä olla tietoinen. Suositeltavaa onkin säännöllisesti tarkastella Angular –parsimisen jälkeen syntynyttä dynaamista DOM –puuta selaimen sovelluskehitystyökaluilla.

3.2.4 Moduuli (Module)

Angularin arkkitehtuuritason organisoinnissa suurin yksikkö moduuli. Moduuliin voidaan lisätä alemman tason komponentteja kuten kontrollereita, direktiivejä, palveluita ja suodattimia. Moduuli luodaan komennolla angular.**module**('moduleName, [riippuvuudet]). Viitatessa moduuliin myöhemmin annetaan funktiolle vain yksi parametri: angular.module(”moduleName”). Koska moduuli vastaa korkeamman arkkitehtitason komponenttia on pienessä sovelluksessa usein vain yksi moduuli, mutta sovelluksen laajetessa koodin jakaminen ylläpidettävämpiin ja loogisesti helpommin hahmotettaviin kokonaisuuksiin on järkevää. Moduulien avulla voidaan myös välttää koodissa käytettyjen nimien päällekkäisyydet pitämällä globaali nimiavaruus puhtaana. Angular ei itse huolehdi moduulien lataamisesta, vaan nämä on luotava koodissa edellä mainitulla tavalla ja tiedostoihin joissa ne sijaitsevat täytyy luoda linkki index.html -tiedostossa. (Karpov 2015, 109.)

Moduuliin voidaan lisätä palveluja tarjoavia komponentteja riippuvuusinjektion (dependency injection, DI) avulla. Ulkopuoliset Palvelut, joita moduuli haluaa käyttää annetaan taulukkomuodossa funktion toiselle parametrille ja ne voivat olla toisia moduuleita, tai muita angular-komponentteja. Angularin pääkomponentit voidaan esittää kaaviolla. Kaavio on mukaelma Dan Wahlinin (Wahlin, Big Picture, 2013) esittämästä kaaviosta:



Kaavio 3. Angularin modulaarinen rakenne. Sovelluksessa jossa on vain yksi näkymä ei tarvita konfigurointia, eikä sitä käsitellä tässä opinnnäytetyössä.

3.2.5 MVC angularissa

Toisin kun aikaisemmassa esimerkissä, sijoitetaan malli ymmärrettävyyden lisäämiseksi yleensä omaan JavaScript –tiedostoonsa. Angularissa mallia vastaa koodin tasolla $scope -objekti. $scopen elämä Angular –maailmassa tapahtuu Controllerin sisällä. Käytettäessä kontekstina ohjelmoinnissa klassikoksi kohonnutta Model – View – Controller (MVC) –suunnittelumallia, vastaavuudet Angularissa menevät kutakuinkin:

(MVC) Angular
Malli (Model) $scope
Näkymä (View) HTML (DOM)
Ohjain (Controller) Controller

MVC –suunnittelumallissa malli vastaa ohjelmassa käsiteltävästä datasta ja sen pohjalle rakennetusta sovellukselle ominaisesta toimintalogiikasta. Näkymä vastaa loppukäyttäjälle näytettävästä käyttöliittymästä, ja MVC –mallin puhtaassa tulkinnassa näkymän olisi oltava tietämätön ja eristetty mallin tilasta. Ohjaimen rooli on toimia välittäjänä mallin ja näkymän välillä: yhtäältä reagoitava käyttäjän toimintoihin näkymässä (käyttöliittymä) laukaisemalla toiminto mallissa, ja toisaalta kyettävä heijastamaan mallissa ta-

pahtunut muutos näkymään. Angularin kaksisuuntainen sidonta luo kuitenkin eräänlaisen välittömän vaikutuksen efektin mallin ja näkymän välille, ja ohjaimen rooli välittäjänä näiden kahden välillä liukenee koodaajan kannalta näkymättömiin. Ohjaimen tehtäväksi muodostuu toimintalogiikan toteutus ja kommunikointi ohjelman muiden osien kanssa (Karpov 2015, 134-135). Marx kuvaa ohjainta(Controller) Angular –sovelluksen aivoiksi. Jatkossa käytetään ohjaimen sijasta sanaa kontrolleri, koska sellaiseksi se koodaajaan päässä väkisinkin muodostuu, sillä Angular –koodissa ohjaimen roolin toteuttaa 'Controller' –niminen objekti. Tutkitaan esimerkkikoodia, jossa Näkymä ja malli on erotettu omiin tiedostoihinsa (vain osa koodista).

Index.html (näkymä)

```
<script type="text/javascript"
src="//ajax.googleapis.com/ajax
/libs/angularjs/1.4.1/angular.min.
js"></script>
    <script type="text/javascript"
src="app.js"></script>
<body ng-
app=”nyrkkivipapeApp”>
<div ng-controller=”nkpCtrl”>
  <label> Name: </label>
  <input type ="text" ng-model
  ="user.nimi">
  <div ng-hide=”user”><label>
  Aseesi: </label>
  <h3> Huomenta {{nimi}}!
  Aseesi: {{ase}}</h3>
  <button ng-click
  ="vaihda">Vaihda
```

App.js (kontrolleri & malli)

```
var app = angular.module(' nyrkkivi-
papeApp, []);
app.controller(' nkpCtrl', [' $scope',
function ($scope) {
  $scope.user = {
     nimi: 'Migu',
     email: 'migu@gmail.com',
   };
  Var i =0;
  $scope.aseet=[
    ”nyrkki”,
    ”kivi”,
    ”paperi”
  ]
  $scope.ase=$scope.aseet[i];
  $scope.vaihda(){
    if(i==2){i=0}
    else{i++}
```

Kuva 3. Näkymä vasemmalla, malli ($scope) ja kontrolleri oikealla

Kaikki index.html –sivun käyttämät tiedostot linkitetään yhteen <scrcipt type> -elementeillä. Tiedostossa index.html näkyy miten angular-lähdekoodi otetaan käyttöön <script type src=”Angular-osoite”> -elementillä ja sovelluksen mallin ja kontrollerin sisältävä 'app.js' –tiedosto linkitetään elementillä <script type="text/javascript" src="app.js"></script>.

Esimerkissä Angular –sovellus on asetettu <body> -osioon atrribuutilla ng-app – vastaava sovellus luodaan app.js tiedoston ensimmäisellä rivillä. Angular-maailma on tällöin näkymälle käytössä koko <body> - elementin sisällä. Mikäli angularia haluttaisiin käyttää koko sivulla, voitaisiin direktiivi asettaa <html> -tagin sisälle

Sovelluksen toiminnosta vastaava kontrolleri on luotu <body> -elementitä seuraavalla elementillä , tällöin kontrollerin tarjoamia toiminnallisuuksia voidaan käyttää kyseisen elementin sisällä. Kontrolleri saa injektoinnin avulla $scope objektin joka – kuten kuvassa 5 näkyy – toimii eräänlaisena liimana kontrollerin ja näkymän välillä (Developer guide, scopes). Sekä kontrollerilla että näkymässä käytetyillä direktiiveillä on viittaus $scopeen, muttei toisiinsa. (Developer guide, scopes) Tämä linkitys mahdollistaa sen että kaikki $scopen muuttujissa (user, ase, aseet) ja toiminnoissa (vaihda) tapahtuvat muutokset heijastuvat automaattisesti näkymään.

3.2.6 Angularin sisäänrakennettujen direktiivien ryhmittely

Direktiivit voidaan jakaa kolmeen ryhmään (Karpov 2015, 134):

Taulukko 1. Angularin direktiivien ryhmittely kaksisuuntaisen sidonnan kannalta.

Direktiiviluokka	Näyttää dataa näkymässä?	Muokkaako dataa?	Esimerkki -direktiivejä
1) Muokkaa vain näkymää	Kyllä	Ei	ngBind, ngBindHtml, ngRepeat, ngShow, ngHide
2) Tapahtumankäsittelijä	Ei	Ehkä (kykenee siihen)	ngClick, ngMouseOver, ngDblclick
3) Kaksisuuntainen sidonta	Kyllä	Kyllä	ngModel (tekstikenttä, pudotusvalikko)

Kuvan kolme koodiesimerkissä löytyy esimerkki jokaisesta direktiiviluokasta. NgHide (ryhmä 1) näyttää elementin sisällä olevat elementit, mikäli sen arvo (tässä:=”user”) on

tosi. Ehtolausekkeissa JavaScript tulkitsee objektin todeksi mikäli sen arvo ei ole 'null' tai 'undefined'. Koska esimerkin app.js –tiedostossa on muuttujalle $scope.user asetettu objekti arvoineen, renderöidään näkymässä -elementti lapsielementteineen. Mikäli koodissa asetettaisiin $scope.user={}, tulkitsisi javaScript tyhjän objektin epätodeksi (JavaScript Doc), ja kaksisuuntaisen sidonnan välittömän vaikutuksen ansiosta näkymään -elementin alla olevaa tietoa ei näytettäisi.

Ryhmään 2 kuuluu tapahtumankäsittelijä ng-click. Näkymässä painikkeen klikkaus kutsuu kontrollerin funktiota vaihda(). Funktio lisää muuttujan i arvoa, mikä edelleen päivittää muuttujan $scope.ase arvoa. Koska ase on kytketty $scope –objektiin, heijastuu muutos välittömästi näkymään. Jotta malliin tehty muutos ilmenisi näkymässä, on muuttuja asetettava kaksinkertaisten aaltosulkeiden sisään {{ase}}, mikä on eräs tapaa käyttää angularin lausekkeita. Tällöin angular kulissien takana kykenee maagisesti reagoimaan muuttujan arvon muutokseen.

Ryhmään 3 kuuluva direktiivi (ngModel) toimii täsmälleen kuten aikaisemmin koodiesimerkissä (kpl. 3.2.2). Jos tekstikenttään syöttää arvon, heijastuu muutos kontrollerin sisällä olevaan $scope.user –objektin 'nimi' –kenttään ja toisin päin. Angularille tämän tekee mahdolliseksi se seikka että tekstikentän direktiiville 'ng-model' on annettu arvoksi mallista löytyvä 'user.nimi'. Muita vastaavalla kaksisuuntaisella tavalla ng-model direktiivejä hydyntäviä angular elementtejä ovat tekstialue (textarea) ja pudotusvalikko (select).

3.2.7 Lausekkeet

Lausekkeet (expressions) on angularin tapa sitoa data html –koodiin (Developer guide, expressions). Näkymissä niitä käytetään ainoastaan kahdella mahdollisella tavalla: kaksinkertaisten aaltosulkeiden sisällä tai direktiivin attribuutissa (Developer guide, expressions). Angular tulkitsee näkymässä olevien lausekkeen sisällön aina vastaavaa kontrollerin $scope –objektia vasten, prosessia kutsutaan interpolaatioksi (interpolation)(Developer guide, expressions). Jos kontrollerissa on esim. Koodipätkä:

```
$scope.myURL ="www.kotisivu.com/tuotteet";
```

$scope.kotisivu = ”Tutustu valikoimaamme!”;

$scope.a = 3;

$scope.b=4;

$scope.functionExpression(){ toiminta}

Tällöin tyypillisiä tapoja käyttää expressioita ovat esim:

Taulukko 2. Angularin lausekkeita.

Html -lähdekoodi	*Lopputulos selaimessa*
<a href="{{myURL}}">{{kotisivu}}	*Tutustu valikoimaamme*
*{{a*b}}*	*12*
* 9 * 7 = {{9 * 7}} *	*9 * 7 = 63*
ng-click="functionExpression()"	*Elementin painalluksesta kutsutaan $scopen funktiota functionExpression()*

Lausekkeet ovat osallisena angularin datasidonnassa näkymän ja mallin välillä. Kaksisuuntaisessa sidonnassa ne voivat olla joko datan muutoksen lähde sekä kohde (textbox, textarea, select) tai yksisuuntaisessa sidonnassa, jossa dataa muutetaan $scopessa, pelkkä kohde.

3.2.8 $scope

Kontrollerin paljastama $scope voidaan nähdä olevan näkymän suunnittelijalle eräänlainen julkinen rajapinta, API (Application Programmin Interface), jota hän käyttää hyväkseen (Ruebbelke, 2015). Sovelluksen suunnittelusta tulee deklaratiivista, jossa näkymässä kerrotaan mitä toimintoja halutaan ja minkä tyyppiseksi rakenteeltaan näkymän halutan muodostuvan, selaimen rakentaessa dynaamisen DOM –puun yksityiskoh-

taisemmin kontrollerin sääntöjen määräämänä. $scopen funktioiden sisäistä toteutusta voidaan muokata näkymän siitä tietämättä. (Karpov, 135.)

Jokaisessa angular –sovelluksessa on vähintään yksi $scope nimeltään $rootScope, joka luodaan automaattisesti aina kun DOM elementissä on ng-app direktiivi (Suomijoki 2015, 14). $rootScope linkitetään selaimen globaaliin $window JavaScript objektiin (ModusCreate). Muiden angular –elementtien, kuten kontrollerin, luomat $scopet lisätään tähän hierarkisesti Angularin ensimmäisen kääntämisen ($compile) aikana.

3.2.9 Kaksisuuntainen sidonta pinnan alla

Sekä näkymä että kontrolleri molemmat näkevät $scopen – joka on tavallinen JavaScript –objekti(Developer Guide, $scopes) - mutteivät toisiaan (Suomijoki 2015, 14). Jokaiseen sellaiseen $scopeen liitettyyn muuttujaan ja funktioon jota käytetään näkymässä Angular liittää dynaamista DOM –puuta rakentaessaan $watch –funktion (Developer guide, $scope), jonka tarkoituksena on objektin arvon muutoksen seuraaminen. $watch –funktio saa kaksi parametria. Ensimmäinen parametri on jokin $scopeen liitetty muuttuja ,funktio tai näistä muodostettu lauseke. Suunnitteluperiaatteena on että kokonaisuutena $scopeen liitetyt $watch –funktiot mahdollistuvat mallin tilan muutosten heijastumisen näkymään. Angular kutsuu $watch -funktiota säännöllisesti tarkoituksena seurata ensimmäisen parametrin arvoa ja verrata sitä edellisellä seurannalla laskettuun arvoon. Mikäli arvo on muuttunut, kutsutaan $watch –funktion toista parametriä, takaisinkutsufunktiota (watcher, angularin dokumentissa listener) (Developer guide, $scope.). Takaisinkutsufunktio pitää kirjaa missä kaikkialla näkymissä on tarkasteltua ensimmäistä parametriä käytetty ja osaa päivittää näkymän tilan muuttuneen arvon perusteella. Suunnittelumallina takaisinkutsufunktio käyttää Observer –patternia (Marx 2015). Observer -suunnittelumallissa lähde (subject, $scope-muuttuja) ylläpitää listaa itsestään riippuvaisista havainnoitsijoista (observers, näkymän lausekkeet) joita se automaattisesti informoi tilansa muutoksista (Wikipedia, Observer pattern). Lista muodostetaan dynaamisen DOM –puun rakentamisen yhteydessä (Marx 2015).

3.2.10 Riippuvuusinjektio (dependency injection, DI) ja palvelut

Ohjelmakoodi käyttää lähes aina apunaan jotain toista koodattua objektia. Suunnittelupäämääränä angularin riippuvuusinjektiossa on ettei tuota käytettävää objektia luoda siellä missä sitä käytetään (Karpov 2015, 218). Sen sijaan se luodaan toisaalla ja annetaan sitä käyttävälle angular –komponentille riippuvuutena. Riippuvuutta käyttävän koodin tarvitsee tietää vain sen tarjoama rajapinta ja miten sitä käytetään. Näin erotetaan riippuvuden käyttö sen luonnista ja ylläpidettävyys lisääntyy; riippuvuutta käyttävää asiakaskoodia ei tarvitse muokata lainkaan vaikka riippuvuutta muutetaan (Developer quide, Dependency injection). Angularissa on sisäänrakennettuja palveluja, joita voidaan ottaa käyttöön riippuvuusinjektiolla($http, $q, ym.), mutta injektoitavia palveluja voidaan luoda myös itse. Angularin sisäisissä injektoitavissa palveluissa on etumerkki $. Esimerkissä kontrollerille on injektoitu käyttöön itse luotu palvelu 'greeter'. Kaikki palvelun tarjoamat julkiset metodit ovat sen jälkeen kontrollerin käytössä.

```
someModule.controller('MyController', ['$scope', 'greeter' , function($scope, greeter) {  …palveluita julkiseen API:iin..}]);
```

Mallit jotka eivät ole yksinomaan jollekin tietylle näkymälle tarkoitettuja on tarkoituksenmukaista sijoittaa palveluihin (services)(Developer guide, conceptual overwiev). Palvelut ovat singleton –tyyppisiä, eli niistä luodaan vain yksi instanssi, joka on sama kaikille ohjelmakoodin osille jotka palvelua käyttävät (Developer guide, services). Angularissa on viittä eri tyyppiä palveluita: service, factory , provider, value ja constant. Jos jotain ominaisuutta tarvitaan globaalisti ohjelman eri osissa, on se luonnollista toteuttaa palveluna, joka sitten riippuvuusinjektiolla annetaan palvelua hyödyntävän komponentin käyttöön.

Useimmiten käytetty palveluista on factory (suomijoki 2015, 18), joka palauttaa itse luodun objektin(Developer guide, services). Tyypillinen tapa on luoda objektille privatteja muuttujia ja metodeja, jotka kapseloidaan objektin julkisiin metodeihin, API:iin, joka riippuvuusinjektiolla annetaan palvelua käyttävälle angular -elementille. Tyypillisimpiä käyttötapoja on luoda rajapinta kommunikointiin tietokannan kanssa. Tällöin rajapinnan voisivat muodostaa metodit: 'create, 'get, 'update', 'delete' (CRUD).

3.2.11 Itse luodut direktiivit (custom directives)

Angularin direktiivit liittyvät yksittäisten DOM –elementtien toimintaan. Direktiivejä on neljää eri tyyppiä : elementit, attribuutit, luokat (classes), joita käytetään css –tyylien yhteydessä, sekä kommentit. Selvästi yleisimmin käytettyjä ovat kaksi ensimmäistä: elementit ja attribuutit. Tässä opinnäytetyössä keskitytään ainoastaan elementti -direktiiveihin. Samoin kuin sisäänrakennettujen, myös itse tehtyjen direktiivien eräs tapa laajentaa selainta on lisätä DOM –puuhun elementtejä. Lisäksi direktiivit voivat luoda elementtejä koskevia omia tapahtumankäsittelijöitään. Säännöt joiden mukaan tämä tapahtuu kirjoitetaan direktiivi –komponentin (directive) luomisesta ja toiminnasta vastaavaan ohjelmakoodiin.

Angular –tiimi (AngularJS 10:15) lähestyy direktiivin tarjoamia mahdollisuuksia HTML –koodin selkeässä kustomoimisessa assosiaation kautta standardi –HTML5: een kuuluvaan range -objektiin:

```
<input type="range" min="1" max="100" value="75">
```

Koodi luo sivulle oman komponentin (slider)

Kuva 4. HTML5 -standardiin kuuluva Range –objekti

Komponenttiin on luotu implisiittisesti oma toimintalogiikkansa. Range -objektissa siihen liitetty arvo(value) muuttuu liu'utetatessa palloa vasemmalle tai oikealle. Arvon voi hakea esim. JavaScript –koodilla osaksi ohjelman toimintaa. Toiminnallisuuden toteutuksesta html -sivun suunnittelijan ei kuitenkaan tarvitse tietää mitään. Samanlainen suunnitteluperiaate on ollut direktiivien kehittämisen taustalla: HTML –koodia pyritään laajentamaan kustomoiduilla helposti ymmärrettävillä elementeillä, esim:

```
<rytmi data="nuotit"></rytmi>   // itse luodun Element –direktiivin 'rytmi' käyttö HTML -sivulla
```

Elementistä kykenee yleisellä tasolla suoraan päättelemään mitä se tekee: näyttää rytmin nuotit, jotka on annettu attribuuttina, kontrollerin $scopeen liitetyssä 'nuotit' –

nimisessä muuttujassa. Toteutuksesta vastaa direktiivi –komponenttiin kirjoitettava koodi.

Direktiivi lisätään angularin korkeimman tason komponenttiin, moduuliin, komennolla:

```
angular.module("myModule").directive("myDirective", function() {
  return myDDO;
})
```

Mikäli direktiivin nimessä on isoja kirjaimia on ne HTML –koodissa muunnettava pieneksi ja laitettava eteen alaviiva ('myDIrective' → <my-directive>). Direktiivi palauttaa erityisen JavaScript –objektin Directive Definition Object, (DDO) tai toiminnoltaan yksipuolisemman link-funktion. Link –funktio sisältyy myös DDO -objektiin, ja sen tarkoitus on eristää angularin dynaamisen DOM –elementin rakentaminen (Deveoper guide, Directives). Dokumentaatio suosittelee käyttämään DDO –objektia (Developer Guide, $compile). DDO koostuu JSON –tyyppisestä kokoelmasta valinnnanvaraisia funktioita jotka määrittelevät direktiiville erityiset ohjeet angularin HTML –kääntäjälle. Tarkastellaan viittä oleellista kenttää:

```
Var myDDO={
  controller: function($scope){…koodi…},
  template: ' //  HTML :ää
  scope:{
    periytyvyys & välitettävä data parent-scopelta
  },
  link: function(scope ,iElem, tAttrs)
  …..//  DOM-manipulointi
  },
  restrict:  // 'E' tai 'A' tai 'EA'   //käytetäänkö direktiiviä näkymässä Elementtinä, Attribuuttina vai molempina
  require: // direktiivit joiden kanssa ollaan vuorovaikutuksessa
```

}

Controller ja template –kentät toimivat keskenään samoin kuin aikaisemmin on näkymän, mallin ja kontrollerin suhteesta kerrottu. Käytettäessä näitä kahta muodostaa direktiivi eräänlaisen mini-angularin, angular -logiikalla toimivan eristetyn osan kokonaisen angular-sovelluksen sisään (Wahlin, 2015). Templatessa olevan tekstin on oltava HTML –koodia ja tämä koodi lisätään dynaamisessa DOM –puussa sen elementin sisään, missä tätä direktiiviä on käytetty. Kontrollerilla ei ole etukäteen määrättyä paremetrilistaa mutta se voi riippuvuusinjektion avulla saada käyttöönsä mm. $scope –objektin, johon liitetyt ominaisuudet ovat näkymän (template) käytössä. Näiltä osin itse tehdyllä direktiivillä voidaan eristää toiminnallisuutta haluttuun alueeseen, mikä lisää ohjelman modulaarisuutta ja uudelleenkäytettävyyttä.

Oletuksena $scope –objekti on sama kuin HTML –elementillä jonka sisällä direktiivi on määritelty. Oletusta voidaan kuitenkin muuttaa scope kentässä kolmella eri tavalla (Karpov 2015, 169):

Taulukko 3. Direktiivin näkyvyysalue.

scope: true	Luo uuden periytetyn $scopen jokaiselle direktiivillä luodulle instanssille
scope:{ }	Luo uuden **eristetyn** $scopen jokaiselle direktiivillä luodulle instanssille. Mahdollisuus välittää parentilta muuttujia $scopeen
scope: false	Oletus. Ei luoda $scopea direktiiville, vaan sama kuin parentilla.

Opinnäytetyössä keskitytään ainoastaan eristetyn (isolated) mallin käyttämiseen. Vaikka direktiivin malli ($scope) luodaan tällöin ikään kuin tyhjänä mallina, voidaan sille välittää muuttujia niistä malleista jotka ovat näkyvissä siinä html-elementissä jossa direktiivi luodaan. Muuttujat voidaan välittää html:ssä direktiivin attribuutilla joko yksi- tai kaksisuuntaisina. Lisäksi direktiivin on mahdollista kutsua ulkopuolista maailmaa itsestään käsin, jos direktiivin attribuutille annetaan se kontrollerin funktio jota direktiivistä halutaan kutsua. Seuraavassa esimerkissä pätkä direktiivin koodia, DDO:n scope –kenttä:

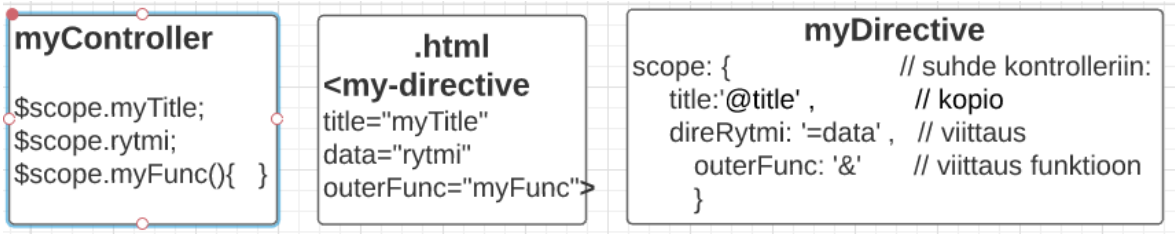
```
scope{
  myVar: ='data'   // kaksisuuntainen muutos direktiivin muuttujassa heijastuu ulos
```

```
    myTitle: '@'      //  yksisuuntainen, muutos heijastuu ulkoa direktiiviin muttei toisin päin
    myFunc: '&'       // direktiivin ulkopuolella oleva funktio jota halutaan direktiivistä kutsua
  }
```

Tällöin, jos HTML:ssä on

```
<div ng-controller="myController">
  <my-directive data="rytmi" my-title="myTitle"  my-func="myFunc"></my-directive>
```

päästäisin direktiivistä käsiksi myControllerin $scopeen linkitettyihin *rytmi* ja *myFunc* – muuttujiin. Direktiivin eristetyn scopen kautta voidaan rakentaa etäänlainen tiedonvälityksen putki, Wahlin kutsuu sitä direktiiviputkeksi (Wahlin, 2015).



Kaavio 4. Direktiiviputki kontrollerin ja eristetyn scopen välillä.

Diretiivin 'myDirective' scope –kentässä on määritelty mitä muuttujia sen muutoin eristettyyn scopeen vodaan ulkopuolelta välittää. Esim kontrollerin myFunc –funktiota voitaisiin nyt direktiivissä kutsua komennolla $scope.outerFunc(). $scope.direRytmi taas on suora viittaus kontrollerin rytmi –muuttujaan ja voi halutessaan muuttaa sitä. Huomioitavaa on että kaksisidonnaisessa vuorovaikutuksessa ('=') ulkopuolissa mallissa on mahdollista käyttää vain muuttujaa, kun taas yksisuuntaisessa on mahdollista käyttää myös lauseketta (Karpov 2015, 175).

Direktiiviä käytetään HTML:ssä lähes aina joko attribuuttina <div myDir> , suoraan elementtinä <myDir></myDir> tai molemmissa. Kentällä 'restrict' määritetään direktiivin käyttötapa ('A', 'E', 'EA'). (Developer guide, directives.)

Jos direktiivin luomaa DOM –elementtiä halutaan muokata tai siihen lisätä tapahtumankäsittelijöitä tapahtuu se link –funktiossa. Link –funktion sijasta voitaisiin käyttää

myös monipuolisempaa funktiota compile, joka myös sisältää link- funktion, mutta opinnäytetyössä ei käsitellä sitä. (Developer guide, Directives.)

3.2.12 Direktiivin kommunikointi

Direktiivin yksityiset kontrollerit tarjoavat direktiivin näkymälle (DDO:n template - kenttä) pääsyn direktiivin yksityiseen malliin kontrollerin $scopen välityksellä (Wahlin). Näin direktiivin sisälle rakentuu eräänlainen mini-Angular –maailma Angular – sovelluksen sisälle (Wahlin), jossa direktiivin sisällä vallitsee Angularin versio MVC - tyyppisestä (malli-näkymä-controller) vuorovaikutuksesta omassa rajatussa kontekstissaan. Kontrollereiden käytön toisena päätarkoituksena on direktiivien välinen kommunikointi (Kurz, 18). Julkinen API (Application Programming Interface), jonka välityksellä kommunikointi tapahtuu, luodaan kontrollerin 'this'-objektiin. (Wahlin). Ohjelmoija voi siis hahmottaa direktiivin kontrollerin koodin organisointia vastaavan jaon kautta: $scopeen yksityiset seikat ja kommunikointi yksityisen html- templaten kanssa ja 'this' objektiin vuorovaikutus muiden direktiivien kanssa.

Direktiiviin API saadaan linkitettyä toiseen direktiiviin DDO:n 'require' –kentällä. Kentälle annetaan direktiivin nimi (tai lista direktiiveistä): *require:'importedDirectiveName'*. Direktiiviin kontrollerin julkiseen API:iin päästään käsiksi `link` –funktiolle annettavassa valinnanvaraisessa neljännessä parametrissa: *link: function(scope, element, attributes, directiveName)* (Developer guide, $compile). Erikoista kyllä, silloinkin kun direktiivin *link* –funktiosta halutaan kommunikoida saman direktiivin kontrollerin kanssa, vaaditaan require kenttä, ja *link* –funktioon neljänneksi parametriksi direktiivin nimi.

3.2.13 Angular näkökulma koodaukseen - revisited

Verrattuna moniin muihin JavaScript –kirjastoihin tai sovellusarkkitehtuureihin Angular kääntää asiat päälaelleen sen suhteen mikä on julkinen API jota käyttäen varsinainen ohjelmointi tehdään. Esimerkiksi JQueryssä asiakkaana on JavaScript –kieli joka käyttää HTML:n ja DOM –spesifikaation tarjoamia palveluja, mutta angularissa julkinen API on paitsi angularin sisäisissä direktiiveissä, myös itse luotujen controllereiden $scope – objekteissa ja kustomoiduissa direktiiveissä. Asiakkaaksi valjastetaan HTML angularin

mausteineen (direktiiviattribuutit ja -elementit), millä toteutetaan halutunlainen käyttöliittymä toiminnallisuuksineen käyttötarkoitukseen suunniteltua API:a hyödyntäen. (Karpov 2015, 158)

3.3 D3.js

D3.js -JavaScript kirjasto on suunniteltu datan visualisointiin verkkosivuilla. Verrattuna moniin muihin visualisointikirjastoihin sillä on mahdollista toteuttaa monipuolisempia, dynaamisia ja vuorovaikutteisia visualisointeja. Tämän toteuttamiseksi D3 valjastaa neljä laajasti tuettua web-standardia: HTML5, CSS, JavaScript ja SVG. Käytännössä kaikki uudemmat web-selaimet tukevat näitä standardeja.

3.3.1 SVG

Digitaalisessa muodossa grafiikkaa toteutetaan kahdella eri tekniikalla: vektoreilla ja bittikartoilla. HTML5 tarjoaa kummankin esittämiseen oman vaihtoehdon: SVG vektoreille ja Canvas bittikartalle. SVG on samalla tiedostomuoto (.svg) jota HTML tukee, ja jolla luotu kuva voidaan upottaa osaksi HTML5 –koodia. SVG tulee sanoista Scalable Vector Graphics, ja nimenmukaisesti SVG:llä toteutettu grafiikka skaalautuu menettämättä kuvantarkkuuden laadussa. Tämän mahdollistaa vektorigrafiikan koostuminen matemaattisista mallista ja funktioista , joiden avulla lasketaan mitkä pikselit annetussa koordinaatistossa määritetään osaksi kuvaa (Jenkov 2013, loc 203). Resoluution muuttuessa matemaattiset mallit sovitetaan muuttuneeseen koordinaatistoon (kaksiulotteisessa grafiikassa X –ja Y –akseleihin) ja kuvan muodot skaalautuneessa koordinaatistossa lasketaan vastaavasti. Kuvan tallennuskoko riippuu yksinomaan matemaattisten mallien mutkikkuudesta, ei pikseleiden määrästä. Tyypillisesti SVG on omiaan esittämään selkeitä, melko yksinkertaisesti matemaattisilla malleilla (monikulmio, ympyrä, ellipsi, käyrät, ym.) esitettäviä kuvia.

SVG –kuvat ja niiden käyttäytyminen määritellään XML –muotoisessa tekstitiedostossa. Lähes kaikki modernit selaimet tukevat SVG –standardia. SVG-elementti (tag <svg></svg>) upotetaan HTML-sivulle – ja se voi edelleen sisältää toisia ennalta määrättyjä SVG –spesifikaatiossa esitettyjä grafiikkaa luovia elementtejä kuten esim.

suorakaiteen toteutuuttava <rect>. SVG –elementeissä voi käyttää CSS-tyylejä samoin kuin muissakin HTML-elementeissä, mikä auttaa sivuston yhtenäisen tyylin luomisessa. Oleellisin tekijä datan käsittelyn kannalta on kuitenkin se että jokaista esitettävää datan arvoa vastaamaan voidaan luoda yksi svg -elementti, mikä mahdollistaa datan yksityiskohtaisen esittämisen. Esimerkki:

```
<svg width="400" height="400">
 <rect x="50" y="20" width="25" height="25" class="nelio"
 style="fill:blue;>
</svg>
```

Yllä on esimerkki SVG –säilöstä, jonka sisälle on asetettu sininen neliö (rect). Neliölle on annettu koordinaatit SVG –elementin vasemmasta yläkulmasta alkaen. Samoin neliölle on annettu css –luokka (nelio). (W3schools SVG.)

Css-luokan antaminen svg-elementille nousee oleelliseksi D3:een perehdyttäessä, sillä luotaessa datan visualisointia, käyttää D3 niitä tyypillisesti datan määrittämisessä tiettyyn ryhmään tai kategoriaan kuuluvaksi. Jos data kuuluisi johonkin toiseen ryhmään, sille annettaisiin jokin toinen luokka, ja luokkaa vastaavasti visuaalisella ilmeellä jokin kategorian yksilöivä piirre, esimerkiksi väri.

Yhden svg –säilön sisään voi antaa useita svg –elementtejä (<rect><circle> ym.), joita puolestaan voi ryhmitellä <g> -elementin sisään. Sidottaessa svg –elementtejä tietyn <g> -elementin sisään, on niitä mahdollista manipuloida yhtenä joukkona, esim. Siirtää kaikkia yhdellä komennolla tietty matka oikealle. (Meeks 2015, 23.)

3.3.2 SVG Path

SVG –spesifikaatioon kuuluvalla <Path> -elementillä voidaan piirtää viivoja ja käyriä. Viivat ja käyrät piirretään peräkkäin käyttäen virtuaalista kynää, jossa seuraavan viivan tai käyrän piirtäminen voidaan aloittaa edellisen loppupisteestä. Virtuaalisen kynän piirtäminen tapahtuu elementin ’d’ –attribuutissa. Piirtäminen tapahtuu SVG –elementin sisäisessä koordinaatistossa. Viisi ehkäpä oleellisinta piirtokomentoa ovat:

M	Siirrä virtuaalinen kynä alkupisteeseen.
L	Piirrä suora viiva annettuun pisteeseen.
A	Piirrä käyrä (ellipsin osa) annettuun pisteeseen.
V	Piirrä annettu matka suoraan vertikaalisesti
H	Piirrä annettu matka suoraan vertikaalisesti

Koodin alussa, d-attrubuutin sisällä virtuaalinen kynä asetetaan aina alkupisteeseen kirjaimella ’M’. Käytettäessä isoja kirjaimia kyseessä on aina absoluuttisesta koordinaatistosta, pieniä kirjaimia käytettäessä (’m’, ’l’, ’a’) viivan loppupisteen koordinaatit lasketaan suhteellisesti virtuaalikynän senhetkisestä sijainnista. Esimerkki:

```
<svg>
        <path d="M50,50                 (1
                 A30,30 0 0,1 35,20     (2
                 L100,100               (3
                 M110,110               (4
                 L100,0"                (5
        style="stroke:#660000; fill:none;"/>
</svg>
```

Yllä oleva svg-koodi luo seuraavanlaisen kuvan:

Kuva 5. Svg –path elementti (Jenkov, SVG).

Esimerkissä ’d’ –attribuutin sisällä ensimmäisellä rivillä virtuaalikynän lähtöpisteeksi asetetaan piirtokomennolla ’M’ SVG -koortinaatiston piste 50,50, josta lähtien toisella rivillä piirtokomento ’A’ muodostaan käyrän pisteeseen 35,20 (kuudes ja seitsemäs parametri). Komennon ’A’ käyttöä ymmärrettäessä auttaa hahmottamaan sen seikan tiedostaminen, että lopputulemana on aina ellipsin osa. Matemaattisesti on todistettavissa että tietynkokoinen ja –mallinen ellipsi voidaan voidaan piirtää kahden annetun pisteen

kautta kahdella (O'Reilly, SVG essentials) tavalla. Tietynmallisen ja –kokoisen ellipsin määrittelee yksiselitteisesti ellipsin sekä vertikaalisen että horisontaalisen säteiden pituudet, jotka piirtokomennon 'A' kaksi ensimmäistä parametriä määrittelevät. Neljäs parametri määrittelee piirretäänkö käyrä määritellyn ellipsin pidempää vai lyhyempää reittiä – esimerkissä arvo nolla valitsee lyhyemmän reitin. Kolmannella ja viidennellä parametrillä voidaan käyrää rotatoida tai muodostaa sen peilikuva (w3Schools, svg tutorial).

Kolmannella rivillä piirretään suora viiva (L) virtuaalikynän senhetkisestä pisteestä (35,20) pisteeseen (110, 110). Neljännellä rivillä virtuaalikynä asetetaan uuteen paikkaan, josta alkaen viidennellä rivillä piirretään uusi suora viiva.

Opinnäytetyön sovelluksessa toteutetaan <path> -elementillä nuottien visualisoinnit luomalla jokaista sovelluksen nuottia vastaava oma piirtofunktio, joka palauttaa merkkijonona sen 'd' –attribuutin arvon.

3.3.3 Mihin D3 on tarkoitettu

D3 ei ole vain kaavioiden ja diagrammien esittämiseen tarkoitettu kirjasto – siihen löytyy helpommin käytettäviä kirjastoja – vaan datan monipuoliseen prosessointiin tarkoitettu kirjasto, sisältäen useamman DOM –elementin vuorovaikutteisen käsittelyn ja animoinnin. D3 taipuu mitä monipuolisimpien visualisointien toteuttamiseen, mutta tässä opinnäytetyössä rajoitutaan näkökulmaan, joka mahdollistaa rytminnuotinnuksen toteuttamisen. Suunnitteluperiaatteena on luoda joukko tietyn tyyppisiä SVG –elementteja vastaamaan tiettyä dataryhmää, esimerkiksi kategoriaa. Tällöin jokaista kategoriaa vastaisi omanlaisensa visuaalinen SVG –elementti, jolloin ne olisivat helposti erotettavissa graafisessa esityksessä. Asiat, joita D3:n avulla useinkin suunnilleen tehdään, voidaan mielestäni hahmottaa seuraavasti:

1. On olemassa jokin HTML –koodi pohjana
2. Sille osalle ohjelmakoodia, joka on käyttää D3:a apunaan annetaan tietty datajoukko (json, tsv, csv, JavaScript –taulukko, ym.). Data harvemmin on raakana

sellaisessa muodossa että sitä suoraan voitaisiin käyttää ohjelmassa, vaan yleensä se vaatii esijärjestelemistä ja suodattamista tarkoituksenmukaiseen muotoon.

3. D3:a käyttävä ohjelmakoodi luo SVG –säilön HTML –koodin sisälle.
4. D3: a käyttävä ohjelmakoodi jaottelee saamansa datajoukon ryhmiksi halutun kriteerin perusteella.
5. Jokaiselle ryhmälle luodaan sitä vastaava SVG –elementti.
6. Jokaiselle alkiolle kussakin ryhmässä luodaan instanssi ryhmän SVG –elementistä.
7. Pohjautuen data -alkion joihinkin arvoihin, sille annetaan sijainti ohjelman toimintalogiikan mukaan koordinaatistossa (SVG –säilö).
8. Lisätään mahdolliset animoinnit ja vuorovaikutus näytettävien data –alkioiden kanssa, Esim. Näytettävän Datajoukon vaihtamiseen tai suodattamiseen saattaa olla SVG –elementin vierellä omat DOM –elementtinsä (painikkeet, pudotusvalikko, ym).
9. Lisätään näytettävän datan hahmottamista ja vertailua tukevia visuaalisia elementtejä, joiden avulla pystyy luontevasti saamaan käsityksen datan merkityksestä, esimerkiksi akseleita arvolukemineen.

Visualisointiprojektit joita D3:n avulla tehdään sisältävät usein valmiin HTML –pohjan, johon dynaamisesti lisätään dataa esittävät DOM & SVG –elementit (Qi Zhi, 36). D3 tarjoaa metodeja, joilla voidaan käsitellä useampia edellä kuvattuja vaiheita yhdessä, tarkemmin ottaen sen mahdollistaa metodien ketjuttaminen (method chaining)(Qi Zhu, 15-21), joka on JavaScriptin yhteydessä erityisen suosittu ohjelmointitapa. Tähän palataan myöhemmin sulkeuma –suunnittelumallin yhteydessä.

3.3.4 DOM –elementtien valinta

D3 tarjoaa DOM –elementtien valitsemiseen kaksi metodia: d3.select(param) ja d3.selectAll(param). Ensimmäisellä metodilla valitaan yksi DOM –elementti, jälkimmäisellä kaikki ne elementit, jotka täyttävät parametrina annetut ehdot. Valinta perustuu samoihin sääntöihin joita käytetään css –tiedostossa (Qi Zhi, 24). Esimerkiksi jos halutaan valita -elementti jonka id –attribuutti on asetettu arvoon ’99’:

d3.select("div#99") → * *

Tämä voi olla hyödyllistä jos halutaan manipuloida jotain yksittäistä DOM –elementtiä, tyypillisin käyttötapaus on kuitenkin käyttää selectAll() –metodia luokkien kanssa. Kaikki tietyllä css –luokalla visualisoidut elementit HTML –sivun <body> -osiossa voidaan valita komennolla:

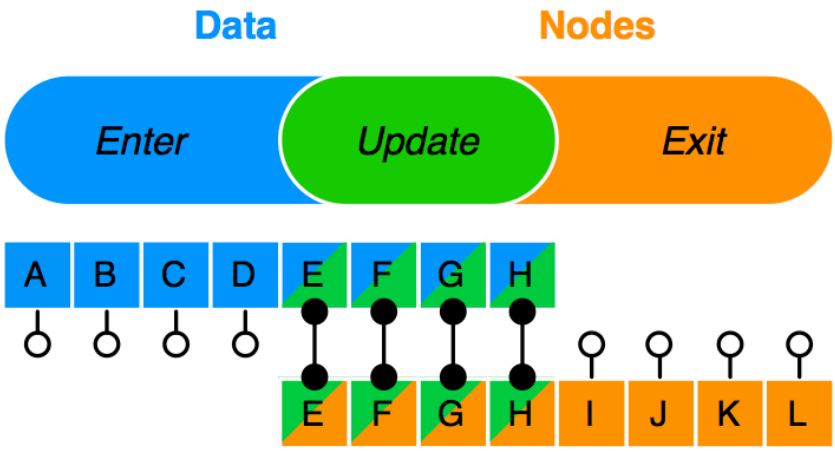
Var elems = d3.select("body").selectAll("div.someClass")

3.3.5 SVG elementtien sitominen dataan

Kun tietyt DOM –elementit on valittu, on niihin sidottava elementtejä vastaava data:

elems.data([14,19,23,23,27,28,25, 30]);

Tässä vaiheessa herää kysymys: data-alkioita on seitsemän, entä jos sivulla on vähemmän luokalla 'someClass' määriteltyjä elementtejä? Nämä voidaan, erikoista kyllä, luoda lennosta valinnalle koodissa myöhemmin. Jos haluttaisiin luoda kappaleessa 3.4.1 esitetyt yhdeksän vaihetta puhtaalla (vanilla) JavaScriptillä koodista tulisi melko monimutkaista. D3 puolestaan on suunniteltu käsittelemään datan ja DOM –elementtien linkittämistä mahdollisimman luontevasti. Sen keinon ymmärtämistä jolla tämä tapahtuu, valaisee valinnan alkioiden visualisointi. D3.SelectAll() – metodilla on muuttujaan asetettu joukko, tätä kutsutaan valinnaksi (selection). On D3:n sisäinen asia miten tämä on toteutettu (taulukko, kokoelma), koodaajan tarvitsee ymmärtää vain toteutuksen julkinen rajapinta. Joukon alkiot kuuluvat kulloisellakin hetkellä yhteen kolmesta ryhmästä, joiden käsittelyyn rajapinnassa on omat metodinsa. Ryhmää voi kutsua myös osavalinnaksi(eng. subselection). (Original D3 paper.).



Kuva 6. Valinnan alkioiden tila osavalinnoissa(Original D3 paper). Alkio kuuluu aina tasan yhteen osavalintaan.

Kun selectAll() valinta on suoritettu kuuluu valinnan kukin yksittäinen alkio yhteen seuraavista kolmesta ryhmästä:

- Enter: alkiolle on annettu data-arvo, muttei DOM –elementtiä (sininen)
- Update: alkiolle on annettu sekä data-arvo että DOM –elementti (vihreä)
- Exit: Alkiolla on DOM –elementti, muttei data –arvoa (oranssi)

Kutakin ryhmistä D3 manipuloi yhtenä joukkona, ts. Selection -objektin useat metodit on kohdennettu toimimaan jonkin tietyn edellä esitetyn ryhmän kanssa. Tämä poikkeaa selkeästi ohjelmoijan yleisesti käyttämästä imperatiivisesta ohjelmoinnista jossa iteroidaan yksityiskohtaisesti kokoelman jokainen alkio niiden käsittelemiseksi (Qi Zhu, 40). Enter -ryhmään alkio kuuluu kun sille on annettu data (selectAll('someClass').data), mutta ei ole vielä luotu sitä vastaavaa DOM –elementtiä. Update-ryhmään kuuluvat alkiot joilla on jo sekä data että DOM –elementti. Update-ryhmällä on myös linkit sekä Enter – ja Exit –ryhmiin (D3 wiki, selections. 2016) alkioiden tilan vaihtamisen ja DOM – elementtien kierrättämisen mahdollistamiseksi. Exit -ryhmään alkio kuuluu, kun sillä on DOM –elementti muttei dataa – tämä on mahdollista tilanteessa jossa jo esitetystä visualisoinnista poistetaan dynaamisesti joitain data-arvoja, eikä grafiikkaa ole vielä sen jälkeen päivitetty. Jaottelua kutsutaan Enter-Update-Exit –malliksi (Qi Zhu, 40) tai yleiseksi päivitysmalliksi (general update pattern, Bostock). Tarkastellaan asiaa koodin avulla:

```
Var ourData = [20, 18, 17,20,26,29,25];
var canvas = d3.select("body")
        .append("svg")
        .attr("width", 600)
        .attr("height", 800);
```

Yllä oleva koodi lisää HTML –sivulle SVG –pohjan. Rakennetaan koodi kolmessa vaiheessa, ensin Enter –ryhmän manipulointi:

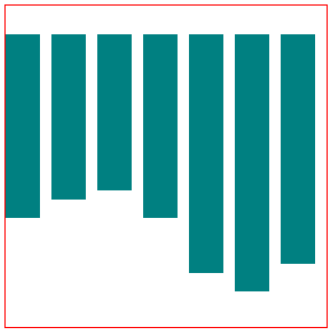
```
// 1) Enter:
var bars = canvas.selectAll("rect.myClass")   // 1) palauttaa (usein tyhjän) valinnan
.data(ourData)              // 2) luo datan valinnalle
.enter()            // 3)  palauttaa ne data-alkiot, joilla on data, muttei DOM-elementtiä
.append()           // 4)lisää edellisen rivin valinnalle DOM-elementit (<rect>)
```

Yllä olevassa koodissa rivi yksi määrittelee sen valinnan, jota myöhemmät muuttujaan ketjutetut metodit sitten manipuloivat. HTML –sivulla joko voi olla tai olla olematta 'myClass' –luokalla määriteltyä <rect> -elementejä, nämä valitaan rivillä yksi, jos elementtejä ei ole yhtään, on joukko tyhjä. Selection -objekti on nyt luotu ja asetettu muuttujaan 'bars'. Rivillä kaksi joukolle annetaan data, samalla joukkoon muodostuu kokoelma alkioita, joilla on annettu 'ourData' –taulukon data-arvot: Mikäli sivulla oli jo valinnan ehdot täyttäneitä DOM –elementtejä, nämä tulevat osaksi valintaa, tarkemmin ottaen ne linkitetään annettuihin data-alkioihin ja siirtyvät update-osioon. Kolmannella rivillä valitaan pelkästään dataa sisältävät alkiot, eli ne joille ei riittänyt sivulla jo valmiina olevia DOM –elementtejä: näille annetaan neljännellä rivillä DOM –elementti (<rect>) luokkatunnuksineen('myClass'), jolloin myös ne siirtyvät update-osioon. Toisessa vaiheessa manipuloidaan Update –osiota, jossa DOM -elementeille annetaan ominaisuuksia niiden dataan perustuen.

```
// 2) Update:
```

```
bars.attr("height", function(d){return d * 10;})//  1) korkeus data-arvon funktiona
 .attr("width", 30)
.attr("x", function(d,i){return i * 40;})    //arvo perustuen indeksiin data-taulukossa
.attr("y", function(d,i){return 50;})
.attr("fill", "teal");    // palkille väri
```

Enter –vaiheen jälkeen valinnan kaikille valinnan alkioille on annettu sekä data että DOM, joten riviltä kaksi alkaen voidaan antaa valinnan alkioille ominaisuuksia ketjutetuilla metodeilla. Jokaisella rivillä lisätään joukon alkioihin yksi ominaisuus. Jokaisen rivin koodi myös palauttaa saman Selection –objektin, johon siis seuraavalla rivillä voidaan lisätä jälleen yksi ominaisuus. Ominaisuuden arvo voidaan laskea myös funktiolla, joka saa parametreinaan datan arvon(d) ja indeksin(i) valinnalle annetusta data-taulukosta: näin tapahtuu esimerkin ominaisuuksille 'height' ja 'x'. Kun koodin ajaa, on tuloksena pylväiden korkeudet taulukon data-arvojen mukaan:



Kuva 7. Svg -koordinaatisto alkaa vasemmalta ylhäältä ja kasvaa oikealle ja alas.

Viimeisessä kolmannessa Exit –vaiheessa valitaan joukon alkiot joillla on DOM-elementti muttei dataa ja poistetaan ne. Tämä on tarpeellista silloin, kun data –joukkoa on muutettu ja visualisointi päivitetään uudelleen.

```
// 3) Exit
bars.data(ourChangedData)
.exit()     // valitsee DOM –elementit joihin ei ole sidottu dataa.
.remove();
```

Varmaankin tyypillisin web –sivuilta julkisesti käytettävää dataa sisältävä tiedosto muoto on JSON (Qi Zhu, 48). Esim.

```
Var lampotilat = [
  {vrk: ”ma”, ”lampotila”: 14}, {vrk: ”ti”, ”lampotila”: 19},
  {vrk: ”ke”, ”lampotila”: 23}, {vrk: ”to”, ”lampotila”: 23},
  {vrk: ”pe”, ”lampotila”: 27}, {vrk: ”la”, ”lampotila”: 28},
  {vrk: ”ma”, ”lampotila”: 25}
]
```

Objekti –taulukko voidaan antaa valinnan dataksi aivan samoin kuin primitiiviarvoja sisältävä taulukko (bars.data(lampotilat)). Tällöin visualisoitaessa joukon alkioiden ominaisuuksia funktiolla viitataan arvoon kuten JSON objektiin yleensäkin; funktion d parametri saa tällöin arvoksi objektikokoelman alkion:

```
bars.attr("height", function(d,i){return d.lampotila * 10;})
```

Käytettäessä funktiota arvon laskemiseen käytetään parametreina vakiintuneen tavan mukaan’d’ ja ’i’ kirjaimia.

Taulukon data-alkiot voidaan ryhmitellä alkioiden jonkin ominaisuuden mukaan `d3.nest` –funktiolla. Jos esimerkiksi edellisen esimerkin taulukkomuuttujassa olisi kaikkien helmikuun päivien (28 kpl) lämpötilat, voidaan ne jakaa ryhmiin vuorokauden perusteella seuraavanlaisella koodilla:

```
var dataByVrk = d3.nest()
   .key(function(dataItem) { return dataItem.vrk; } )   // ryhmittelyperuste
   .entries(lampotilat);      // datajoukko, jonka alkiot halutaan ryhmitellä
```

Tuloksena olisi seitsemän objektia json-objektiliteraalissa, joista jokaisella olisi avaimen määrittelemä kenttä ’key’, sekä taulukkomuodossa kaikki avaimen määrittelemään ryhmään kuuluvat alkiot tallentava kenttä ’values’.

```
{”key”: ”ma”, ”values”: [
```

{"vrk":"ma", "lampotila": 14},{"vrk":"ma", "lampotila": 15},{"vrk":"ma", "lampotila": 16},{"vrk":"ma", "lampotila": 17}] },

{"key":"ti", "values":[{"vrk":"ti", "lampotila":32},{"vrk": ….jne…….

…{"key":"su", "values": [{"vrk":"su","lampotila":24}, {..}, {..}, {"vrk":"su", "lampotila": 21}] }

] }

Lopputulemassa on siis kaikki samat alkiot kuin alkuperäisessäkin, mutta jaoteltuna alkion tietyn kentän mukaan.

3.3.6 Svg path elementin käyttö d3:ssa

Kappaleessa 3.3.2 havainnollistettiin kuinka <path> -elementin piirto tapahtuu sen 'd' –attribuutille annettavalla virtuaalikynän ohjeilla. D3 sisältää valmiina monia funktoita jotka kykenevät vastaanottamastaan datasta muokkaamaan virtuaalikynälle koodin halutunlaisen graafisen elementin piirtämiseksi. Tälläisiä funktioita ovat esim. line ja arc, jotka piirtävät viivoja ja kaaria, mikäli datan tietyistä kentistä ('x', 'y') löytyvät koordinaatiston pisteet. Valmiit funktiot on tarkoitettu piirto-operaatioiden helpottamiseksi. (Meeks 2015, 109.)

Opinnäytetyössä toteutettvassa sovelluksessa toteutetaan kuitenkin omat piirtofunktiot jokaiselle tuetulle ääntä vastaavalle nuotille. Kun <path> -elementtiin pohjautuvan valinnan 'd' attribuutille annetaan funktio, piirtää se nuotin grafiikan annetun datan koordinaatistotietojen perusteella.

3.3.7 Valinnan tapahtumankäsittely

Valinnan tapahtumankäsittely tapahtuu *on –funktiolla.* Tapahtuman rekisteröinti valinnalle, ja siis kaikille valinnan DOM –elementeille, tulee kirjoittaa seuraavasti:

valinta.on("tapahtumankäsittelijänimi", tapahtumankasittelijaFunktio);

Edellisen kappaleen esimerkille voitaisiin antaa pylväille hiiren klikkaukseen reagoiva tapahtumankäsittelijä joka tulostaa konsoliin vuorokauden lämpötilan:

bars.on("click",

```
function(d,i){ console.log("pylväs nro: " + i + ". " + d.vrk + ",  lämpötila: " + d.lampotila);
```

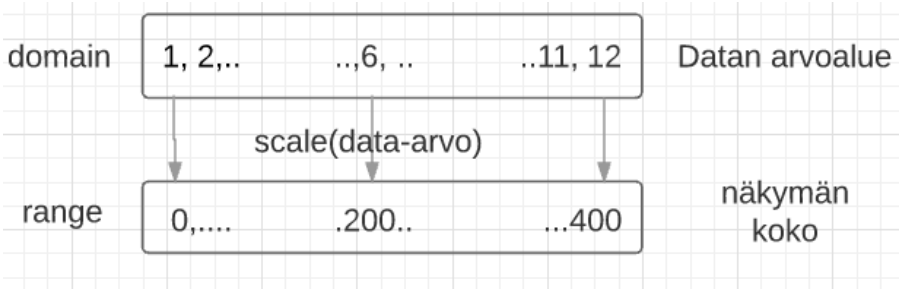
Yllä olevassa on käytetty anonyymia funktiota, mutta *on* –funktion parametriksi voitaisiin myös antaa nimetty funktio.

Muita hyödyllisimpiä tapahtumankäsittelijöitä ovat esimerkiksi: 'dblclick', 'mousedown', 'mouseenter', 'mouseleave', 'mousemove', 'mouseout', 'mouseover' ja 'mouseup' (Qi Zhu, 237).

3.3.8 Skaalaus ja akselit

Datajoukon tietyllä kentällä on pienin ja suurin arvo, toisaalta html –koodiin upotettavalla svg –elementillä on tietty leveys ja korkeus. Scale, Domain ja Range –objektien avulla voidaan datan arvojoukko sovittaa skaalautuvasti svg –elementin kokoon. Domain määrittelee datajoukon arvoalueen rajat, ja range SVG-elementin rajat. Lineaarisessa esityksessä tämä käy seuraavasti:

```
var scale = d3.scale.linear()
.domain([minValue, maxValue])  //
.range([0, svgWidth]);              //
```



Kuva 8. Data -arvojen skaalaus SVG -canvakselle. .domain([1,12]).range.[0,400]).

Funktion `scale` avulla etsitään data-arvoa vastaava kohta svg:n datapiirtoalueen leveydestä tai korkeudesta.

Edellisen kappaleen esimerkkikoodi vaatii vain pieniä muutoksia muuntuakseen pystysuunnassa skaalautuvaksi, lisäämällä muuttuja ja funktio –objekti:

```
canvasHeight = 500;  // kun tätä muuttaa pylväät skaalautuvat pystysuorassa vastavasti
var heightScale = d3.scale.linear()
                .domain([0, d3.max(ourData)])  //
                .range([0, canvasHeight]);  //arvot sovitetaan canvaksen korkeuteen
```

Tällöin muuttujia tarvitsee käyttää svg –objektin ja valinnan 'height' attribuuteissa:

```
canvas.attr("height", canvasHeight);
bars.attr("height", function(d){return heightScale(d);});
```

Myös leveyskoordinaatin laskeminen skaalautuvaksi on melko suoraviivainen toimenpide huomioimalla käytetyn taulukon kokonaispituus svg-elementin leveyteen nähden ja laskemalla tästä pylväiden leveys ja x-suuntaiset koordinaatit.

3.3.9 Sulkeuma (eng. closure)

D3 –kirjaston luoja, Mike Bostock, on esittänyt uudelleenkäytettävien visualisointiin tarkoitettujen komponenttien luomiseen käytäntöä, joka käyttää suunnitteluperiaatetta nimeltä sulkeuma (Bostock). Tavoitteena on että asiakaskoodi kykenisi uudelleenkäyttämään samaa komponenttia joustavasti vaikka sen tietyt ominaisuudet muuttuisivatkin. Esimerkiksi pylväsdiagrammi voidaan esittää useilla tavoin, sen pohjana oleva data tai diagrammin metatiedot voivat muuttua, myös SVG –alustan koko HTML-sivulla voi muuttua. Komponentin tulisi sen julkisen rajapinnan kautta kyettävä vastaamaan mahdollisimman hyvin sitä käyttävän koodin tarpeisiin vaatimusmäärittelyssä esiin tulevilla tavoilla.

Sulkeuma –suunnittelumallin haltuunotto voi mielestäni onnistua javaScriptin perusteet osaavalle kun ymmärtää kolme JavaScriptille tyypillistä ominaispiirrettä: funktioiden objektiluonteen, objektien dynaamisen muunnettavuuden sekä muuttujien näkyvyysalueen ketjutuksen toiminnan. JavaScriptissa kaikki funktiot ovat objekteja jotka voivat pitää sisällään avain/arvo –pareja(Qi Zhu, 17). Funktioiden sisälle voidaan muuttujiin tallentaa myös toisia funktioita, tällöin puhutaan sisäkkäisestä funktioista (Qi Zhi, 18). JavaScript soveltaa muuttujiin leksikaalista näkyvyysaluetta (lexical scoping. Wikipedia, JavaScript). Kun koodissa on muuttuja, objektia tai arvoa johon se viittaa etsitään Ja-

vaScript –kääntäjän toimesta seuraavasti: ensin etsitään viittausta paikallisesti, jos viittausta ei löydy, sen jälkeen etsitään parent-objktista, ja edelleen rekursiivisesti parent-ketjua ylätasoon edeten kunnes viimeisenä etsitään JavaScriptin globaalista objektista (Qi Zhu, 19). Tarkastellaan mitä tämä tarkoittaa sisäkkäisten funktioiden kohdalla.

```
Function chart(){
        var mySvg, myData;
        var width = 720, height=80; // defaults
        function notation(svg){
        mySvg = svg;
        /*generate chart initialization here. */
       }
        notation.width = function(value){
          If (!arguments.length) return width;  // jos ei argumenttia → palauteteaan ominaisuus
          width=value;   // välitettiin argumentti leveydelle →
          return notation;  // palautetaan visualisointikomponentti jossa leveys muutettu
       }
        notation.height = function(value){
          If (!arguments.length) return height;
          height=value;
          return notation;
      my.data = function(value){
          If (!arguments.length) return myData;
          myData=value;
          return notation;
       }
  return notation;   // julkinen rajapinta joka palauteteaan asiakkaalle
}
```

Yllä mukaelma Mike Bostockin (Bostock, 2012) sulkeuma -esimerkistä. Funktion *chart* sisällä on funktio *notation*. Koska JavaScriptissä funktiot ovat objekteja ja dynaamisesti muunnettavissa, niihin voidaan assosioida koodin avulla vielä luontitapahtuman jälkeen paitsi muuttujia ja taulukoita(array), myös toisia objekteja, muun muuassa funktioita (Qi Zhu, 18). Esimerkissä toimitaan näin kolmesti: *notation.width, notation.height* ja *notation.data* – kaikki nämä funktiot assosioituvat samaan *notation*-objektiin, joka palautetaan asiakkaalle ulomman funktion kutsussa:

1) var newNotation =new chart(svgFromHTML); // viittaus chart- funktion sisällä määritel-tyyn notation-funktioon

JavaScriptin kaikille objekteille, siis myös funktioille, luodaan linkitys siihen objektiin, jonka sisällä ne on määritelty (NewCircle, 2014). Kun siis Asiakas on luonut rivillä 1 *newNotation* –muuttujan, joka siis luotiin *chart* –funktion sisällä, on sillä myös viittaus siihen objektiin jonka sisällä se luotiin, eli *chart –funktioon.* Kun objektiin on olemassa yksikin viittaus, ei se joudu JavaScriptin roskienkerääjän uhriksi, vaan pysyttelee elossa (NewCircle 2014). Vaikka ulompaan funktioon ei ole jäljellä mitään suoraa muuttujan viittausta koodissa, tarjoaa sulkeuma eräänlaisen epäsuoran linkin objektiin, objektien ketjutuksen kautta (NewCircle 2014), joka pitää sitä elossa.

Koska kaikki *chart-* funktion sisällä määritellyt funktiot assosioitiin samaan asiakkaalle palautettavaan *notation* –funktioon, tulevat ne osaksi julkista rajapintaa. Kaikki ne ovat myös luotuja saman objektin (chart) sisällä ja omaavat siihen siten linkin. Täten funktion sisällä olevilla muuttujilla on pääsy sen objektin muuttujiin, jonka sisällä ne on määritelty. Closure -mekanismi mahdollistaa toteutuksen kapsuloinnin (encapsulation): toteutus voidaan tehdä ulomman funktion niillä muuttujilla ja funktioilla joita ei assosioida siihen funktioon joka palautetaan (*notation*). Palautettuun funktioon assoisoidaan julkisen rajapinnan funktiot. Toisin kuin useimmat olio-ohjelmointikielet (Java, C++), ei JavaScript tarjoaa sisäänrakennetusti kapsulointia mahdollistavaa mekanismia. Sulkeuma on tyypillisin tapa toteuttaa kapsulointi JavaScriptissa. (NewCircle Training.)

Closure –menetelmässä objektin tietyn ominaisuuden luku- ja kirjoitustoiminnoille ei ole omia metodejaan, vaan metodi on molemmissa tapauksissa sama. Metodin sisällä tutkitaan onko sille välitetty parametriä (`If (!arguments.length)`). Mikäli argumentti puuttuu, palautetaan ominaisuuden arvo, esimerkiksi `width`. Mikäli argumentti on annettu, muutetaan ominaisuuden arvoa, ja palautetaan koko objekti (`notation`).

Kaikki *chart-* funktion sisäkkäiset funktiot palauttavat sulkeuman saman objektin, julkisen rajapinnan muodostaman funktion *notation.* Tämä mahdollistaa sulkeumaa käyttävälle asiakkaalle metodien ketjutuksen:

```
2) newNotation.data(myJSON)   // annetaan ominaisuuksia komponentille
.width(domWidth)    // palauttaa notation –objektin, joten voidaan kutsua suoraan uudelleen
.height(domHeight);
```

Rajapinnan kautta komponenttia voidaan muokata ketjuttamalla seuraava rajapinnan metodi pisteellä erotettuna edellisen perään, mikä nopeuttaa ohjelmointia. D3 – kuten uset muutkin JavaScript-kirjastot (NewCiecle Training) on toteutettu kautta linjan sulkeuma -suunnittelumallia käyttäen, mikä näkyy D3:a käyttävien ohjelmien koodissa ketjutuksen muodossa.

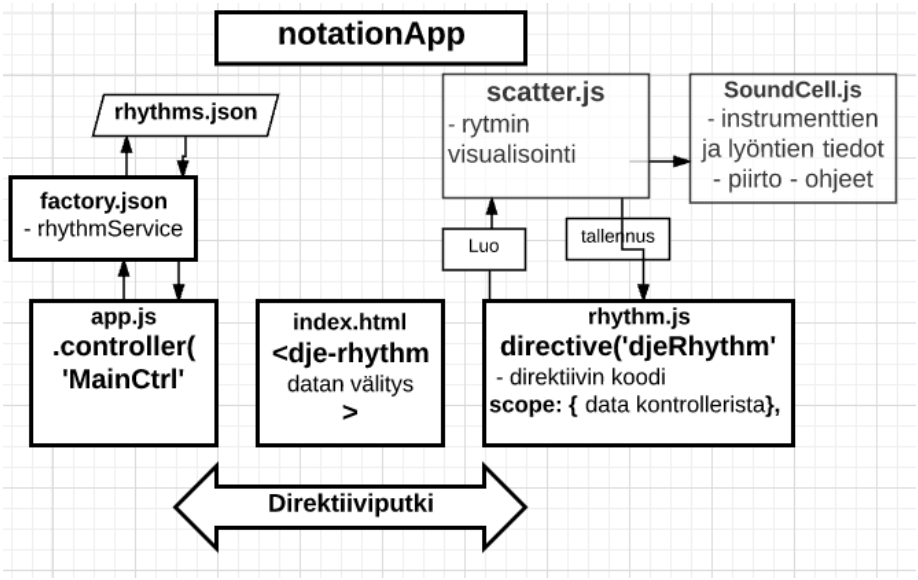
Luotaessa uusi funktio sulkeumaa käyttäen kannattaa kiinnittää huomiota sen luontitapaan. JavaScriptin funktio saa luonnin yhteydessä implisiittisesti aina muassaan avainsanalla *this* merkityn objektin. *This* –objekti kertoo kontekstista jossa funktio luotu. *New* –avainsanaa käytettäessä this assosioituu luotuun objektiin, ja objektin sisällä olevat muuttujat (var height, width, ym.) linkittyvät kyseiseen objektiin (funktio chart). Mikäli *new* –avainsana jätetään pois ei this –kontekstia ole määritelty, ja *this* linkittyy javaScriptin globaalin objektiin (selaimissa window), jonne luodaan silloin funktion sisällä luodut objektit (width, data, ym.). Tämä saattaa aiheuttaa hämentäviä virhetilanteita mikäli ohjelmassa muuallakin on luotu samannimisiä muuttujia globaaliin objektiin, koska yhdellä nimellä yhdessä kontekstissa (globaali) on mahdollista olla vain yksi objekti. (Cockford.)

4 Ohjelma

Sovelluksen vastuualueet AngularJS ja D3:n välillä ovat seuravat: angular vastaa rytmin valinnasta, uuden rytmin luonnista ja rytmin tallentamisesta muutosten jälkeen json –tiedostoon. D3:lla toteutetaan komponentti joka näyttää valitun rytmin visualisoinnin sekä mahdollistaa muutokset siihen. Angular –osan ulkopuolelle kuuluu myös JavaSriptillä toteutettu SoundCell –funktioliteraali, johon on tallennettu sovelluksen tukemien lyöntien tiedot, sekä niiden piirto ohjeet.

4.1.1 Rakenne

Ohjelma koostuu kuudesta kooditiedostota, yhdestä tekstitiedostosta sekä paikallisesti importoidusta d3.js –lähdekoodista (versio 3.4). Alla kaavio käytetyistä tiedostoista:



Kaavio 5. Sovelluksen rakenne ja komponenttien välinen vuorovaikutus. Angular –komponenttien reunukset on lihavoitu.

Sovellus käyttää hyödykseen sekä Angularia että D3:a, kaikki muut tiedostot on kuuluvat angularin vaikutuksen piiriin paitsi visualikomponentin toiminnasta vastaava scatter.js ja lyöntien tietoja varastoiva SoundCell.js. Kaaviosta on yleisellä tasolla hahmotettavissa ohjelman toimintalogiikka. Tarkastellan tiedostoa index.html:

```
<html ng-app="notation">
```

```
// sisällytetään käyetyt *.js tiedostot:
<script src="files/d3.js" charset="UTF-8"></script>
<script src="files/angular.js" charset="UTF-8"></script>
<script src="src/factories.js"></script>
<script src="src/app.js"></script>
<script src="src/rhythm.js"></script>
<script src="src/soundCells.js"></script>
<script src="src/scatter.js"></script>
<link href="src/app.css" rel="stylesheet">
<body><div ng-controller="mainCtrl">  //sovelluksen kontrolleri
Rytmit:
<select ng-model="selRhythm" ng-options="rhy.name for rhy in rhythms">
</select>
<dje-rhythm rhythm="selRhythm" saverhythm="saverhythm" instruments="instruments">
</dje-rhythm>
</div>
</body>
</html>
```

Heti alkuun <html> -tagissa luodaan angular sovellus **`myNotation,`** joka luodaan *app.js* –tiedostossa sen ensimmäisellä rivillä. Toisella rivillä luodaan kontrolleri joka hallinnoi index.html –sivun -elementin sisällä olevaa koodia:

```
angular.module('notation', [])
.controller('mainCtrl', ["$scope", "d3", "SimpleHttpLoader", "rhythmService", "instrument-
Service" ,
function ($scope, d3, SimpleHttpLoader, rhythmService, instrumentService) {
```

```
$scope.selRhythm = '';  // use in HMTL with two-way binding for selected rhythm
$scope.rhythms = {};   // data source for rhythms , use in dropdown
```

Kontrolleri käyttää hyväkseen kolmea itse luotua palvelua: 'd3' ' SimpleHttpLoader', ja 'rhythmService', jotka kaikki on luotu factories.js –tiedostossa. 'd3' –palvelu yksinkertaisesti tarjoaa käytettäväksi sen d3 –nimisen muuttujan, jonka d3.js –lähdetiedosto sulkeuma –sunnittelumallin toteutuksella palauttaa. 'rhythmService' hakee ohjelman käynnistyksen yhteydessä kaikkien tallennettujen rytmien notaatiot tiedostosta 'rhythms.json' (Liite 1) malliin sidottuun *$scope.rhythms* –muuttujaan. JSON –tiedosto on taulukkomuodossa, jossa jokainen rytmi on esitetty omana objekti-literaalinaan. Eräs angularin sisäänrakennettua kaksisuuntaista sidonta-direktiiviä 'ng-model' hyväksikäyttävistä elementeistä on pudotusvalikko (select):

```
<select ng-model="selRhythm" ng-options="rhy.name for rhy in rhythms">
</select>
```

Koska *rhythms* –taulukkomuuttuja on sidottu $scopeen sitä voidaan käyttää datalähteenä näkymän pudotusvalikkoon sovellettavalla angularin *ng-options* –direktiivillä: jokaisen rytmin nimi näytetään (*rhy.name for rhy in rhythms*) pudotusvalikossa. Valittu rytmi asettuu direktiivissä *ng-model* esitetyn muuttujan *selRhythm* arvoksi. Jos esimerkiksi käyttäjä valitsi rytmin 'soli', tulee *selRhyhtm* – muuttujan arvoksi taulukkomuotoisesta rhythms.json - tiedostosta se alkio kokonaisuudessaan, jonka *name* –ominaisuuden arvo on 'soli'. Alkio pitää sisällään rytmin soli kaikki tiedot nuotinnuksineen.

Seuraavana index.htm –tiedostossa käytetään itse luotua direktiiviä. Direktiivin nimestä ('djeRhythm') on isot kirjaimet muutettava pieniksi ja laitetteva niiden eteen väliviiva:

```
<dje-rhythm rhythm="selRhythm" saverhythm="saverhythm" instruments="instruments">
</dje-rhythm>
```

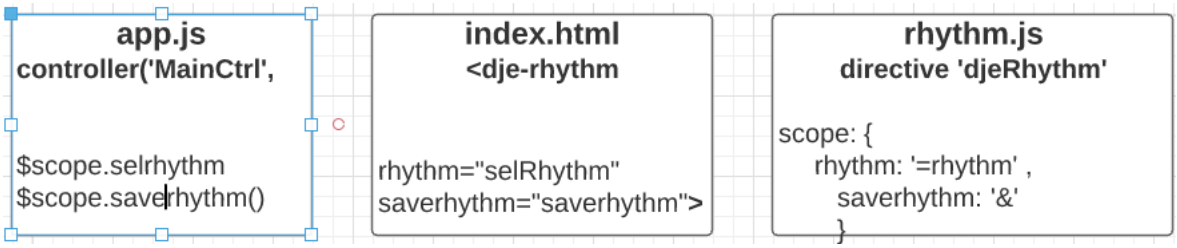
Direktiivi saa attribuuttina arvokseen edellisellä rivillä pudotusvalikosta valitun rytmin. Pudotusvalikko asetti rytmin kontrollerin $scope.selRhythm –muuttujan arvoksi, joten sitä voidaan käyttää direktiivielementin sisällä. Direktiivin koodi on rhythms.js –tiedostossa:

```
angular.module('notation.)
.directive('djeRhythm', [    // direktiivin nimen isoa kirjainta edeltää – html:ssä!
   function($http){
    // DDO –objecti (Directive Definion Object)
   return {
    restrict: 'E',      // to be used only as an element in HTML
    scope: {
      rhythm: '=rhythm',   // object passed from HTML with reference (two-way)
      instruments:'=instruments',
      saverhythm: '&'   // method in outer world (app.js) that can be called
    },
      require:'djeRhythm',  // must reference itself in order to use controller in link-function!
       controller: function ($scope) { ….
          this.buidJSON(rawRhythm) { // muuntaa rytmin JSON:in sopivaan muotoon }
      …… },
    link: function(scope, iElem, attrs, rhythmCtrl)
       //add svg to the directives DOM TODO size dynamically
      svg = d3.select(iElem[0]).append('svg')
                .style("border", "solid 3px gray") // to be seen during development
            // Watch changes of the data passed in scope
            scope.$watch('rhythm', function(newVal, oldVal, scope) {
              if(rhythmCtrl.isDirty())  { saveRhythm(sPlot.data);
              rhythmCtrl.buildJSON(scope.rhythm);
               }
            }, true);
```

Yllä on direktiivin toiminnan ymmärtämisen kannalta oleellisimmat viisi DDO –objektin kenttää:

1) ***restrict***: 'E', rajoittaa direktiivin käytön HTML –koodissa koskemaan ainoastaan <elementtejä>. (<dje-rhythm……..></dje-rhythm>)

2) ***scope*** -kentässä luodaan direktiiville kokonaan oma eristetty scopensa. Direktiivin scopelle kuitenkin annetaan viittaus objektiin joka <dje-rhythm> -elementin atrribuutilla 'rhythm' -välitetään, eli valitun rytmin dataan json-muodossa. Samoin välitetään kontrollerista funktio, jota direktiivistä voidaan kutsua:



Kuva 9. Putki direktiivin ja ulkomaailman välillä.

Direktiivillä on nyt pääsy kontrollerin muuttujaan ***selrhythm***. Samoin direktiivi kykenee kutsumaan kontrolerin funktiota ***saverrythm***. Näitä käytetään paitsi välittämään valittu rytmi kontrollerin ja index.html:än kautta kulkevaa putkea pitkin direktiiville, myös toiseen suuntaan välittämään rytmin data takaisin kun rytmin nuotinnusta on muutettu.

3) ***controller*** -kentässä tapahtuu tietojen ohjaus eri ohjelmakomponenttien kesken. Tänne voidaan tallentaa muuttujia, mm sPlot, johon talletetaan nuotinnuksesta vastaava objekti.

4) ***require*** kentän arvoksi tulee direktiivi itse, tämä on välttämätöntä, jotta ***link*** -funktio kykenee kommunikoimaan ***controller*** –funktion kanssa!

5) **`link`** -kenttä ajetaan kertaalleen HTML:ssä luotua instanssia kohden ja siinä luodaan DOM –manipulointi ja tapahtumankäsittelystä vastaava koodi. Neljäntenä parametrina annetaan se direktiivi joka require –kentässä luotiin, ja jonka controllerin julkiseen rajapintaan (**`this`**- objekti) link –funktiolle halutaan antaa pääsy, tässä tapauksessa tapauksessa saman direktiivin julkiseen rajapintaan.

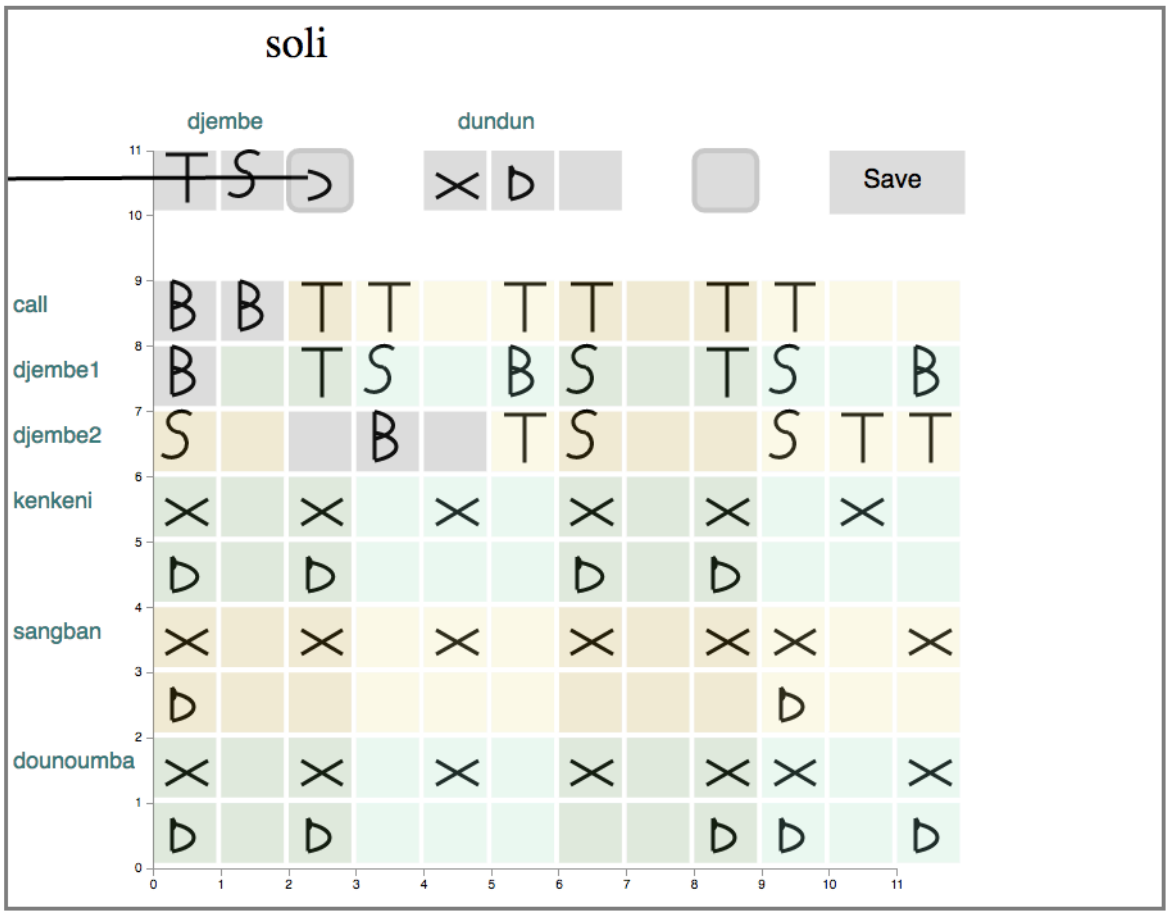
`Link –`funktiossa luodaan svg-elementti, johon nuotinnuksen visualisointi luodaan. Svg –elementti luodaan parametrina saatuna direktiivi-elementtiin (<dje-rhythm>). Koodi *d3.select(iElem[0]).append('svg')* manipuloi DOM –puuta, selaimen web-työkaluilla tarkalteltaessa havaitaan että se on luonut seuraavanlaisen osan:

```
<div>
<dje-rhythm rhythm="selRhythm" instruments="instruments" class="ng-isolate-scope">
    <svg transform="translate(50, 0)" style="border: 3px solid gray; width: 900px; height: 700px;">
```

Angular tarkkailee mallin muuttujan arvon muutosta $watch –funktiolla. Tämä on mahdollista korvata ylikirjoittamlla (overwrite) oma funktio kyseisen tilalle, ja suorittaa haluttu funktio muutoksen tapahtuessa. Tämä tehdään **`Link –`**funktiossa asettamalla watch() –funktio kuuntelemaan muuttujan **`$scope.rhythm`** arvon muutosta. Kun pudotusvalikosta valitaan uusi rytmi, heijastuu se DDO:n **`scope–`**kentässä tehdyn määrityksen mukaan suoraan direktiivin **`$scope.rhythm`** muuttujaan ja watch-funktiolle toisena argumenttina annettu funktio ajetaan. Funktiossa tarkastatetaan ensin onko mahdollisesti aiemmin näytetyn rytmin nuotinnusta muokattu, joilloin kysytään halutaanko se tallentaa. Uuden rytmin tiedot annetaan argumenttina kontrollerin julkisessa rajapinnassa määritellylle buildJSON(rawRhythm) –metodille.

4.2 Haluttu data visualisointikomponentille

Alla on kuva sovelluksen käyttöliittymästä tilanteessa, jossa se näyttää rytmin soli.json tiedot.



Kuva 10. Otos ohjelman käyttöliittymästä. Rytmin soli nuotinnus.

Rytmin nimen yläpuolella oleva rivi nuottimerkkejä ei kuulu itse rytmiin, vaan on nuottipaletti, josta instrumentin haluttu nuotti valitaan, ja jonka valinnan jälkeen kyseistä nuottia voidaan napsautella haluttuihin paikkoihin. Rytmin nuotit alkavat riviltä ’call’.

Opinnäytetyössä toteutettavassa prototyypissä käytössä olevat instrumentit ovat aina samat: ’call’, ’djembe1’ ,’djembe2’, ’kenkeni’, ’sangban’ ja ’dounoumba’. Näistä ’call’ ei edes ole instrumentti, vaan djembellä soitettava kutsu, eräänlainen rytmin aloitusmerkki joka soitetaan vain kerran, jonka jälkeen muut instrumentit aloittavat soiton jatkaen sitä toistuvasti kuvassa nähtyjen nuottien mukaan – vasemmalta oikealle, vasemmalta oikealle, uudelleen ja uudelleen. Todellisuudessa rytmi ei toki ole niin toisteisen tylsä, vaan siinä tapahtuu useimpien instrumenttien kohdalla variaatioita perusrooliin nähden. Lisäksi soittajat soittavat sooloja, paitsi instrumentoituja, myös vamiiksi harjoiteltuja, ja siten nuotinnettavissa olevia. Lisäksi käytettävien instrumenttien kokoonpano voi vaih-

della, näytölle on oltava mahdollista lisätä ja poistaa niitä instrumentteja joita halutaan tarkastella.

Data voi siis (lopullisessa versiossa) olla rytmin kaikkien mahdollisten instrumenttien ja roolien osalta huomattavasti laajempi kuin se mitä yhdellä kertaa käyttöliittymässä näytetään. Näin ollen se pitäisi esikäsitellä siten, että se data joka oletuksena halutaan näyttää suodatetaan siitä. Tässä prototyypissä data on kuitenkin aina samassa muodossa ja voidaan välittää suoraan metodille joka muuntaa sen visualisointikomponentin ymmärtämään muotoon. Metodi on **`buildJSON().`**

Rytmin metatiedot instrumenttien äänilähteiden yhteislukumäärää lukuun ottamatta saadaan suoraan raa'an JSON-tiedoston tiedoista:

```
[{
  "name":"soli",            // rytmin metatieto
   "pattern_length":12,  / /rytmin metatieto
   "measure":3,           // rytmin metatieto
  "instruments":{
     "call":{
        "name":"call",
        "instrument": "djembe",
        "length":12,
         "repeat":1,
         "sound_channels":1,
        "notes":[
        {
           "djembe":"slap"
        },
        {
           "djembe":"_empt"
        },
…….} //end of "call"
// Toinen ja neljäs instrumentti
,{"djembe1": {"name":"djembe1","instrument":"djembe", ….jne
```

,{"kenkeni": {"name:"kenkeni", "instrument": "dundun",

….., "sound_channels": 2,

"notes":[{

"dundun" : "open",

"bell" : "open" }, {"dundun": "_empt", "bell":"_empt",……jne

Kaikkien instrumenttien äänilähteiden lukumäärä saadan iteroimalla instrumentit ja ynnäämmällä yksittäisen instrumenttien äänilähteet, samalla voidaan kerätä taulukko jossa on jokaisen instrumentit tiedot. Nuottien parsimiseksi täytyy käydä läpi koko puurakennemallinen JSON –tiedosto.

Direktiiville HTML:stä välitetytty json-tiedosto on *1)* ***taulukko instrumenteja,*** joiden sisällä on instrumentteja kuvaavien kenttien lisäksi *2)* ***taulukkomuodossa instrumentin nuotit***. Nuotit on esitetty instrumentille tietyssä ajallisessa kohtaa kuviota. Lisäksi tässä tietyssä kohtaa instrumentilla saattaa olla *3)* ***useampi äänilähde***. Luonnollista on tällöin käydä tiedosto läpi kolmella sisäkkäisellä silmukalla ja kerätä iteroitaessa halutut tiedot.

```
// 1) Iteroidaan rytmin jokainen instrumentti  (inside builtJSON):
   angular.forEach(rawRhythm.instruments, functi-
on(value, key) {
soundChannels += value.soundChannels;
instruments.push(value) // kerätään instrumenttien tiedot.
 // 2) iteroi kaikki instrumentin nuotit
for(i=0;i<value.notes.length;i++){
// 3)iteroi instrumentin äänilähteet kyseisessä kohdassa
 angular.forEach(value.notes[i], function(value, key){
```

Komento *angular.forEach* saa ensimmäisenä parametrinaan kokoelman iteroitavia JSON -objekteja. Taulukon jokaiseen alkioon sovelletaan toisena parametrina saatua funktiota, jolle annetan argumentteina JSON- objektin avain ja arvo (value, key).

Yllä olevassa koodissa kolmannen sisäkkäisen silmukan sisällä käydään läpi objektia, joka sisältää yhden instrumentin kaikki äänilähteet arvoineen tietyssä kohtaa kuvion pituutta. Esimerkkejä millainen objekti voi tällöin olla:

{ "djembe":"slap" } tai { "bell":"open", "dundun":"bass" }

Alkuperäisessä tietovaraston rhythms.json tiedostossa ääni saa aina jonkin seuraavista arvoista, ts. kyseessä ovat äänet joita ohjelma tukee:

Taulukko 4. Sovelluksen tukemat lyönnit.

Instrumenttityyppi	**Äänilähde**(key)	**isku**	Ääni (value)	**SVG - symboli**
djembe	djembe	avoin	open	S
djembe	djembe	suljettu	closed	T
djembe	djembe	basso	bass	B
dundun	dundun	basso	bass	B
dundun	dundun	suljettu	closed	⊙
dundun	bell	kello	open	x

Kyse on siis avain-arvo-pareista : ’äänilähde’:ääni., nämä yksilöivät lyöntityypin. Sekä äänilähde että arvo tulevat sovelluksessa osaelementeiksi sellaisista data-alkioista, jotka mahdollistavat taulukossa 4 esitetyn nuottien visuaalisen esityksen, vaatimusmäärittelyssä esitettyine tapahtumankäsittelijöineen. Yhden data-alkion arvot ovat taulukkomuodossa esitetyn nuotinnuksen yhden solun taustalla. Seuraava taulukko kuvaa yhdelle data-alkiolle annettavat ominaisuudet ja niiden tarkoituksenmukaisuuden nuotinnuksen visualisointikomponentin kannalta:

Taulukko 5. Rytmin yhden nuotin data-alkion tiedot.

avain	Arvon laskeminen sisäkkäisten silmukoiden sisällä	Tuettava ominaisuus / tapahtumankäsittely visualisoinnissa	Esimerkkejä arvosta
x	Silmukan 2 arvo i	Horisontaalinen sijainti taulukossa.	0,1,2,..kuvion pituus
y	Nuotinnoksessa näytettävät instrumentit iteroidaan järjestyksessä: pidetään kirjaa vertikaalisesta sijainnista läpi silmukoiden.	Vertikaalinen sijainti taulukossa.	0,1,2,….äänilähteiden yhteislukumäärä + 2
sound	Value silmukassa 3	Äänen identifiointi	'open','slap','bass'
soundSource	Key silmukassa 3	Äänilähteen identifiointi	'djembe', 'dundun', 'bell'
inst		Ei tarvita visualisoinnissa. Instrumenttityyppi. Säilön itemiksi SoundCell:issä	'djembe', 'dundun'
instSpe	value.name uloimmassa silmukassa	Vertikaalinen sijainti, tietyn instrumentin poisto nuotinnuksesta.	'call', 'djembe1', 'djembe2', 'kenkeni', 'sangban' ,'dounoumba'
id	patternLen * yPosition + xPosition	Data-alkion yksilöinti kaksiulotteisessa taulukossa sen sijainnin mukaan.	1,2.3,…maxYPosition * PatternLength

Data-alkiot lisätään taulukkomuuttujaan $scope.curRhythm:

```
$scope.curRhythm.push({
    x:i,
```

```
 y: vertPos -rivi,          //  sijainti vertikaalisesti
 sound: value,              // ääni
 soundsource: key,     //sound source: { 'djembe', dunun' tai bell
instrument: muuttujan arvo, //djembe1, djembe2, sangban..
 id:    // muuttujan noteId -arvo,
});
```

Kunkin data –alkion arvot voidaan siis laskea sisäkkäisten iteraatioiden eri vaiheissa. Jotta tämä olisi mahdollista edellyttää se että siinä vaiheessa kun builtJSON- funktiolle annetaan argumenttina instrumentit nuotteineen, on data jo suodatettu halutunlaiseksi visuaalisen esityksen kannalta.

Jokaisessa mahdollisessa kohtaa nuotinnusta (kaksiulotteista taulukkoa) ei toki ole ääntä - silloin kentälle 'sound' annetaan arvoksi 'emp'.

4.2.1 Visualisointikomponentin luonti ja logiikka

Nuottien visualisoinnista ohjelmassa vastaa sulkeumalla toteutettu visualisointipalikka, joka on toteutettu tiedostossa scatter.json. Sulkeuma palauttaa *djeNotation*- objektin, johon assosioidaan visualisointipalikan julkisen rajapinnan metodit. Alustusmetodissaan (initializer) objekti saa parametrina sen svg –elementin johon visualisointielementit d3:n avulla lisätään. Samalla alustusmetodissa luodaan kolme ryhmittelyelementtia <g> : yksi x-akselille toinen y-akselille ja kolmas näytettäville nuoteille ja nuottipaletille (*notesG*). Lisäksi julkisessa rajapinnassa on metodit rytmin metatiedoille: käytetyille instrumenteille, rytmin nimelle, kuvion pituudelle (pattern_length), tahtilajille (measure), näytettävien instrumenttien äänilähteiden yhteismäärälle (soundChannels), svg- elementin leveydelle ja korkeudelle. Nämä metodit ovat kaksisuuntaisia, nillä voidaan sekä lukea että kirjoittaa ominaisuuksien arvoja. Lisäksi julkisena lukumetodina on isDirty(), jolla voidaan tutkia onko nuotinnusta muokattu viimeisen tallennuksen jälkeen. Svg –elementin leveys ja korkeus on saatavissa `link` –metodin parametrista 'iElem' (Kurz

2015, 28). Tällöin svg-canvaksen leveyden määräisi **<dje-rhythm>** -elementin koko sivulla ***index.html***.

Nuotinnuskomponentilla on lisäksi julkinen metodi ***reset()***, jolla tyhjätään näytettävän vanhan rytmin tiedot. Tätä kutsutaan ensimmäisenä annettaessa komponentille uusi rytmi. Vielä yksi julkinen motodi on ***create(),*** jolla nuotinnuksen visualisointi varsinaisesti luodaan. Komponentti ***sPlot*** luodaan ***rhythm.json***- tiedostossa niillä ominaisuuksilla jotka pääasiassa ***builtJSON()*** –metodissa laskettiin:

```
sPlot = !sPlot ? new d3.djeNotation.scatPlot(): sPlot;  // luodaan uusi jos ei vielä ole
sPlot(svg)        // alustus
.reset()         // wanhan datan ja visualisointielementtien tyhjennys
 .width(width) .height(height)
 .data($scope.curRhythm)
 .rhythmName(rawRhythm.name)
 .patternLength(rawRhythm.pattern_length)
 .soundChannels(rSc)
 .measure(measure)
 .instruments(rh_instruments)
 .create();    // luo uuden visualisoinnin
```

Tämän koodin ajon jälkeen rytmin nuotinnuksen visuasointi ja käyttöliittymä tulee näkyviin (kuva 9). Yllä olevassa koodissa käytettiin julkisen rajapinnan kaikkia muita metodeja paitsi isDirty().

Create –metodi toteuttaa varsinaisen visualisoinnin neljän yksityisen metodin avulla: ***updateGrid, createNotePalette, createSave ja updateNotes***. Metodien nimet kertovat mitä ne tekevät.

updateGrid luo kaksiulotteisen taulukon jossa sarakkeiden määrä saadaan rytmin kuvion pituudesta. Rivien määrä on äänilähteiden yhteislukumäärällä plus kaksi. Tämä

siksi että samaan taulukkoon asetetaan myös rivi nuottipalettille, sekä yksi tyhjä rivi nuottien ja nuottipaletin väliin. Nuotinnus-taulukon leveys ja korkeus saadaan kun svg:n vastaavista vähennetään annetut marginaalit. Marginaaleja tarvitaan vasemmalla instrumenttien nimille ja ylhäällä rytmin nimelle. Sovelluksessa marginaalit on toistaiseksi kovakoodattu.

```
var margin = {top: 110, right: 40, bottom: 30, left: 140};
innerWidth = width - margin.left - margin.right;
innerHeight = height - margin.top - margin.bottom;
```

Data-alkioiden sijainti näytöllä lasketaan funktioilla jotka osaavat asettaa datan sen 'x' ja 'y' –ominaisuuksien mukaan svg:n nuotinnusalueelle:

```
xScale = d3.scale.linear()
    .domain([0, patternLength ])
    .range([margin.left, margin.left + innerWidth]);
yScale = d3.scale.linear()
    .domain([0, soundChannels + 2 ])
    .range([innerHeight + margin.top , margin.top]) // kasvaa alhaalta ylös kuten domain
```

`d3.scale.linear` –funktiolle annetaan sekä minimi että maksimi kahdesta eri arvojoukosta (domain, range). Kutsuttaessa funktiota annetaan sille argumenttina jokin arvo domain arvojoukosta (0,1,…patternLength) , jolloin funktio interpoloi lineaarisesti tulokseksi vastaavan arvon rangelle annetun minimin ja maksimin välillä, eli sen sijainnin svg –taulukossa.

`updateGrid` –funktiossa lasketaan myös yksittäisen nuotin esittämiseen käytettävän solun leveys ja korkeus. Laskutoimenpiteet ovat suoraviivaisia.

Funktio `createNotePalette` luo nimensä mukaisesti nuottipaletin. Apuna tässä käytetään `SoundCell` –funktio-objektia, joka pitää sisällään sovelluksen tukemat lyönnit ja niiden nuottien piirto-ohjeet. Tätä tiedostoa voi pitää metatietona sovelluksen tukemille instrumenttityypeille, instrumenttityyppien äänilähteille, ja edelleen ääniläh-

teillä tuotettaville lyönneille. Optimaalisessa tilanteessa – ja lopullisessa versiossa tämä on tarkoitus - nämä kaikki tiedot haettaisiin tietokannasta.

Kaksiulotteisen taulukon ulommalle tasolla on intstrumenttityypit (’djembe’ ja ’dundun’). Sen sisältämllä objekteilla on kentät instrumenttityypin nimelle ja sen äänilähteiden määrälle. Lisäksi sillä on taulukko –kenttä ***values***, joka tallentaa lyönnin yksilöimisen mahdollistavat äänilähde- ja name(lyönnin nimi) –kentät, sekä lyönnin piirto-ohjeet sisältävän `drawNote` –funktion, jonka parametri vastaanottaa yhden nuotin data-alkion.
Kaksiuloitteisen `instrumentTypes` –taulukkomuuttujan (Liite 2) rakenne:

- **djembe**
- "name":"djembe"
- "soundChannels":2
- ”values”: // taulukko objekteja joilla on kentät:
  - ”soundSource”
  - ”name”
  - ”drawnote”:function(d){ }
- **dundun**
- "name":"dundun"
- "soundChannels":2
- ”values”: // taulukko objekteja joilla kentät:
  - ”soundSource”
  - ”name”
  - ”drawnote”:function(d){ }

Sovelluksessa ’djembe’ –instrumenttityyppiin kuuluvat ’call’, ’djembe1’ ja ’djembe2’. Dundun –tyyppiin kuuluvat instrumentit ’kenkeni’, ’sangban’ ja ’dounoumba’. Taulukon 4 äänilähde-isku -pareja vastaavat arvoparit löytyvät identtisesti instrumenttityyppien `values` –taulukon ***soundSource*** ja ***name*** –kentistä. Sovelluksen tukemat lyönnit on tallennettu sinne.

`SoundCell` alustetaan nuotinnus-objektin tiedoilla ***noteWidth, noteHeight, xScale, yScale*** ja käytettyjen instrumenttien tiedolla ***instrsData***. Objektissa on kaksi julkista metodia: ***getSoundCell (soundSource, sound)***, joka palauttaa

tiettyä lyöntiä vastaavan objektin parametrina saatujen äänilähteen ja äänen perusteella, sekä ***getSoundCells,*** joka palauttaa edelläkuvatun lyönnit instrumenttityypeittäin jaottelevan kaksiulotteisen taulukon.

Funktiossa ***createNotePalette*** iteroidaan ***SoundCell.getSoundCell*** – funktion palauttama kaksiulotteinen taulukko sisäkkäisillä silmukoilla. Sisemmässä silmukassa iteroidaan instrumenttityypin **values** –taulukkoa. Kaikkien instrumenttityyppien kaikkille lyönneille lisätään sen identifioiva data-alkio samaan json-objektiin johon aikaisemmin lisättiin nuotteja vastaavat data- alkiot. Lyönnin yksilöimiseen ja sen tapahtumankäsittelyn mahdollistavat tarvittavat kentät löytyvät taulukosta 6. Sisemmässä silmukassa iteroidaan siis objekteja jonka kentät ovat: ***soundSource***, ***name*** ja ***drawNote***.

Taulukko 6. Nuottipaletin yhden nuotin data-alkion kentät.

avain	Arvon laskeminen sisäkkäisten silmukoiden sisällä	Tuettava ominaisuus / tapahtumankäsittely visualisoinnissa	Esimerkkejä arvosta
x	Kasvatetaan arvoa joka lyönnin jälkeen.	Horisontaalinen sijainti taulukossa.	0,1,2,..lyöntien määrä
y	Aina sama. taulukon ylin rivi.	Vertikaalinen sijainti taulukossa.	=äänilähteiden yhteislukumäärä + 2
soundSource	Iteroitavan data-alkion soundSource	Äänilähteen identifiointi	'djembe', 'dundun', 'bell'
sound	Iteroitavan data-alkion 'name' -kenttä	Tuotettava ääni	'open','slap','bass'
inst	Ulomman silmukan 'name' -kenttä	Nimi paletin yläpuolelle. Instrumentin äänet alle.	'djembe', 'dundun'
selection		Merkkaus –kenttä. Olemassaolo identifioi nuottipalettiin kuuluvaksi.	
id	patternLen * maxYPosition + xPosition	Data-alkion yksilöinti kaksiulotteisessa taulukossa sen sijainnin mukaan.	maxYPosition * PatternLength + x

Nuottipaletin taustalla olevan data-alkion tiedot eroavat rytmin nuottien tietojen osalta ainoastaan yhden kentän osalta: paletin tiedoilla on kenttä ***selection***, kun taas nuoteilla on instrumentin yksilöivä kenttä ***instSpe.*** Kaikki muut viisi kenttää alkioilla ovat samat, joten sen jälkeen kun sekä nuottipaletin että nuottien data-alkiot on lisätty samaan JSON-objektiin, voidaan niille yhteisten ominaisuuksien perusteella luoda niiden visuaalinen representaatio sekä antaa niille paikka luodussa koordinaatistossa niiden 'x'- ja 'y' ominaisuuksien avulla. Toisaalta niiden tapahtumankäsittely voidaan erottaa niiden yhden kentän eroavaisuuden perusteella.

4.2.2 Visuaalisuuden luonti ja tapahtumankäsittely

Luoduille sekä nuottipaletin että yksittäisten nuottien data-alkioille annetaan visuaalinen ilme funktiossa `updateNotes.` Funktio iteroi kaikki sovelluksen tukemat (kuusi kpl) äänilähde-arvo –parit, luo kullekin niistä vuorollaan d3 –valintaobjektin, jota se manipuloi SoundCell –objektissa määritellyllä tavalla.

Alkiot ryhmitellään d3.nest –funktiolla kahden kentän perusteella: ***soundSource*** ja ***sound***:

```
var dataByNotes = d3.nest()
    .key(function(el) {return el.soundSource})
    .key(function(el){return el.sound})
    .entries(notesData); // nuottipaletin ja yksittäisten nuottien data
```

Koodi jakaa lyönnit ryhmiin äänilähteen ja äänen perusteella (kts. Taulukko 4). Data-alkiot voidaan iteroida ensisijaisen erotteluperusteen, eli äänilähteen perusteella:

```
dataByNotes.forEach(function (element, index, array) {
```

Funktion parametri ***element*** pitää sisällään kulloinkin iteroitavan avaimen alle kuuluvat data-alkiot. Jaottelun perusteena ollut avain (äänilähde) on sen '***key***' –kentässä ja varsinaiset data-alkiot '***values***' –kentässä, joka vuorostaan iteroidaan sisemmässä kentässä:

```
element.values.forEach(function (el, in, ar) {
```

Nyt toissijaisen jaotteluperusteen, eli äänen, arvo on ensimmäisen parametrin (`el)` key -kentässä ja äänilähde-ääni -arvoparin perusteella suodatetut data-alkiot `el.values` –kentässä. `SoundCell` –funktio-objektista voidaan noutaa äänilähde/ääni –paria vastaavan objektin nuotin piirtofunktio:

```
var drawFunc= soundCell.getSoundCell(element.key,
el.key).drawNote;
```

Perustuen äänilähteeseen ja ääneen voidaan luoda d3:n valintaobjekti, jolla voidaan kontrolloda kullakin iteraatiokierroksella kentän `el.values` data-alkioita, eli rytmissä esiintyviä tietyntyyppisiä iskuja. Valintaobjekteja luodaan itse asiassa kaksi, valintojen avulla luodaan yhdellä koodipätkällä jokaista valintoihin kuuluvaa data-alkiota vastaava yksi <path>- ja yksi <rect> -elementti.

Valintojen dataksi annetaan kahdella iteraatiolla suodatettu datajoukko `el.values`. Ensimmäisen valinnan DOM –elementiksi lisätään <path> -jonka piirto-ohjeet sen 'd' –attribuuttiin saadan luodulla äänilähde-ääni –paria vastaavan SoundCell -objektin `drawNote(d)` funktiokutsulla. Aiemmin komponentin alustusmetodissa notaatiota varten luotuun ryhmittelyelementtiin(<g>) notG luodaan <path> elementti, jolle annetaan nuotin tyypin identifioiva css-tunnus yhdistämällä äänilähteen ja äänen nimi alaviivalla:

```
soundClass = element.key+"_"+el.key; // identifioija
var myPaths  =  notG.selectAll("path." + soundClass)
.data(el.values)  // data-items under soundSource-sound
// give dom for data given previously
myPaths.enter()
.append("path")
.attr("class", soundClass)
.attr("d", drawFunc)  // nuottia vastaava piirtofunktio
.attr("stroke-width", pathWidth)
```

```
//remove DOMs that have no specified data
 myPaths.exit()
.remove();
```

Tarkastellaan <Path> -elementin piirto-ohjeet sisätävää ***drawNote*** –funktiota, jota valinnan jokaiselle data-alkiolle automaattisesti kutsutaan. Otetaan esimerkiksi djembe-äänilähteen 'slap' lyönnin piirtofunktio:

```
drawNote: function(d){
polku = "M" + parseInt(xSca(d.x) - (noteWidth  / 2 - horNoteMar)) + " " +    parseInt(ySca(d.y)  + pathWidth)+ " L" + parseInt(xSca(d.x) + (noteWidth  / 2 - horNoteMar) ) + " " + parseInt(ySca(d.y) + pathWidth)
polku = polku + " M" + parseInt(xSca(d.x)) + " " + parseInt(ySca(d.y)+ pathWidth) + "v" + (noteHeight - verNoteMar)
return polku;
```

SoundCell –funktiolle oli jo sen alustuksessa annettu yksittäisen solun leveys, korkeus sekä sen svg- alustan sijainnin kykenemään laskevat skaalausfunktiot (xSca, ySca). Lisäksi alustuksen yhteydessä lasketaan solun sisälle pienet (esim. 10% koosta) marginaalit (horNoteMar ja verNoteMar) sekä vaaka- että pystytasossa, jotteivat vierekkäisten solujen nuottien piirrokset tule toisiinsa kiinni. Solun piirtoalueen leveys ja korkeus on siis 90% solun leveydestä. Funktio palauttaa piirto-oheet 'T' –kirjaimen mallisen kuvion piirtämiseksi: ensin vaakasuora viiva solun yläosaan, sitten toinen pystysuora viiva solun piirtoalueen ylhäältä alas keskelle sijoittuen. Parametrina funktio saa nuotin data-alkion tiedot, jonka perusteella skaalausfunktiot osaavat laskea sen sijainnin. Esimerkiksi koodi ySca(d.y) + verNoteMar laskee data-alkiota vastaavan solun piirtoalueen yläreunan svg –alustalla. Koodi ySca(d.y) + noteHeight - verNoteMar puolestaan laskee solun piirtoalueen alalaidan. Tälläisiä komentoja yhdistämällä svg:n virtuaalikynän ohjeisiin ('M', 'L', 'A', 'V', ym.) on yksinkertaisten svg –kuvioiden piirtäminen skaalautuvalla tavalla suhteellisen nopeaa. Jos Svg –elementin kokoa muutetaan välittyy muunnos scatter –komponentin skaalausfunktioihin ja edelleen SoundCell –objektiin (xSca, ySca).

Samalla periaatteella datajoukoille luodaan toinen valinta joka luo data-alkioille suorakulmaisen <rect> -elementin. Rect elementtien luonti on periaatteeltaan samanlainen kuin edellä kuvattu <path> -elementin luominen.

Kuva 11. Solun piirtoalueen sisällä djemben 'slap' lyönnin nuotti. Taustalla suorakaide <rect> ja sen sisällä 'T' –kirjaimen mallinen <path>.

Elementin <rect> tarkoitus on tunnistaa käyttäjän hiiritapahtumat nuotin piirtoalueen kohdalla. Tunnistukset tehdään erikseen nuottipaletin ja rytmin nuoteille. Nuottipaletin tapahtumat tunnistetaan sen data-alkioille annetusta 'selection' –kentästä käyttäen d3:n valinta-objektin suodatinfunktiota filter:

```
soundRects
 .filter(function(d) { return d.selection != null })
 .on("click",function(d){
 selSound = d;  // tallennetaan nuottipaletin valittu nuotti
 d3.select(this).style("stroke", "black")
.classed("selected", true) // valitun nuotin tunniste-css
```

Tapahtumankäsittelijässä siis asetetaan muuttujaan valitun nuottipaletin nuotin data, ja muutetaan sen visuaalista ilmettä css- tyylitiedostoon pohjautuen, vaihtamalla sen DOM –elementin <rect> css –luokkaa.

Vastaavalla tavoin varsinaisen nuotinnuksen nuottien tapahtumat tunnistetaan niiden data-alkioiden '***instSpe***' –kentästä, jolla suodatettujen elementtien tapahtumankäsittely suoritetaan funktiossa ***handleNoteClick(d)***. Klikattu solu kuuluu tietyn instrumentin tiettyyn äänilähteeseen, joten ensin tutkitaan kuuluuko myös nuottipaletin

valittu nuotti samaan äänilähteeseen. Instrumentille ei voi saada antaa toisen instrumentin, tai saman instrumentin toisen äänilähteenkään, lyöntien nuotteja! Tyhjä nuotti voidaan kuitenkin antaa aina. Jos ehto toteutuu vaihdetaan klikatun solun taustalla olevan data-alkion 'sound' –kentän arvo nuottipaletin valitun nuotin vastaavaksi ja päivitetään nuotinnus.

```
function handleNoteClick(d){
if((!!selSound&&selSound.soundSource==d.soundSource)||
selSound.sound=="emp"){
```

…. Etsii klikatun elementin data-alkion 'id' -kentän arvon perusteella notesData –JSON-taulukosta, poistaa elementin ja lisää notesDataan uuden data-alkion jolla on 'sound' –kenttää lukuun ottamatta samat arvot kuin vanhalla alkiolla. 'sound' –kentän arvoksi asetetaan nuottipaletista valitun nuotin 'sound' –kentän arvo…..(kts koodi scatter.js, funktio `handleNoteClick`.

```
updateNotes();  // päivittää nuottien visuaalisuuden
dirty = true; // rytmin nuotinnusta on muutettu.
```

Kun datataulukkosta on poistettu vanha ja lisätty uusi data-alkio, kutstutaan nuottien päivittämiseksi funktiota `updateNotes.` Funktion ajon aikana sovelletaan siinä `handleNoteClick` –funktiossa käsiteltyihin alkioihin valinnan 'Enter – update – exit' –mallia. Uusi data-alkio on lisätty nuottien taustalla olevaan dataan. `updateNotes` –funktiossa sillä on data, muttei sitä vastaavaa DOM –elementtiä joten se päätyy valinnan enter- osioon ja sille luodaan DOM -elementit. Vanhan nuotin data-alkio puolestaan poistettiin datajoukosta – sillä on kuitenkin valinnassa edelleen DOM –elementit (<rect> ja <path>). Täten se päätyy exit -osioon ja koko alkio poistetaan valinnasta.

Koska valintojen soundRects ja soundPaths taustalla on sama datajoukko (notesData) voidaan niiden avulla luoda sovelluksen tilaa kuvaavia visuaalisia vihjeitä käyttäjälle. Tietyn solun sisällä olevaa <Rect> -elementtiä vastaava <Path> löydetään solun yksilöivän 'id' –kentän avulla. Käyttäjälle voidaan esimerkiksi visuaalisesti kertoa hiiren olevan tietyn nuotin kohdalla muuntamalla sen <rect> ja <path> -elementtien visuaalista ilmettä 'mouseover' –tapahtumassa.

```
soundRects
.on("mouseover",function(d){
d3.select(this).style("opacity", 0.42)  // this viittaa yksittäiseen DOM -elementtiin
myPaths.filter(function(d2){return d.id==d2.id})
    .attr("stroke-width", pathWidth+2)  // lihavoidaan Path-viivaa
})
.on("mouseout", function(d){ …..arvot oletuksiinsa…}
```

Yllä olevassa koodissa 'this' viittaa valinnan siihen DOM –elementtiin johon tapahtuma mouseover kohdistettiin. Piirtoalueen väriä vahvennetaan ja Nuotin <rect> -elementtiä vastaavan <path> -elementin viivaa paksunnetaan, kunnes 'mouseout' –tapahtumassa ominaisuudet muutetaan alkuperäisiksi.

Samalla tavoin nuottipaletin tilan ja nuotinnuksen välille voitaisiin helposti luoda visuaalisia vihjeitä koska niiden kentät ovat oleellisilta osin samat, esim. nuotinnuksesta voitaisiin visuaalisesti korostaa rivit, joilla on sama äänilähde kuin valitulla nuotilla.

4.2.3 Direktiiviputkea pitkin takaisin

Hyvin luodussa sovellusarkkitehtuurissa eri komponentit ovat löyhästi sidottuja(loose coupling) toisiinsa (Booktype, 2013). Hyväksikoeteltua paradigmaa noudattaen jaotellaan paitsi visualisointikomponentin metodit, myös sen tapahtumat julkisiksi ja yksityisiksi. Sovelluksessa ainoat tapahtumat joiden suhteen visualisointikomponentin ulkopulisen maailman on kyettävä sen kanssa vuorovaikutukseen ovat: rytmin vaihtaminen, uuden luonti ja muutetun rytmin tallentaminen. Näistä vain yksi lähtee komponentista asiakaskoodille – rytmin tallentaminen – ja siksi komponentin luojan, direktiiviin `djeRhythm,` on kyettävä kuuntelemaan rytmin tallennustapahtumaa.

Uudellenkäytettävästä visualisointikomponentista välitetään tapahtumia asiakkaalle dispatch –menetelmällä (Booktype, 2013), jonka käyttöönotto sovelluksessa vaatii vain neljä koodiriviä. Luodaan dispatch objekti muuttujien määrittelyosiossa, ensimmäisenä

parametrinä annetaan asiakkaalle sulkeumasta palautettava julkisen rajapinnan sisältävä objekti, toiselle parametrille annetaan tapahtumalle jokin nimi:

```
1) var dispatch = d3.dispatch(djeNotation, "saving");
```

Jo Jotta asiakas kykenee kuuntelemaan tapahtumaa, on sille julkisen API:n lisäksi tiedotettava kustomoidusta tapahtumasta. Muutetaan tapaa jolla sulkeuma palauttaa julkisen API:n – lisätään siihen myös julkinen tapahtuma jota asiakas voi kuunnella:

```
//return djeNotation; // mere public API not enough

2) return d3.rebind(djeNotation, dispatch, "on");
```

Asiakaspuolella asiakas rekisteröidään tapahtuman kuuntelijaksi. Tapahtuman lauetessa ajetaan funktio, joa saa parametrikseen (muutetun) nuotinnuksen datan:

```
3) sPlot.on("saving", function(rhythm_data) {

// TODO: saatu nuotinnusdata pitäisi konvertoida samanlaiseen struktuurin kuin millä se alun perin tietovarastosta haettiin: poistaa 'x' & 'y' –kentät, ym.

$scope.rhythm = rhythm_data; // two-way binding→ näkyy app.js:lle
```

Laukaistaan kustomoitu tapahtuma. Luonnollisesti 'Save' –painikkeen tapahtumankäsittelijässä:

```
4) dispatch.saving(notesData); // muutettu data asiakkaalle.
```

Direktiivin tapahtuman laukaisevassa funktiossa parametrina saatu nuotinnus-data tulisi muuntaa samanrakenteiseen muotoon kun millaisena se alun alkaen haettiinkiin. Siitä olisi mm. karsittava sen ainoastaan visualisointikomponentille tarkoitetut sijaintikoordinaatit 'x' ja 'y'. Lisäksi funktiossa tehdään kaksi asiaa. Direktiivin DDO –kentässä scope direktiiville annettiin suora viittaus app.js: -tiedoston muuttujaan, tätä havainnollistettiin direktiiviputkella kuvassa 9. Asetetaan tallennusmuotoon muokattu rytmin datatiedosto muuttujaan joka direktiiviputken kautta näkyy ulkomaailmaan Angular –sovelluksen pääkontrollerille `'MainCtrl'`, ja kutsutaan niin'ikään direktiiviputkella asetettua `MainCtrl`:n tallennus –funktiota.

```
$scope.rhythm = rhythm_data; // heijastuu MainCtrl:ään
```

```
$scope.saverhythm();  // direktiiviputkesta välitetty Main.ctrl:n metodi
```

5 Pohdinta ja jatkokehitys

Opinnäytetyön teko koki melkoisia muutoksia matkan varrella, alkuperäisenä suunnitelmana oli toteuttaa ohjelma kokonaan tai suurimmalta osin angularin itsetehdyillä direktiiveillä ja käyttää D3:a apuna ehkä vain pieniltä osin. Tehtävä osoittautui vaikeaksi, ja D3:a samalla opiskellessa iski mieleen ajatus tavasta, jolla visualisoinnin voisi sillä toteuttaa, ja niin työn painopiste kallistui enemmän D3:n puolelle. Koin ratkaisun toteutuksen kannalta järkeväksi, vaikka iso osa tehtyä työtä valuikin hukkaan. D3 sopii taulukkomaiseen esitykseen oivallisesti. Sovellusta on helppoa jatkaa eteenpäin: esimerkiksi instrumenttien (’djembe1 ja 2’, ’kenkeni’, ym) poistonappula on varmasti helpohko toteuttaa antamalla nuotinnuksen taustalla olevalle taulukolle kaksi saraketta lisää ja laskemalla <rect> -alkioille oikea korkeus järjestyksessä olevien instrumenttien äänilähteitä laskemalla, antamalla instrumenttia vastaavan data-alkion jollekin kentälle instrumentin nimi, ja edelleen poistaen nuotinnoksesta solut jolla on vastaava nimi jo valmiina `'instSpe'` -kentässä.

Rytmin eri instrumenttien erottaminen toisistaan tulisi erottaa eri visuaalista kanavaa pitkin kuin tahtien erottaminen (Cairo 2013, 114). Nyt molemmat erotetaan värin voimakkuuden muutoksilla, mikä hämmentää aivoja visuaalisia vihjeitä tulkitessa. Instrumentit voisi erottaa toisistaan pienellä välillä.

Angularilla luotiin sovelluksessa yksi itseluotu direktiivi joka näyttää attribuuttina saadun rytmin tiedot. Koska direktiivit elävät itsenäisesti ja omassa $scopessaan (Körner 2015, 102), niillä on mahdollista luoda kaksi rytmiä luomalla HTML –sivulle kaksi direktiivi-instanssia ja antamalla niille eri rytmin tiedot.

<dje-rhythm rhythm="rhythm1"…….> // controlleriin $scope.rhythm1

<dje-rhythm rhythm="rhythm2"…….> // controlleriin $scope.rhythm2

Tällä tavoin olisi helppoa vertailla kahden eri rytmin tietoja samanaikaisesti. Samoin direktiivi on upotettavissa osaksi Angularilla toteutettavaan suurempaa sovellusta. Esimerkiksi datan hallinnan ja ja mahdollisen tietokantakommunikoinnin tekoon Angular

taipunee oivallisesti. Eräs mahdollinen tietokantaratkaisu voisi olla pilvessä toimiva JSON –formaattia hyvin tukeva firebase. Direktiivi –elementti taipuu myös sellaisen web-kehittäjän käyttöön joka ei ohjelmoi.

Tutkimuskysymysten kannalta kimuranteimmaksi osoittautui modulaarisen rakenteen kehittäminen. Puutteellinen osaaminen JavaScriptissä hidasti työtä vielä entisestään. Tällä hetkellä tietovarastoon kuuluvat tiedot ovat erikoisesti levällään: rytmin json –tiedostossa ja lyöntien tiedot SoundCell –tiedostossa. Lyöntien tiedot olisi toki haettava samasta paikasta (ja sitten direktiiviputkeapitkin ja edelleen SoundSourcelle). Ongelmana tässä on SVG –komponentit, miten ne tallentaa tietovarastoon siten että tulisivat skaalautuviksi visuaalisointikomponenteille? Vastaus lienee attribuutissa 'viewport': jos jokaiselle solulle antaa oman svg-elementtinsä, voinee ne skaalata koko rytmin pohjalla olevan svg:n kokoon suhteutuvasti?

Nuotinnuksen esittäminen intuitiivisesti on rytmin näin yksinkertaisten osien kannalta lähes triviaalia. Asia muuttuu kun kuvioon lisätään monipuolisempia rytmiikan elementtejä, joita tässä työssä ei käsitelty ollenkaan.

Jatkokehityksen kohteista ei todellakaan ole pulaa, ei näinkään triviaalin nuotinnuksen esittämisessä, saatikka sitten laajemmassa.

Projektinhallinta oli lyhyesti kuitattuna kimpoilevaa, mutta tänne saakka päästiin.

Lähteet

Angular API, $compile. 2016. Angularin jatkuvasti päivittyvä verkkoresurssi. Luettavissa: https://docs.angularjs.org/api/ng/service/$compile. Luettu: 17.3.2016.

Angular API, ng-repeat. 2016. Angularin jatkuvasti päivittyvä verkkoresurssi. Luettavissa: https://docs.angularjs.org/api/ng/directive/ngRepeat. Luettu: 17.3.2016.

Angular API, $rootScope. Angularin jatkuvasti päivittyvä verkkoresurssi. Luettavissa: https://docs.angularjs.org/api/ng/service/$rootScope. Luettu: 15.3.2016.

AngularJS. 2014. AngularJS & D3: Directives for visualizations. Opetusvideo. Katsottavissa: https://www.youtube.com/watch?v=aqHBLS_6gF8. Katsottu 14.3.2016

Booktype 2013. The Reusabable API. Dispatching Events. Verkkoresurssi. Luettavissa: http://backstopmedia.booktype.pro/developing-a-d3js-edge-sample-chapter/chapter-3/. Luettu 27.11.2016.

Bostock, Mike. 2012. Toward reusable charts. Verkkoresurssi. Luettavissa: https://bost.ocks.org/mike/chart/ .Luettu 08.05.2016.

Cairo, Alberto. 2013. The functional art: an introduction to information graphics and visualization. New Riders. Berkeley, CA

Cockford, Douglas. 2015. Cockford on JavaScript – Functions. Verkkoresurssi. Katsottavissa: https://www.youtube.com/watch?v=lVnnxfdLdlM. Katsottu 19.11.2016

Dag-Inge. 2014. Understanding template compiling in AngularJS. Artikkeli. Luettavissa: http://daginge.com/technology/2014/03/04/understanding-template-compiling-in-angularjs/. Luettu 19.4.2016

Developer guide, bootstrap. 2016. Angularin dokumentaatio. Verkkoresurssi. Luettavissa: https://docs.angularjs.org/guide/bootstrap. Luettu 35.4.2016.

Developer guide, conceptual overview. Angularin jatkuvasti päivittyvä verkkoresurssi. Luettavissa: https://docs.angularjs.org/guide/concepts. Luettu 25.3.2016

Developer guide, dependency injection. Angularin jatkuvasti päivittyvä verkkoresurssi. Luettavissa: https://docs.angularjs.org/guide/di. Luettu: 1.4.2016

Developer guide, directives. Angularin jatkuvasti päivittyvä verkkoresurssi. Luettavissa: https://docs.angularjs.org/guide/directives. Luettu: 26.3.2016

Developer guide, expressions. Angularin jatkuvasti päivittyvä verkkoresurssi. Luettavissa: https://docs.angularjs.org/guide/expressions. Luettu: 26.4.2016.

Developer guide, $scope. Angularin jatkuvasti päivittyvä verkkoresurssi. Luettavissa: https://docs.angularjs.org/guide/scope. Verkkoresurssi. Luettu: 2.4.2016.

D3 wiki, selections. 2016. Verkkoresurrsi. Luettavissa: https://github.com/d3/d3/wiki/Selections. Luettu 4.3.2016

Hernly, Patrick. 2010. Effect of Knowledge in Music Notation Systems on College Music Majors' Transcription of West African Drumming Music. St. Petersburg College, U.S.A

Jenkov, Jakob, SVG. 201X. Svg Path Element. Verkkoresurssi. Luettavissa: http://tutorials.Jenkov.com/svg/path-element.html. Luettu 21.11.2016.

Jenkov, Jakob. 2013. SVG Compressed. Jenkov Apps. Copenhagen.

Karpov, Valeri & Netto, Diego. 2015. Professional AngularJS. John Wiley & sons. Indianapolis.

Kurz, Josh. 2015. Mastering AngularJS Directives. .[PACKT] Publishing. Birningham.

Körner, Christoph 2015. Data Visualization with D3 and AngularJS. [PACKT] Publishing. Birningham.

Marx, Mitchell. 2015. How AngularJS works. Verkkoresurssi. Katsottavissa: https://www.youtube.com/watch?v=fzWtSdgwgzU. Katsottu 4.4.2016

Meeks, Elijah. 2015 . D3.js In Action. Manning Publications co. Shelter Island, NY.

ModusCreate. 2014. The magic of AngularJS two-way binding. Article. Luettavissa: http://moduscreate.com/the-magic-of-angular-js-two-way-binding/. Luettu: 11.4.2016

NewCircle Training. 2014. JavaScript Scope Chains and Closures. Verkkoresurssi. Katsottavissa: https://www.youtube.com/watch?v=zRZNb4GDOPI. Katsottu 19.11.2016.

O'Reilly, 2010. Svg essentials. Verkkoresurssi. Luettavissa: http://commons.oreilly.com/wiki/index.php/SVG_Essentials/Paths. Luettu 29.11.2016.

Original D3 paper. D3 Data-driven documents. Michael Bostock, Vadim Ogievetsky ja Jeffrey Heer. Verkkoresurssi. Luettavissa:

Polak, Rainer. 2009. The Jenbe Realbook, vol.2. 2008. Bibiafrica records.

Rabenda, Clayton. 2014. Introduction to Angular's MVC: the magic behind $scope.$watch, $parse, and $scope.$apply. Verkkovideo. Katsottavissa: https://www.youtube.com/watch?v=rDgFfBG2e4o. Katsottu: 9.4.2016

Ruebbelke, Lukas. 2015. Angular application development. Verkkoresurssi, katsottavissa: https://app.pluralsight.com/library/courses/angular-application-development/table-of-contents. Katsottu: 30.3.2016

Sotelo, Kaleb 2012. Verkkoresurssi. Evolution of the single page application. Luettavissa: http://paislee.io/evolution-of-the-single-page-application/. Luettu 25.3.2016

Suomijoki, Juha. 2015. AngularJS. Yksisivuisen web—sovelluksen käyttöliittymän toteutus AngularJS:llä. Metropolian Ammattikorkeakoulu. Opinnäytetyö. Luettu 29.3.2016

Wahlin Dan. Learn to build AngularJS Custom Directives with Dan Wahlin. 2015. Verkkovideo. Maksullinen opetusvideosarja. Katsottavissa: https://www.udemy.com/angularjs-custom-directives/learn/#/. Katsottu 16. - 19.3.2016

Wahlin, Dan Big Picture. 2013. Using an AngularJS Factory to interact with Restful service. Verkkoresurssi. Luettavissa: https://weblogs.asp.net/dwahlin/using-an-angularjs-factory-to-interact-with-a-restful-service. Luettu 28.11.2016.

Waikar, Manoj, 2015. Data-oriented Development with AngularJS.[PACKT] Publishing. Birningham.

Wikipedia, AngularJS. 2016. Verkkoresurssi. Luettavissa: https://en.wikipedia.org/wiki/AngularJS. Luettu: 29.3.2016.

Wikipedia, djembe. 2016. Verkkoresurssi. Luettavissa: https://fi.wikipedia.org/wiki/Djembe. Luettu 16.5.2016

Wikipedia, Doundoun. 2016. Verkkoresurssi. Luettavissa: https://nl.wikipedia.org/wiki/Doundoun. Luettu 16.5.2016

Wikipedia, JavaScript. 2016. Verkkoresurssi. Luettavissa: https://en.wikipedia.org/wiki/JavaScript. Luettu 19.11.2016

Wikipedia , Single-page application. Verkkoresurssi. Luettavissa: https://en.wikipedia.org/wiki/Single-page_application. Luettu: 29.3.2016.

W3schools SVG. Verkkoresurssi. Luettavissa:
http://www.w3schools.com/graphics/svg_intro.asp

Zhu, Nick Qi. 2013. Data Visualization with D3.js Cookbook. [PACKT] Publishing. Birningham.

5.1 Liite 1 rhythms.json rytmien notaatiot .json-muodossa

Kahden rytmin (soli ja djansa) nuotinnus tallennettuna JSON –muotoon:


```
[{
  "name":"soli",
   "pattern_length":12,
   "measure":3,
  "instruments":{
     "call":{
        "name":"call",
        "instrument": "djembe",
        "length":12,
         "repeat":1,
         "sound_channels":1,
        "notes":[
           { "djembe":"flam_slap" },
           { "djembe":"_empt"},
           {"djembe":"slap"},
            {"djembe":"slap"},
           {"djembe":"_empt"},
           { "djembe":"slap"},
           { "djembe":"slap"},
           {"djembe":"_empt" },
         {"djembe":"slap"},
         {"djembe":"slap"  },
       {"djembe":"_empt"},
       {"djembe":"_empt"}
       ]
     },
     "djembe1":
     {
        "name":"djembe1",
      "instrument":"djembe",
```

```
  "length":12,
  "repeat":1,
   "sound_channels":1,
  "notes":[
     {"djembe":"open"},
     { "djembe":"_empt" },
     { "djembe":"slap"},
      {
        "djembe":"open"
     },
     { "djembe":"_empt"},
     { "djembe":"dje_bass" },
       { "djembe":"open"},
     {"djembe":"_empt"},
      {  "djembe":"slap"},
     {"djembe":"open"},
     {"djembe":"_empt"},
       { "djembe":"dje_bass"}
   ]
},
 "djembe2":
{
    "name":"djembe2",
  "instrument":"djembe",
  "length":12,
  "repeat":1,
   "sound_channels":1,
  "notes":[
     { "djembe":"open"},
     {"djembe":"_empt"},
     { "djembe":"_empt"},
     {"djembe":"open"},
     {"djembe":"slap"},
      { "djembe":"slap"},
```

```
            { "djembe":"open"},
            { "djembe":"_empt"},
            { "djembe":"_empt"},
            { "djembe":"open"},
          { "djembe":"slap" },
          { "djembe":"slap"}
          ]
      },
"kenkeni":
       {
         "name":"kenkeni",
        "instrument":"dundun",
        "length":12,
        "repeat":1,
         "sound_channels":2,
        "notes":[
          {
            "bell":"open",
            "dundun":"bass"
          },
          {
            "bell":"_empt",
            "dundun":"_empt"
          },
          {
            "bell":"open",
            "dundun":"bass"
          },
          {
            "bell":"_empt",
            "dundun":"_empt"
          },
          {
            "bell":"open",
```

```
      "dundun":"_empt"
    },
    {
      "bell":"_empt",
      "dundun":"_empt"
    },
     {
      "bell":"open",
      "dundun":"bass"
    },
    {
      "bell":"_empt",
      "dundun":"_empt"
    },
    {
      "bell":"open",
      "dundun":"bass"
    },
    {
      "bell":"_empt",
      "dundun":"_empt"
    },
    {
      "bell":"open",
      "dundun":"_empt"
    },
    {
      "bell":"_empt",
      "dundun":"_empt"
    }
  ]
},
 "sangban":
{
```

```
 "name":"sangban",
"instrument":"dundun",
"length":12,
"repeat":1,
 "sound_channels":2,
"notes":[
  {
    "bell":"open",
    "dundun":"bass"
  },
  {
    "bell":"_empt",
    "dundun":"_empt"
  },
  {
    "bell":"open",
    "dundun":"_empt"
  },
  {
    "bell":"_empt",
    "dundun":"_empt"
  },
  {
    "bell":"open",
    "dundun":"closed"
  },
  {
    "bell":"_empt",
    "dundun":"_empt"
  },
  {
    "bell":"open",
    "dundun":"closed"
  },
  {
    "bell":"open",
    "dundun":"closed"
  },
```

```
      {
        "bell":"_empt",
        "dundun":"_empt"
      },
      {
        "bell":"open",
        "dundun":"_empt"
      },
      {
        "bell":"open",
        "dundun":"bass"
      },
      {
        "bell":"_empt",
        "dundun":"_empt"
      },
      {
        "bell":"open",
        "dundun":"_empt"
      }
    ]
  },
  "dounoumba":
  {
     "name":"dounoumba",
    "instrument":"dundun",
    "length":12,
    "repeat":1,
     "sound_channels":2,
    "notes":[
      {
        "bell":"open",
        "dundun":"bass"
      },
```

```
{
  "bell":"_empt",
  "dundun":"_empt"
},
{
  "bell":"open",
  "dundun":"bass"
},
{
  "bell":"_empt",
  "dundun":"_empt"
},
{
  "bell":"open",
  "dundun":"_empt"
},
{
  "bell":"_empt",
  "dundun":"_empt"
},
{
  "bell":"open",
  "dundun":"_empt"
},
{
  "bell":"_empt",
  "dundun":"_empt"
},
{
  "bell":"open",
  "dundun":"bass"
},
{
  "bell":"open",
```

```
            "dundun":"bass"
          },
          {
            "bell":"_empt",
            "dundun":"_empt"
          },
          {
            "bell":"open",
            "dundun":"bass"
          }
        ]
      }
    }
},
 {
  "name": "djansa",
  "pattern_length": 16,
  "measure": 4,
  "instruments": {
    "call": {
      "name": "call",
      "instrument": "djembe",
      "length": 16,
      "repeat": 1,
      "sound_channels": 1,
      "notes": [
        {"djembe": "dje_flam_slap"},
      {"djembe": "_empt"},
      {"djembe": "slap"},
      {"djembe": "slap"},
        {"djembe": "_empt"},
      { "djembe": "slap"},
        {"djembe": "_empt"},
        {"djembe": "slap"},
```

```
  { "djembe": "slap" },
    {"djembe": "_empt"},
  { "djembe": "slap" },
    {"djembe": "_empt" },
  { "djembe": "slap"},
    {"djembe": "_empt"},
  {"djembe": "_empt"},
    {"djembe": "_empt"}
  ]
},
"djembe1": {
  "name": "djembe1",
  "instrument": "djembe",
  "length": 16,
  "repeat": 1,
  "sound_channels": 1,
  "notes": [
    {"djembe": "open"},
  {"djembe": "_empt"},
  {"djembe": "_empt"},
  {"djembe": "open"},
  { "djembe": "open" },
  { "djembe": "_empt"},
  { "djembe": "slap"},

  { "djembe": "slap"},
  { "djembe": "open"},
    {"djembe": "_empt"},
  {"djembe": "_empt"},
    {"djembe": "open"},
  {"djembe": "open"},
    {"djembe": "_empt"},
  {"djembe": "slap"},
  {"djembe": "slap},
```

```
        }
    ]
    },
    "djembe2": {
      "name": "djembe2",
      "instrument": "djembe",
      "length": 16,
      "repeat": 1,
      "sound_channels": 1,
      "notes": [
        {"djembe": "bass"},
        {"djembe": "_empt"},
      { "djembe": "slap"},
      {"djembe": "slap"},
      {"djembe": "_empt"},
        {"djembe": "_empt"},
      { "djembe": "open"},
      { "djembe": "_empt" },
        {"djembe": "bass"},
        {"djembe": "_empt"},
      { "djembe": "slap"},
        {"djembe": "slap"},
      {"djembe": "_empt"},
        {"djembe": "_empt" },
      { "djembe": "open" },
        {"djembe": "_empt"}
    ]
    },
    "kenkeni": {
      "name": "kenkeni",
      "instrument": "dundun",
      "length": 16,
      "repeat": 1,
      "sound_channels": 2,
```

```
"notes": [{
  "bell": "open",
  "dundun": "bass"
}, {
  "bell": "_empt",
  "dundun": "_empt"
}, {
  "bell": "open",
  "dundun": "_empt"
}, {
  "bell": "open",
  "dundun": "closed"
}, {
  "bell": "_empt",
  "dundun": "_empt"
}, {
  "bell": "open",
  "dundun": "bass"
}, {
  "bell": "open",
  "dundun": "bass"
}, {
  "bell": "_empt",
  "dundun": "_empt"
}, {
  "bell": "open",
  "dundun": "bass"
}, {
  "bell": "_empt",
  "dundun": "_empt"
}, {
  "bell": "open",
  "dundun": "_empt"
}, {
```

```
      "bell": "open",
      "dundun": "closed"
    }, {
      "bell": "_empt",
      "dundun": "_empt"
    }, {
      "bell": "open",
      "dundun": "bass"
    }, {
      "bell": "open",
      "dundun": "bass"
    }, {
      "bell": "_empt",
      "dundun": "_empt"
    }]
  },
  "sangban": {
    "name": "sangban",
    "instrument": "dundun",
    "length": 16,
    "repeat": 1,
    "sound_channels": 2,
    "notes": [{
      "bell": "open",
      "dundun": "bass"
    }, {
      "bell": "_empt",
      "dundun": "_empt"
    }, {
      "bell": "_empt",
      "dundun": "_empt"
    }, {
      "bell": "open",
      "dundun": "bass"
```

```
}, {
  "bell": "_empt",
  "dundun": "_empt"
}, {
  "bell": "_empt",
  "dundun": "_empt"
}, {
  "bell": "open",
  "dundun": "bass"
}, {
  "bell": "_empt",
  "dundun": "_empt"
}, {
  "bell": "_empt",
  "dundun": "_empt"
}, {
  "bell": "_empt",
  "dundun": "_empt"
}, {
  "bell": "open",
  "dundun": "closed"
}, {
  "bell": "_empt",
  "dundun": "_empt"
}, {
  "bell": "_empt",
  "dundun": "_empt"
}, {
  "bell": "_empt",
  "dundun": "_empt"
}, {
  "bell": "open",
  "dundun": "bass"
}, {
```

```
      "bell": "_empt",
      "dundun": "_empt"
    }]
  },
  "dounoumba": {
    "name": "dounoumba",
    "instrument": "dundun",
    "length": 16,
    "repeat": 1,
    "sound_channels": 2,
    "notes": [
     {
      "bell": "open",
      "dundun": "closed"
    }, {
      "bell": "_empt",
      "dundun": "_empt"
    }, {
      "bell": "open",
      "dundun": "bass"
    }, {
      "bell": "open",
      "dundun": "bass"
    }, {
      "bell": "_empt",
      "dundun": "_empt"
    }, {
      "bell": "open",
      "dundun": "bass"
    }, {
      "bell": "_empt",
      "dundun": "_empt"
    }, {
      "bell": "open",
```

```
        "dundun": "bass"
      }, {
        "bell": "open",
        "dundun": "bass"
      }, {
        "bell": "_empt",
        "dundun": "_empt"
      }, {
        "bell": "open",
        "dundun": "bass"
      }, {
        "bell": "_empt",
        "dundun": "_empt"
      }, {
        "bell": "open",
        "dundun": "_empt"
      }, {
        "bell": "_empt",
        "dundun": "_empt"
      }, {
        "bell": "open",
        "dundun": "_empt"
      }, {
        "bell": "_empt",
        "dundun": "_empt"
      }]
    }
  }
}
]
```


5.2 Liite 2 Sovelluksen tukemat lyönnit (SoundCell.js)

// TODO sulkeumaksi & oikeaksi koodiksi johon voi välittää tietoja.

```
this.instrumentTypes = [{
        "name": "djembe",
        "soundChannels":1,
        "values":[{
        soundSource:"djembe",
        name:"slap",
        drawNote: function(d){
var polku = "M" + parseInt(SoundCell.xScale(d.x) - (noteWidth  / 2 -horNoteMar)) + " " +
parseInt(SoundCell.yScale(d.y)  + pathWidth)+ " L" + parseInt(SoundCell.xScale(d.x) + (no-
teWidth  / 2 -horNoteMar) ) + " " + parseInt(SoundCell.yScale(d.y) + pathWidth )
      //Vertical line
     polku = polku + " M" + parseInt(SoundCell.xScale(d.x)) + " " + par-
seInt(SoundCell.yScale(d.y)+ pathWidth) + "v" + (noteHeight - verNoteMar)
return polku
       }
    },
    {
       soundSource:"djembe",
       name:"open",
       drawNote: function(d){
     // return "M" + parseInt(xScale(d.x) - (noteWidth  / 2 -horNoteMar)) +
      polku = "M" + parseInt(SoundCell.xScale(d.x)+  horNoteMar/3) + " " + par-
seInt(SoundCell.yScale(d.y) + verNoteMar / 4) + " A" + (noteWidth  / 5) + ", " + note-
Height / 6 + " 0 1,0 " + parseInt(SoundCell.xScale(d.x) - horNoteMar / 2)  + ", " + par-
seInt(SoundCell.yScale(d.y) + (noteHeight - verNoteMar -3)  / 2)

     polku += " M" + parseInt(SoundCell.xScale(d.x) - horNoteMar / 2)  + ", " + par-
seInt(SoundCell.yScale(d.y) + (noteHeight - verNoteMar - 3)  / 2) + " A" + (noteWidth  / 6)
+ ", " + parseInt(noteHeight / 6) + " 0 0,1 " + parseInt(SoundCell.xScale(d.x) - 2 * horNo-
teMar) + ", " + parseInt(SoundCell.yScale(d.y) + noteHeight - verNoteMar -4)
return polku;
       }
    }
   ,
```

```
{
soundSource:"djembe",
 name:"bass",
  drawNote: function(d){
   //Symbol resembling letter 'B'
   polku = "M"+ parseInt(SoundCell.xScale(d.x) - (noteWidth  / 2 - (1.5 * horNoteMar)))
+ " " + SoundCell.yScale(d.y) +   verNoteMar +  "v" +         (noteHeight - verNoteMar)
   polku = polku + " M" + parseInt(SoundCell.xScale(d.x) - (noteWidth  / 2 -(1.5 * hor-
NoteMar))) + ", " + SoundCell.yScale(d.y) + verNoteMar + " A" + (noteWidth  / 2 + 3) + ",
" + noteHeight / 4 + " 0 0,1 " + parseInt(SoundCell.xScale(d.x) - (noteWidth  / 2 -(1.5 *
horNoteMar))) + ", " + parseInt(SoundCell.yScale(d.y) + (noteHeight - verNoteMar/ 2)  / 2)

   polku+= " M" + parseInt(SoundCell.xScale(d.x) - (noteWidth  / 2 -(1.5 * horNote-
Mar))) + ", " +  parseInt(SoundCell.yScale(d.y) + noteHeight   / 3 )  + " A" + (noteWidth  /
2 + horNoteMar/2) + ", " + noteHeight / 4 + " 0 0,1 " + parseInt(SoundCell.xScale(d.x) -
(noteWidth  / 2 -(1.5 * horNoteMar))) + ", " + parseInt(SoundCell.yScale(d.y) + noteHeight -
verNoteMar + 1)
return polku;
  }
}
,
{
  soundSource:"djembe",
  name:"flam_slap",
  drawNote: function(d){
     console.log("in drawDjeSlapHH , sound: " + d.sound + "x: " + d.x + ", y: " + d.y);
 var polku = "M" + parseInt(SoundCell.xScale(d.x) - (noteWidth  / 2 -horNoteMar)) + " "
+   parseInt(SoundCell.yScale(d.y)  + pathWidth)+ " L" + parseInt(SoundCell.xScale(d.x) +
(noteWidth  / 2 -horNoteMar) ) + " " + parseInt(SoundCell.yScale(d.y) + pathWidth )
 //Vertical line
 polku = polku + " M" + parseInt(SoundCell.xScale(d.x)) + " " + par-
seInt(SoundCell.yScale(d.y)+ pathWidth) + "v" + (noteHeight - verNoteMar)
 return polku
  }
```

```
}] }
,
 {
    "name":"dundun",
    "soundChannels":2,
    "values":  [{
    soundSource:"bell",
    name:"open",
   drawNote: function(d){
var polku = "M" + parseInt(SoundCell.xScale(d.x) - noteWidth / 2 + horNoteMar) + " " +
parseInt(SoundCell.yScale(d.y) + noteHeight - verNoteMar) + " L" + par-
seInt(SoundCell.xScale(d.x) + noteWidth / 2 - horNoteMar) + " " + par-
seInt(SoundCell.yScale(d.y) + (1.5 * verNoteMar))

  polku += " M" + parseInt(SoundCell.xScale(d.x) - noteWidth / 2 + horNoteMar) + " " +
parseInt(SoundCell.yScale(d.y) + (1.5 * verNoteMar)) +" L" + parseInt(SoundCell.xScale(d.x)
+ noteWidth / 2 - horNoteMar) + " " + parseInt(SoundCell.yScale(d.y) + noteHeight - ver-
NoteMar)
  return polku;
    }
 }
,
{
   soundSource:"dundun",
   name:"bass",
   drawNote: function(d){
  //Symbol resembling letter 'B'
polku = "M"+ parseInt(SoundCell.xScale(d.x) - (noteWidth  / 2 - (1.5 * horNoteMar))) + "
" + parseInt(SoundCell.yScale(d.y) +  verNoteMar) +  "v" +        (noteHeight - 2 * verNo-
teMar)
    polku = polku + " M" + parseInt(SoundCell.xScale(d.x) - (noteWidth  / 2 -(1.5 * hor-
NoteMar))) + ", " + parseInt(SoundCell.yScale(d.y) + verNoteMar) + " A" + (noteWidth  / 2
+ 3) + ", " + noteHeight / 4 + " 0 0,1 " + parseInt(SoundCell.xScale(d.x) - (noteWidth  / 2 -
```

```
(1.5 * horNoteMar))) + ", " + parseInt(SoundCell.yScale(d.y) + (noteHeight - verNoteMar/ 2)
/ 2)

        polku+= " M" + parseInt(SoundCell.xScale(d.x) - (noteWidth  / 2 -(1.5 * horNote-
Mar))) + ", " +  parseInt(SoundCell.yScale(d.y) + noteHeight   / 3 )  + " A" + (noteWidth  /
2 + horNoteMar/2) + ", " + noteHeight / 4 + " 0 0,1 " + parseInt(SoundCell.xScale(d.x) -
(noteWidth  / 2 -(1.5 * horNoteMar))) + ", " + parseInt(SoundCell.yScale(d.y) + noteHeight -
verNoteMar + 1)
return polku;
        }
    }
    ,
    {
        soundSource:"dundun",
        name:"closed",
        drawNote: function(d){
      var polku = "M" + SoundCell.xScale(d.x) + ", " + parseInt(SoundCell.yScale(d.y) + ver-
NoteMar) + " A" + (noteWidth -  (2 * horNoteMar)) / 2 + ", " + parseInt((noteHeight - 1.5 *
verNoteMar) / 2) + " 0 0, 1 " + SoundCell.xScale(d.x) + ", " + parseInt(SoundCell.yScale(d.y)
+ verNoteMar - 6)
        }
    }]
    }
    ]
```