Haaga-Helia
ammattikorkeakoulu Oy

Sovelluskehysvertailu – Play 2 ja Spring Boot

Katja Lusma

Opinnäytetyö
Tietojenkäsittelyn koulutusohjelma
2016

Haaga-Helia
ammattikorkeakoulu Oy

Tiivistelmä

Tekijä

Lusma Katja

Koulutusohjelma

Tietojenkäsittelyn koulutusohjelma

Opinnäytetyön nimi

Sovelluskehysvertailu – Play 2 ja Spring Boot

Sivu- ja liitesivumäärä

33 + 2

Opinnäytetyön tavoite oli tutustua Play- ja Spring Boot -sovelluskehyksiin ja vertailla näiden ominaisuuksia. Lisäksi tavoitteena oli syventää ymmärrystä sovelluskehysten toiminnasta etenkin modernin web -sovelluskehityksen näkökulmasta. Muut sovelluskehykset rajattiin työn ulkopuolelle.

Vertailu aloitettiin hankkimalla tietoa sovelluskehyksistä yleensä, perehtymällä mikropalveluihin sekä siihen, miten sovelluskehykset liittyvät mikropalveluihin. Teoriaa syvennettiin tutkimalla kummankin sovelluskehyksen erityispiirteitä ja toimintatapaa.

Jotta sovelluskehysten toiminnasta saatiin oikeanlainen käsitys, molempia sovelluskehyksiä hyväksikäyttäen toteutettiin pienet RESTful -sovellukset Java -ohjelmointikielellä.

Play- ja Spring Boot -sovelluskehykset olivat hyvin erilaisia ja niiden todettiin vastaavan erilaisten projektien tarpeisiin. Spring Bootin havaittiin olevan odotuksia epäitsenäisempi osa Spring Frameworkia, ja soveltuvan hyvin perinteisiin Java -projekteihin. Playn katsottiin soveltuvan parhaiten funktionaalisen ohjelmistokehityksen sovelluksiin, etenkin jos toteutuskielenä on Scala.

Asiasanat

sovelluskehys, Spring Boot, Play, mikropalvelu, RESTful

Sisällys

1 Lyhenteet ja termit

API	Lyhenne sanoista Application Programming interface, eli ohjelmointirajapinta, jonka avulla ohjelmat voivat keskustella keskenään.
Boilerplate	Boilerplate –koodilla tarkoitetaan koodikokonaisuuksia, jota toistetaan useassa kohdassa samanlaisena. (Wikipedia 2016b)
CRUD	Lyhenne sanoista Create, Read, Update, Delete, eli perustoiminnallisuus tietokannan kanssa toimimiseen.
HTML	Lyhenne sanoista Hyper Text Markup Language. Ohjelmointikieli verkkosivujen rakentamiseen.
HTTP	Hypertext Transfer Protocol, eli protokolla jota WWW-palvelimet ja selaimet käyttävät tiedostojen siirtoon. (Wikipedia 2016d)
IDE	Lyhenne sanoista Integrated Development Environment. Ohjeilmointiympäristö, joka toimii editorina ja ohjelmointikielen kääntäjänä.
ORM	Lyhenne sanoista Object-relational Mapping. Tekniikka olioiden ja tietokantataulujen välisen liikenteen automatisointiin.
MVC	Model View Controller (malli-näkymä-käsittelijä) –malli on ohjelmistoarkkitehtuurityyli jossa ohjelma jaetaan selkeästi kolmeen eri osaan. (Wikipedia 2016a).
REST	Representational State Transfer eli REST on HTTP-protokollaan pohjautuva arkkitehtuurimalli, jolla toteutetaan ohjelmointirajapintoja. (Wikipedia 2016c)
RESTful	REST -rajapintaa noudattavia verkkopalveluja kutsutaan RESTful -palveluiksi.

2 Johdanto

Sovelluskehys on eräänlainen runko rakennettavalle ohjelmistolle. Sen avulla usein määritetään projektin rakenne, tietyt ohjelmointikäytännöt ja käytettävät kirjastot. Sovelluskehys voi käyttää useampaa ohjelmointirajapintaa, joista se koostaa mielestään parhaat käytännöt yhdeksi helposti käytettäväksi paketiksi.

Sovelluskehysten tarkoitus on tarjota tiettyjä yksinkertaistettuja toimintatapoja yleisimpien toiminnallisuuksien suorittamiseen, jotta vältytään koodin toistamiselta, eli ns. boilerplate -koodilta. Kehysten tarjoamat valmiit kirjastot auttavat sovelluskehittäjää, jotta pyörää ei tarvitse keksiä uudestaan. Sovelluskehystä käyttämällä ohjelmistoprojektin rakenne pysyy selkeänä.

Ohjelmointityössä päätyy useimmiten käyttämään jotain sovelluskehystä, koska monesti ne tarjoavat yksinkertaistettuja ratkaisuja monimutkaisiin asioihin ja helpottavat ohjelmoijan työtä, mahdollistaen siten ohjelmiston nopeamman tuotantoon viemisen.

Useimmat sovelluskehykset ovat avoimen lähdekoodin ratkaisuja, ja hyviä sovelluskehyksiä on tarjolla jo sen verran, että niistä on varaa valita itselleen sopiva. Sovelluskehyksiä tulee myös uusia, ja monet keskittyvät joko tiettyyn ohjelmointikieleen tai tiettyyn tarkoitukseen. Suurin osa nykyään käytetyistä sovelluskehyksistä ovat ns. web -sovelluskehyksiä, eli pyrkivät tarjoamaan toiminnot verkkopalveluiden rakentamiseen. Sovelluskehykset sisältävät usein valmiit ratkaisut mm. tietokantakäsittelyyn, session hallintaan, autentikointiin, tietoturvallisuuteen ja HTML -sivujen luomiseen.

Sovelluskehyksiä on aina toisinaan kritisoitu siitä, että ne lisäävät sovelluskehyksen omaa monimutkaista koodia sovelluksessa, kuin myös siitä, että sovelluskehyksen käyttö vaatii ohjelmointikielen lisäksi sovelluskehyksen ja sille tyypillisten ohjelmointitapojen opettelun. Sovelluskehystä käytettäessä turha boilerplate -koodimäärä kuitenkin vähenee, ja sovelluskehykset voivat auttaa tekemään koodista entistä selkeämmin jäsenneltyä.

Java EE -sovellusalusta on ollut omalla tavallaan uranuurtaja nykyaikaisille sovelluskehyksille. Kuten useat sovelluskehykset, myös Spring Framework nojaa Java EE:n spesifikaatioihin ja määrityksiin.

Tässä opinnäytetyössä tutustutaan kahteen Java -ympäristön web -sovelluskehykseen ja vertaillaan niiden eroja ja samankaltaisuuksia. Opinnäytetyön tarkoituksena on samalla myös perehtyä sovelluskehysten toimintaan modernin websovelluskehityksen työkaluina.

Vertailussa on mukana kaksi kehystä, koska useamman mukaanottaminen olisi tehnyt työstä liian laajan, ja toisaalta vain yhtä tutkimalla ei saisi tarpeeksi laajaa kuvaa siitä miten sovelluskehykset hyödyttää kehitystyötä. Vertailuun otetaan mukaan Play- sekä Spring Boot -sovelluskehykset. Verrattuna aikaisempiin, massiivisempiin sovelluskehyksiin (Spring Framework, JavaEE) näiden kahden pitäisi tarjota kevyempi ja suoraviivaisempi tapa tehdä websovelluksia.

Spring Boot on vertailussa mukana, koska Spring on ollut yksi käytetyimmistä sovelluskehyksistä jo pitkään, ja Boot -laajennus tuo tähän uuden näkökulman. Play on mukana, koska se oli ensimmäisten sovelluskehysten joukossa joihin tutustuin. Muut sovelluskehykset sekä tässä rakennettavien sovellusten jatkokehitys rajataan tämän opinnäytetyön ulkopuolelle.

Vertailu toteutetaan tekemällä pienet websovellukset molempia kehyksiä hyödyntäen. Tietokannan ja käyttöliittymän on tarkoitus olla yhteinen, jotta voidaan keskittyä vertailemaan sovelluskehysten toimintaa niiden ydintoiminta-alueilla.

3 Sovelluskehykset mikropalveluissa

Kun aiemmin saatettiin keskustella siitä, mikä ohjelmointikieli oli paras tai mielenkiintoisin, nykyään vastaava keskustelu käydään usein sovelluskehysten välillä. Sovelluskehykset ovat monesti niitä joiden mukana tulevat ohjelmoinnin uudet ideat, käytännöt ja filosofiat. (Wayner 2015)

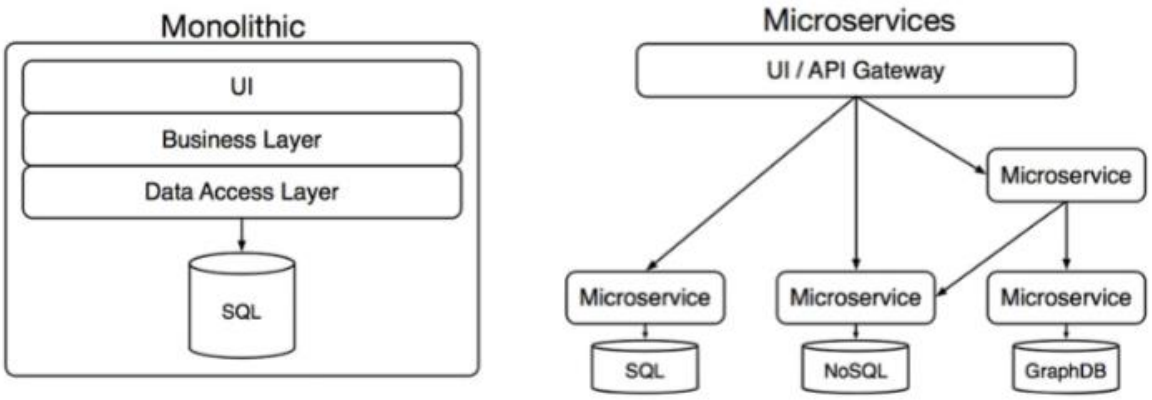
Ohjelmoinnissa oli aiemmin tärkeintä se, että ohjelmoija osasi käyttämänsä kielen kaikki nyanssit - pointterit, funktiot ja näkyvyysalueet - mahdollisimman tarkasti. Nykyään automaatio hoitaa ison osan näistä, ja siksi suuri osa nykyisestä ohjelmakoodista on peräkkäisiä API -kutsuja. Tämän takia on tärkeämpää ymmärtää rajapintoja, sovelluskehyksiä ja niiden toimintaa, kuin hallita tiettyä yhden ohjelmointikielen erityispiirrettä. Sovelluskehyksiä ja niiden rajapintoja käyttäessä ohjelmoija mahdollisesti välttää salakavalat virheet, koska yleisimpiä kehyksiä on testattu ja käytetty vuosia. (Wayner 2015)

Useat sovelluskehykset noudattavat MVC -mallia erottaakseen käyttöliittymän ja liiketoimintalogiikan tietomallista. MVC -mallin käyttöä pidetään hyvänä toimintamallina, koska siten koodi pysyy modulaarisena ja uudelleenkäytettävänä. (Wikipedia 2016g).

Ohjelmointikielet eivät itsessään rajoita, mitä kielellä voi tehdä. Joissain kielissä voi olla rajoituksia tietyille ilmaisuille jotta koodi pysyisi virheettömänä, mutta kielessä ei suoraan ole rajoitusta millekään toiminnolle. Sovelluskehykset puolestaan tuovat aina mukanaan omat rajoitukset ja säännöt. Jos kehys ei tue jotain asiaa, sitä ei olla toteutettu sen API:in. Sopivaa sovelluskehystä valitessa pitää ottaa huomioon, että sen mahdollisuuksien ja rajoitusten kanssa on voitava toteuttaa koko projekti. (Wayner 2015)

Yritysmaailman sovelluskehityksessä suosiotaan nostaa mikropalvelut. Mikropalvelut tarjoavat pienempiä koodikokonaisuuksia, integraatioita, nopeaa kehitystä ja vaiheittaisia käyttöönottoja. Useat sovelluskehykset keskittyvät juuri mikropalveluihin. (Woods 2016) Näistä kevyemmistä, nopean sovelluskehityksen sovelluskehyksistä esimerkkeinä ovat Play ja Spring Boot.

Mikropalveluilla pyritään rakentamaan tietojärjestelmiä suurten monoliittijärjestelmien sijaan pienempien, itsenäisten komponenttien varaan. Mikropalvelut edesauttavat nopeaa kehittämistä, koska yksittäiset kehitystiimit voivat keskittyä rakentamaan itseänäisiä komponentteja, jolloin kehitysvauhti on nopeampaa. (Gutierrez 2016, 309)



Kuva 1. Monoliittijärjestelmä vs mikropalvelu (Gutierrez 2016, 309)

Mikropalvelut ovat pieniä ja keskittyvät hoitamaan yhtä tehtävää. Pelkkä pieni koko ei kuitenkaan ole merkittävin tavoite. Jos palvelu on liian pieni, sillä toki maksimoidaan yksittäisestä mikropalvelusta saatava hyöty mutta samalla kasvatetaan ohjelmiston monimutkaisuutta lisäämällä liikkuvia osia. (Newman 2015, 2-3)

Mikropalveluiden komponentit ovat mahdollisimman itsenäisiä ja tilattomia. Mikropalveluiden yksittäiset komponentit kommunikoivat toistensa kanssa kevyiden rajapintojen avulla, ja tuottavat yhdessä sovelluksen kokonaistoiminnan. Komponentit toimivat asynkronisesti, eli ne eivät jää odottelemaan vastauksia toisiltaan - ellei vastauksen välitön saaminen ole välttämätön edellytys toiminnon jatkumiselle. (Hämäläinen 2016)

Kevyistä rajapinnoista yhtenä esimerkkinä on REST -rajapinta. Yhteisten rajapintojen vuoksi jokainen mikropalvelun osa voi olla kirjoitettu eri ohjelmointikielellä. (Hämäläinen 2016)

Nopean sovelluskehityksen sovelluskehysten toiminta ei kuitenkaan ole täysin vailla kritiikkiä. Nopeaa kehitystä tavoitellessaan jotkut näistä kehystyökaluista saattavat rohkaista huonoon ohjelmointitapaan, joka saattaa kostautua pitkällä tähtäimellä. Eräässä konferenssissa kerrottiin Spring Bootin eduksi sen tapa muodostaa olioita suoraan tietokannan rakenteesta ja käyttää näitä olioita suoraan rajapinnoissa. Tällainen yhteys olion ja tietokannan välillä aiheuttaa useimmissa tapauksissa enemmän ongelmia kuin se, että nähtäisi vaivaa erotella nämä käsitteet. (Newman 2015, 54.).

Verrattuna Spring Frameworkiin Spring Boot on kuitenkin tehokas työkalu mikropalveluiden kehittämisessä. Bootin toiminta on yksinkertaistettua ja sillä on kyky

tuoda riippuvuudet modulaarisina sovelluksen käyttöön helpottaen siten RESTful -palveluiden kehittämistä. (Woods 2014)

4 Spring Boot

Spring -sovelluskehys on saanut osakseen kritiikkiä kömpelön XML -määrityksen ja monimutkaisen riippuvuushallinnan takia (Woods 2014). Spring Boot on suunniteltu helpottamaan Spring -pohjaisten sovellusten tekoa poistamalla tai korjaamalla nämä kritiikkiä aiheuttaneet osa-alueet. Verrattuna perinteisiin Spring -projekteihin, Spring Boot vaatii merkittävästi vähemmän määrityksiä, jotta kehittäjä pääsee nopeammin rakentamaan varsinaista liiketoimintalogiikkaa sovellukseen.

Spring Bootia kuvataan Springin sivuilla seuraavasti:

> Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run". We take an opinionated view of the Spring platform and third-party libraries so you can get started with minimum fuss. Most Spring Boot applications need very little Spring configuration. (http://projects.spring.io/spring-boot/)

Mikropalveluiden suosion kasvaessa skaalautuvien, helposti saatavien ja vakaiden sovellusten luomisessa, Spring Boot sopii täydellisesti näiden palveluiden kehittämiseen, jättäen varsinaisen raskaan työn Springille (Gutierrez 2016, 6). Spring Bootin ajatuksena on tarjota tapa, jolla voi helposti tehdä itsenäisiä Spring –sovelluksia. Spring Boot ei vaadi lainkaan XML -määrityksiä kuten perinteinen Spring. (Spring 2016b)

Spring Bootin erikoisuuksiin sisältyy oma komentorivityökalu, suuri määrä kirjastopaketteja, sisäänrakennettu web -palvelin, ja valvontatyökalu nimeltä Actuator.

Spring Boot tukee hyvin RESTful -tapaa tehdä sovelluksia.
Springin RESTful -verkkopalveluissa HTTP -pyynnöt käsitellään kontrollerissa. Oleellisin ero perinteisen MVC -kontrollerin ja REST -kontrollerin välillä on HTTP -vastauksen bodyn luominen. Perinteinen MVC -kontrolleri tuottaa palvelimella datan HTML:ksi, kun REST -kontrolleri luo ja palauttaa JSON -olion. (Spring 2016a).

4.1 Komentorivityökalu, CLI

Spring Bootin sisältämä Command Line Interface (CLI) on työkalu, jonka avulla voi helposti testata ja suorittaa Spring Boot –sovelluksia ilman varsinaista sovelluskehitysympäristöä. CLI käyttää Bootin kirjastopaketteja ja määritysnotaatioita sovellusta käynnistettäessä, ja tietää mitä riippuvuuksia lisätä, jotta sovellus toimisi. Command Line Interface on voimakas työkalu, mutta ei pakollinen osa Spring Bootia. (Walls 2016, 6.)

Command Line Tool voi olla erityisen hyödyllinen nopeassa prototyyppikehitystyössä ja proof-of-concept -tyyppisen projektin kehityksessä, koska kehittäjän ei tarvitse huolehtia oletusmäärityksistä eikä tietyistä import -lauseista jotka CLI osaa itse tuoda koodin kääntämisvaiheessa. (Woods 2014)

4.2 Projektin luonti ja rakenne

Spring Boot projektin voi luoda usealla tavalla: Spring Initializr:n avulla Springin omilta verkkosivuilta tai Spring Boot CLI:lla omalta komentiriviltä, tai omassa kehitysympäristössä Intellij IDEA:lla tai Eclipse -pohjaisessa Spring Tool Suitessa tekemällä suoraan uuden Spring projektin (Walls 2016, 12.). Spring Boot tarjoaa projektin kokoamiseen joko Mavenia tai Gradlea, ja Spring Bootin käyttäminen vaatii tietyt määritykset näiden projektinhallintatyökalujen määritystiedostoihin.

Helpoiten ja täysin ilman ohjelmointiympäristöä projektin voi luoda Spring Initializr:illa osoitteesta http://start.spring.io. Initializr opastaa valinnoissa ja antaa valita tarvittavat aloituskirjastopaketit jo tässä vaiheessa. Gradlea käyttävä Initializrilla luotu projekti muodostaa oletuksena seuraavanlaisen projektirakenteen:

```
├── build.gradle
└── src
    ├── main
    │   ├── java
    │   │   └── myapp
    │   │       └── Application.java
    │   └── resources
    │       ├── application.properties
    │       ├── static
    │       └── templates
    └── test
        └── java
            └── myapp
                └── ApplicationTests.java
```

Kuva 2. Initializrilla luotu projektiperusta (Walls 2016, 14.)

4.3 Starter dependencyt eli kirjastopaketit

Spring Boot tarjoaa yli 40 erilaista ”starter modulea”, kirjastopakettia, helpottamaan sovelluksen alkumäärityksiä. Nämä kirjastopaketit ovat ikään kuin koosteita tiettyyn asiaan tarvittavista riippuvuuksista – esimerkiksi tietokantayhteys, testaustyökalut tai websovellus. (Antonov 2015, 48.)
Näitä kirjastopaketteja voi halutessaan käyttää kokonaisuudessaan, muokata haluamaltaan osin tai olla käyttämättä kokonaan.

Mikäli rakentaisi projektin riippuvuudet alusta asti itse, pitäisi miettiä mikä versio on käytössä mistäkin. Valmiit kirjastopaketit helpottaa siinäkin – versiota ei tarvitse itse kertoa, vaan Spring Bootin versio määrittää suoraan myös kirjastopakettien versiot (Walls 2016, 35.).

Esimerkiksi ”spring-boot-starter-web” moduuli sisältää kaikki seuraavat kirjastot:

- org.springframework.boot:spring-boot-starter,
- org.springframework.boot:spring-boot-starter-tomcat,
- org.springframework.boot:spring-boot-starter-validation,
- com.fasterxml.jackson.core:jackson-databind,
- org.springframework:spring-web,
- org.springframework:spring-webmvc

Spring Bootin automaattiset määritykset käyttävät hyödykseen muun muassa näitä kirjastopaketteja määritelläkseen, mitä sovelluksen käynnistämiseen tarvitaan. Starter-web -pakettia käytettäessä ei tarvitse enää erikseen määritellä muualla, että sovellus on web -sovellus joka tarvitsee käyttöönsä Jacksonin ja Tomcatin. Lisäksi paketin mukana tulee Spring Frameworkin tarjoamat yleisimmin käytetyt osat juuri web -sovelluksille.

Mikäli Spring Boot ei suoraan tarjoa tarvittavaa starter -kirjastopakettia, sellaisen voi rakentaa itse.

4.4 Automaattiset määritykset

Spring Bootin yksi olennaisin osa on sen sisältämät automaattiset määritykset. SpringBootApplication –annotaatio yhdessä sen sisältämän EnableAutoConfiguration -annotaation kanssa mahdollistaa se, että Boot itse päättelee itse mitä sovelluksen ajamiseen tarvitaan. Automaattiset määritykset poistavat suuren osan boilerplate -määrityksiä, jotka vaivaavat perinteisiä Spring -sovelluksia (Walls 2016, 22).

Yksinkertaistettuna automaattiset määritykset ovat sovelluksen käynnistämisen aikasia prosesseja, jotka ottavat huomioon useita tekijöitä ja päättävät niiden perusteella, mitkä Spring Frameworkin määritykset ovat kyseisen sovelluksen kannalta oleellisia ja mitkä eivät ole ja mitä niistä otetaan käyttöön. Spring Boot hoitaa jopa lähes kaksisataa päättelyä aina sovelluksen käynnistyessä. (Walls 2016, 37).

Spring Bootin automaattiset määritykset tarkastelevat luokkapolkua, annotaatioita ja mahdollisia muita määrityksiä ja lisää siihen sovellukseen sopivat teknologiat. Eli kaikki

annetut annotaatiot kertovat sen, kuinka Boot -sovellus määritetään ja kootaan. (Gutierrez 2016, 40.). Näiden kaikkien määritysten perusteella Spring Boot osaa mm. luoda puuttuvat Spring -beanit ja käynnistää sovelluksen sisäänrakennetun Tomcat -palvelimen.

Suuret määritysmäärät ovat keskeinen osa Spring Frameworkiä, ja niitä toki tarvitaan jotta Spring tietää miten sovelluksen kuuluu toimia. Automaattiset määritykset piilottavat Springin määritysluokat näkyvistä, mutta ne kuitenkin tuodaan sovelluksen luokkapolulle. Automaattiset määritykset nojaavat Spring 4.0:ssa julkaistuun ”Conditional” -määritykseen, joka mahdollistaa sen, että tietyt määritykset ovat sovelluksen saatavilla mutta ne voidaan jättää huomiotta mikäli niille ei ole käyttöä. (Walls 2016, 45)

Spring Bootin automaattiset määritykset ovat kuitenkin tarvittaessa helposti ohitettavissa ja korvattavissa omilla määrityksillä missä projektin vaiheessa tahansa. Lisättäessä esim. kokonaan oma DataSource -bean, automaattisen määrityksen tuoma vastaava bean väistyy. Myös tietty yksittäinen automaattisten määritysten tuoma luokka on helposti estettävissä ”exclude” -attribuutilla. (Spring 2016c)

4.5 Actuator

Kun muut Spring Bootin osat tarjoaa helpotusta itse ohjelmointiin, Actuatorin avulla on mahdollista tutkia tarkemmin, miten sovellus toimii. Actuator tarjoaa tuotantovalmiita työkaluja mm. monitorointiin ja mittaamiseen, ja sen avulla voi tarkastella mm. Automaattisten määritysten tekemiä päätöksiä, säikeiden tilaa, sovelluksen käsittelemiä http -pyyntöjä ja muistin käyttöä. (Walls 2016, 7.)

Actuatorin data on mahdollista saada nähtäväksi usealla tavalla: REST -rajapinnan tai SSH:n kautta – SSH:lla jopa sovelluksen ollessa käynnissä (Walls 2016, 7.). Dataa voi tarkastella myös JMX:n (Java Management Extensions) kautta.

Actuatorin käyttöönotto vaatii hieman määrityksiä, mutta työkalua hyödyntämällä on mahdollista saada sovelluksesta optimoidumpi ja skaalautuvampi. Actuatorin kautta saa myös tietoa Spring Bootin automaattisista määrityksistä ja siitä miksi jotain osioita on määrityksissä mukana ja joitain ei (Walls 2016, 128).

4.6 Käyttöönotto

Spring Boot tarjoaa monenlaisia tapoja ottaa sovellus käyttöön. Koska kyseessä on Spring, tarjolla on tietenkin perinteinen tapa tehdä sovelluksesta WAR -paketti Mavenilla

tai Gradlella. Tämä sopii ympäristöihin, joissa sovellus pyörii Java -palvelimella tai pilvipalvelussa.
Spring Boot -sovelluksesta voi tehdä myös JAR -sovelluksen Mavenilla, Gradlella tai Bootin omalla CLI:lla. Tämä on parempi ratkaisu esim. konttiympäristöihin, kuten Docker. Spring Boot tarjoaa nyös mahdollisuuden tehdä sovelluksesta Groovy -skripti. (Walls 2016, 161-162.)

Deployment artifact	Produced by	Target environment
Raw Groovy source	Written by hand	Cloud Foundry and container deployment, such as with Docker
Executable JAR	Maven, Gradle, or Spring Boot CLI	Cloud environments, including Cloud Foundry and Heroku, as well as container deployment, such as with Docker
WAR	Maven or Gradle	Java application servers or cloud environments such as Cloud Foundry

Kuva 3. Spring Boot -sovelluksen käyttöönottovaihtoehdot. (Walls 2016, 162.)

5 Play 2

Play on Java- ja Scala -kieliä tukeva web -sovelluskehys, joka yhdistää modernin websovelluskehityksen tarvitsemat komponentit ja API:t. Play pohjautuu kevyeen ja tilattomaan arkkitehtuuriin. Reaktiivinen malli, johon Play perustuu, tekee Play:stä skaalautuvan ja vähän resursseja (muisti, säikeet ja prosessori) kuluttavan.
(Play 2016f)

Playn suosii ”convention over configuration” -mallia (Van der Mersch 28.1.2016); tämä tarkoittaa sitä, että sovelluskehys tarjoaa valmiiksi mietityt ratkaisut mahdollisimman pitkälle, jolloin sovelluskehittäjän harteille jää vain käytännöstä poikkeavien asioiden määrittely (Wikipedia 2016f).

Ensimmäinen Play sovelluskehys oli rakennettu Javalla, mutta Play 2 on toteutettu kokonaan uusiksi, pääosin Scalalla paremman suorituskyvyn takaamiseksi. Play sopiikin erityisen hyvin juuri web -sovellusten sovelluskehykseksi - vaikkakaan se ei suoraan auta kehittäjää niinkin yleisien asioiden kuin kirjautumis- ja käyttäjärekisteröintitoimintojen kanssa. (Sargent 2015, luku 1)

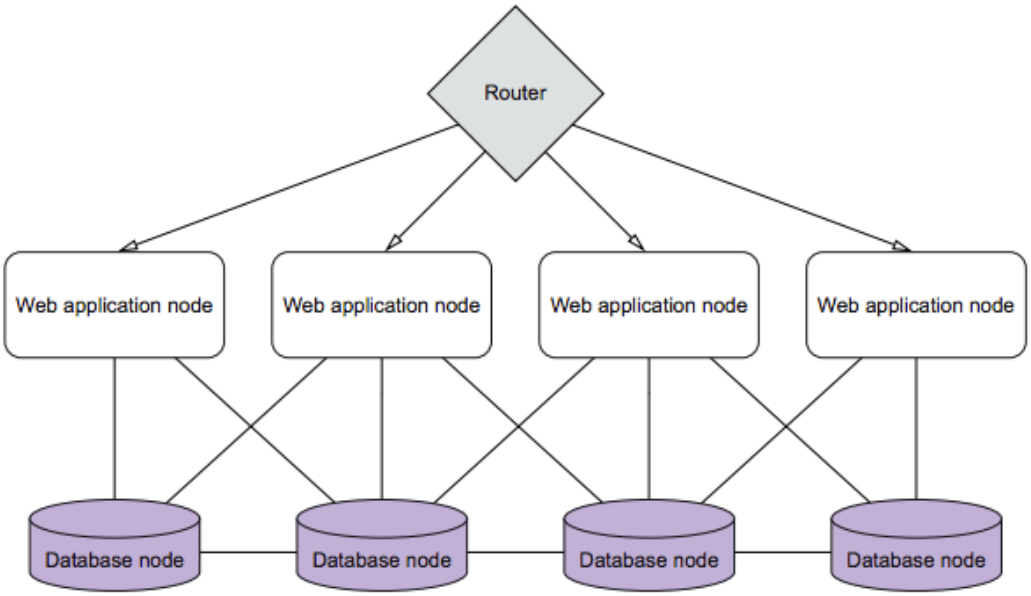
Nykyaikaisten web –sovellusten pitää tukea rinnakkaisuutta, joten myös web -sovelluskehysten tulee tarjota tälle tuki. Play on rakennettu toimimaan usean pitkäkestoisen rinnakkaisen pyynnön kanssa, joten se tarjoaa asynkronisen HTTP API:n perinteisen Servlet API:n sijaan. (Play 2016a)

Play:n ensimmäinen versio ei noudattanut J2EE -abstraktioita, joten Play 2 ei noudata niitä myöskään. Vuosia sitten kehitetyt J2EE:n abstraktiot helpottivat tietyssä määrin sovelluskehittäjän työtä, mutta nykypäivänä ne voivat rajoittaa kehittäjän tekemisiä liikaa. J2EE:n abstraktioiden ohittaminen mahdollistaa Play:lle sen, että serveriä ei tarvitse aina käynnistää uudestaan koodimuutosten testaamiseksi ja sovelluksen kääntämiseksi.
(Petrella 2013, 250-255)

5.1 Tilattomuus

Horisontaalisessa arkkitehtuurissa sovelluksen sama versio on sijoitettu monelle palvelimelle. Palvelimet voi tässä tapauksessa olla fyysisiä tai virtuaalisia, mutta tärkein piirre arkkitehtuurissa on, että palvelimet eivät tiedä toisistaan eivätkä jaa tilaa keskenään. Tällaista jakamattomuutta kutsutaan tilattomaksi arkkitehtuuriksi - jokainen palvelin on itsenäinen, eikä sen olemassaolo tai poissaolo vaikuta millään tavalla muihin palvelimiin

(lukuunottamatta mahdollista kuorman tasausta). Tällaisen arkkitehtuurin etuja ovat mm. helppo skaalautuvuus palvelimia lisäämällä ja helpottuneet palvelinpäivitykset. (Bernhardt 2016, 16)



Kuva 4. Horisontaalinen arkkitehtuurimalli. (Bernhardt 2016, 16)

Play on täysin tilaton, eli se ei säilö mitään tietoa tilasta palvelimella (Petrella 2013, 251). Playn tilaton arkkitehtuuri mahdollistaa edellä mainitun horisontaalisen skaalauksen ja on siten optimaalinen usean yhtäaikaisen pyynnön käsittelyyn ilman että tilaa, kuten sessiota, tarvitsee jakaa pyyntöjen kesken. (Van der Mersch 28.1.2016)

5.2 Versiointi

Playn ensimmäinen versio julkaistiin vuonna 2007, ja nykyinen, 2.5 versio keväällä 2016 (Wikipedia 2016e). Näiden välillä on ollut useita versiota, jotka ovat sisältäneet enemmän ja vähemmän korjauksia ja muutoksia edellisiin versioihin nähden. Playn versiokehitys on ollut aktiivista, ja vaikka pääversioita on ilmestynyt noin kerran vuodessa, pienempiä versioita julkaistaan muutaman kuukauden välein.

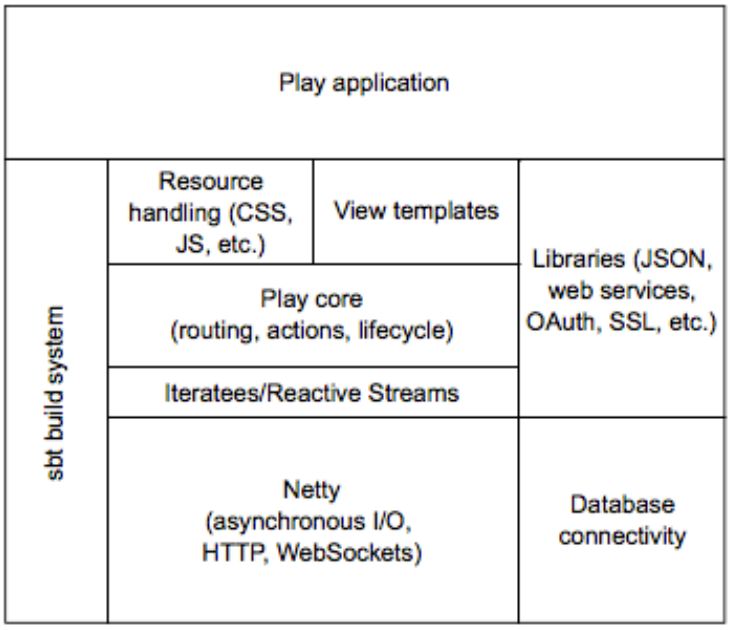
Playn aktiivisesta ja dynaamisesta versiokehityksestä johtuen, useat aikanaan pätevät ja hyödylliset julkaisut, kirjat ja dokumentaatiot eivät pidä paikkaansa Playn uusimmassa versiossa. Myös Playn sisällä on versioiden välillä muutoksia, jotka vaikuttavat suoraan koodin toimivuuteen. Playn aiemmat versiot eivät välttämättä toimi uudessa Playssa, koska joitain toiminnallisuuksia on voitu joko muuttaa tai poistaa kokonaan. (Play 2016d, Play 2016e)

Tässä opinnäytetyössä keskitytään pelkästään Play 2:n ominaisuuksiin.

5.3 Simple Build Tool

Playn Activator sisältää olennaisen osan Playn toiminnallisuutta – Simple Build Toolin. Kuten Spring Bootin CLI:lla, myös tällä työkalulla sovellusta voi ajaa tai testata ilman varsinaista erillistä sovelluskehitystyökalua. Simple Build Tool on avoimen lähdekoodin työkalu projektin rakentamiseen ja riippuvuuksien hallintaan Java ja Scala –projekteille, vastaavasti kuin esim. Maven on Javalle.

Kuten kuvassa 4 näkyy, Play antaa sbt:n haltuun tyypilliset web -sovelluksiin liittyvät asiat kuten selainpuolen resurssit, projektin kääntämisen ja paketoinnin.



Kuva 5. Play -sovelluskehyksen korkean tason arkkitehtuuri. (Bernhardt 2016, 8)

Simple Build Tool on rakennettu Scalalla, ja sen määritykset tehdään myös Scalalla. Scala -kielen hallinta ei ole pakollista sbt:n määrittelyyn, mutta siitä voi olla hyötyä.

5.4 Projektin rakenne ja mallipohjat

Play –projektin voi helpoiten luoda käyttämällä jotain valmista mallipohjaa (template). Templatet ovat valmiita projekteja, jotka sisältää mahdollisesti jo kaikki projektin tarvitsemat osat – tietokantayhteydet tarvittavalle tietokannalle, ajurit kolmansien osapuolten sovelluksille, tai mitä tahansa mitä mieleen saattaa tulla tarvita. Templatet

voivat olla puhtaasti esimerkkejä joita seuraamalla on helpompi rakentaa oma sovellus, tai valmiita projektin alkuja, joiden kanssa pääsee helposti alkuun omassa projektissa (Play 2016b).
Useat templatet ovat kuitenkin vanhemmille Play –versioille, eivätkä ne siten ole yhteensopivia uusien versioiden kanssa.

Playn projektirakenne eroaa hieman Spring Bootin rakenteesta. Playssa kansiot on selkesti jaoteltu MVC –mallin mukaisesti.

Kuva 6. Play -sovelluksen rakenne ensimmäisen kääntämisen jälkeen. (Play 2016c)

Playn “app” –kansio sisältää kaikki MVC -mallin mukaiset paketit. Määritystiedostoista vain projektin päämääritystiedosto, build.sbt, on suoraan juuressa, muut ovat joko conf tai project -kansioiden alla. Conf -kansio on sovelluksen omille määritystiedostoille, ja kaikki oman projektin lisämäärityket on suositeltavaa tallettaa sinne. Project -kansio on SBT:n määritystiedostoille. Public -kansiossa ovat resurssit, joita palvelin pääsee suoraan käyttämään – tyylitiedostot, javascriptit ja kuvat. (Play 2016c)

5.5 Akka

Play käyttää Akka:n päälle rakennettua täysin asynkronista mallia, joka auttaa sovelluskehystä skaalautumaan helposti ja ennakoitavasti ja minimoimaan resurssien käytön. (Play 2016g)

Akka, kuten Scala ja Play sovelluskehys, on osa Typesafe Stack 2 -alustaa (uudelta nimeltään Lightbend Reactive Platform). J2EE:n perustuessa pääosin servletteihin, jolloin jokainen pyyntö vaatii käyttöönsä yhden säikeen, Akkan Actor Model toimii yhtäaikaisten pyyntöjen osalta eri tavalla; palvelimen kyky käsitellä säikeiden määrää ei rajoita yhtäaikaisten käyttäjien määrää. J2EE -servlettien toiminta on säikeiden suhteen blokkaavaa, ja Akka tuo mukanaan blokkaamattoman tavan toimia. (Petrella 2013, 170)

5.6 Käyttöönotto

Käyttöönottoa ennen Playssä on määriteltävä Application Secret application.conf -tiedostoon. Play käyttää Secretiä mm. istunnon evästeisiin ja muihin salauksiin.
(Play 2016h)

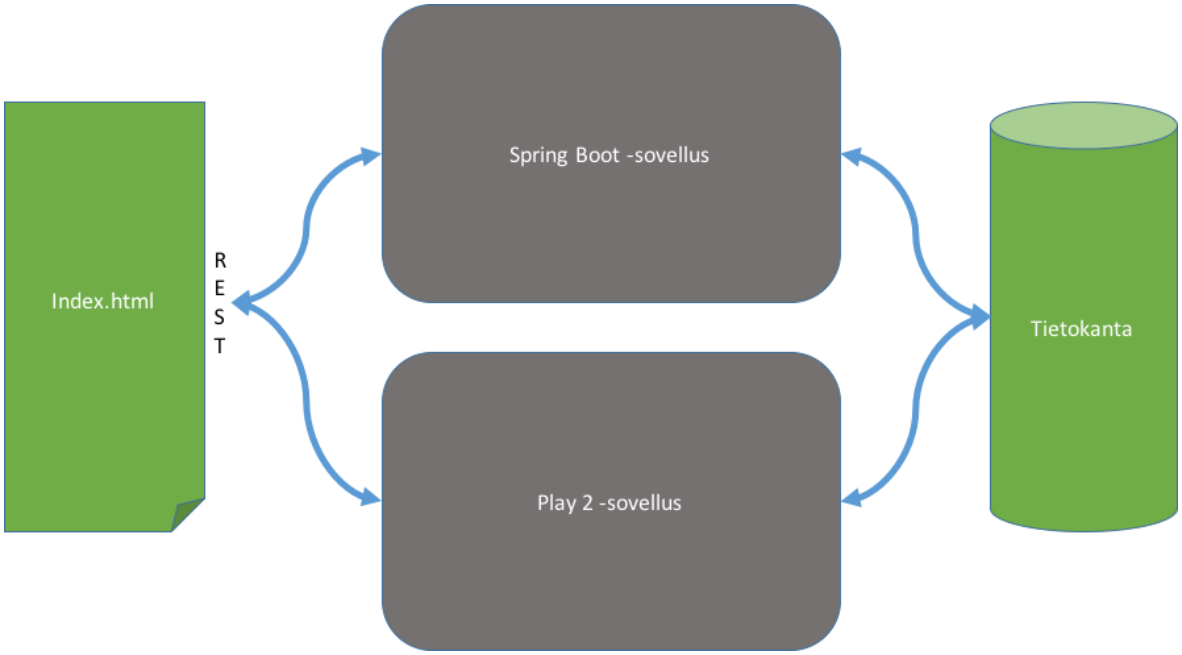
Play -sovelluksesta muodostetaan ZIP -paketti, johon Play paketoi kaikki tarvitsemansa JAR -tiedostot ja API-dokumentaation. Pakattu ohjelmapaketti ei vaadi mitään muita riippuvuuksia, lukuunottamatta Java -asennusta, ja se voidaan helposti asentaa mihin tahansa pilvipalveluun tai palvelimelle. ZIP -paketoinnille Play tarjoaa myös useita muita vaihtoehtoja, kuten tar.gz ja Microsoft Installer (MSI) -tiedostomuodot.
(Play 2016i)

6 Ideasta koodiksi

Opinnäytetyön tuotoksena oli ajatus rakentaa yksinkertaiset kauppalistasovellukset, josta käyttäjä voisi katsoa ostettavat asiat, lisätä listalle ostettavia asioita ja poistaa niitä. Suunnitelmissa oli myös rakentaa kirjautumistoiminto sivustolle, koska sitä kautta pääsisi tutkimaan mitä sovelluskehykset tarjoavat autentikoinnin avuksi. Sivuston ulkonäköön ei ollut tarkoitus kiinnittää suurta huomiota.

6.1 Suunnitelma

Kumpaakin sovelluskehystä käyttäen oli tarkoitus tuottaa niin kutsuttu moderni yhden sivun RESTful websovellus. Koska haluttin keskittyä enemmän näiden sovelluskehysten toimintaan kuin sovelluksen muihin osiin, oli ajatuksena, että tietokanta ja käyttöliittymä on identtinen kummallekin sovellukselle – mahdollisuuksien mukaan.



Kuva 7. Ajatus sovelluksen arkkitehtuurista

Käyttöliittymä suunniteltiin yksinkertaiseksi ja selkeäksi ilman ylimääräisiä toimintoja. Käyttöliittymältä piti voida nopeasti ja helposti lisätä ja poistaa tuotteita.

Kuva 8. Käyttöliittymäproto

Tietokannaksi valittiin PostgreSQL, koska se oli entuudestaan tuttu ja se on osoittautunut helpoksi ja toimivaksi.

Ennen varsinaista sovelluskehyksen asennusta ja koodausta, kuvattin tietokanta sekä rajapinnat, jotta itse kehitystyö sujuisi jouhevammin. Tietokantaan tuli toistaiseksi yksi taulu joka sisälsi ostettavan tavaran minimitiedot – suunnitelmissa oli laajentaa tietokantaa mikäli kirjautuminen tulisi rakennettavaksi. Sovelluskehykseen tutustumisen kannalta laajempi tietokanta ei ollut oleellinen.

Tietokanta sisälsi tässä vaiheessa vain taulun ”item”. Tietotyypit on jätetty määräämättä, koska tietokantataulu luodaan sovelluksen käynnistyksen yhteydessä.

Item
id
name
amount

Kuva 9. Tietokantataulun rakenne

JSON objektimalli oli yksinkertainen, ja koska toimintoja sovelluksella oli kolme, kolme rajapintaa riitti:

```
{"id": 1, "name": "appelsiineja", "amount": "7"}
```

Kuva 10. JSON objektimalli

Kauppalistan haku:	GET	/shoppinglist
Kauppalistaan lisäys:	POST	/shoppinglist
Kauppalistasta poisto:	DELETE	/shoppinglist/id

6.2 Spring Boot –sovellus

Ensimmäiseksi otettiin työn alle Spring Boot, koska se oli täysin vieras, vaikkakin Spring Framework itsessään oli jo melko lailla tuttu. Kirjallisuutta tästä aiheesta löytyi onneksi reilusti, vaikkakin enemmän perinteisestä tavasta rakentaa sovellus Springin omalla MVC -rakenteella, ja vähemmän RESTful -tavasta hyödyntää Spring Bootia.

Sovelluskehys itsessään oli helppo asentaa ohjeiden mukaan, CLI:n avulla komentoriviltä. Projektinkoontityökaluksi valittiin Gradle, koska Maven oli aiemmin tuttu ja tässä vaiheessa haluttiin tutustua samalla tähänkin uuteen asiaan.

6.2.1 Määritykset kohdilleen

Tietokantayhteyden toimivuutta varten piti ensin määrittää riippuvuudet kahteen paikkaan – Spring Bootin omaan määritystiedostoon nimeltä Application.properties, sekä Gradlen määritystiedostoon Build.gradle.

Application.properties ja Build.cradle vaativat seuraavat lisäykset riippuvuuksien määrityksiin, jotta yhteys postgreSQL:ään toimisi:

```
spring.datasource.driver-class-name=org.postgresql.Driver
spring.datasource.url=jdbc:postgresql://localhost:5432/shoppinglist

compile group: 'postgresql', name: 'postgresql', version: '9.1-901.jdbc4'
```

Kuva 11. Tietokantayhteyden määritykset

Build.gradleen piti lisätä frontendiä varten vielä muutama riippuvuus, kuten node.js, npm ja rest, jonka vaatimat riippuvuudet sai kätevästi Spring Bootin omasta kirjastopaketista ”spring-boot-starter-data-rest”.

6.2.2 Backend kuntoon

Määritysten asettamisen jälkeen luotiin tietokanta, ja lähdettiin rakentamaan backendin toiminallisuutta. Sovelluksen logiikka on yhdessä kontrollerissa, joka ottaa vastaan selaimelta tulevat pyynnöt ja käsittelee ne.

```
@RestController
@RequestMapping("/shoppinglist")
public class ShoppingListController {

	private ShoppingListRepository shoppingListRepo;

	@Autowired
	public ShoppingListController(ShoppingListRepository shopListRepo) {
	this.shoppingListRepo = shopListRepo;
	}

	@ResponseBody @RequestMapping(method = RequestMethod.POST)
	public Item addItem (@Validated @RequestBody Item item) {
		shoppingListRepo.save(item);
		return item;
	}

	@ResponseBody @RequestMapping(value="", method = RequestMethod.GET)
	public Iterable<Item> getAllItems() {
		return shoppingListRepo.findAll();
	}

	@RequestMapping(value="/{id}", method=RequestMethod.DELETE)
	public Long delete(@PathVariable Long id) {
		shoppingListRepo.delete(id);
		return id;
	  }
}
```

Kuva 12. Spring Boot -sovelluksen kontrolleri

RestController ja RequestMapping -annotaatiot kertovat Spring Bootin automaattisille määrityksille, että kyseessä on websovellus. RequestMapping -annotaatio kertoo, että kaikki pyynnöt, jotka tulee /shoppinglist -URL:iin, käsitellään tällä kontrollerilla.

Autowired -annotaatio luo repository olion, jotta sitä voidaan käyttää kontrollerin metodeissa tietokantakäsittelyyn.

Tietokantataulu luodaan sovelluksen käynnistyksen yhteydessä suoraan Java -luokasta. Koska tässä sovelluksessa käytetään JpaRepositorya, Item -luokka piti annotoida Entityksi.

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

@Entity
public class Item {

		@Id
		@GeneratedValue(strategy=GenerationType.IDENTITY)
		private long id;

		@NotNull
		@Size(min = 1, max = 30)
		private String name;

		@Size(max = 20)
		private String amount;

		public long getId() {
			return id;
		}

		public void setId(long id) {
			this.id = id;
		}
		public String getName() {
			return name;
		}
		public void setName(String name) {
			this.name = name;
		}
		public String getAmount() {
			return amount;
		}
		public void setAmount(String amount) {
			this.amount = amount;
		}
}
```

Kuva 13. Spring Boot -sovelluksen Item -luokka

Sovelluksen repository valittiin käyttämään JpaRepositorya, koska sillä on laajemmat ominaisuudet kuin CrudRepositorylla, joka olisi ollut toinen vaihtoehto. JpaRepositorylla voi esim. suoraan valmiilla metodilla hakea kaikki tietokannassa olevat tiedot. Tietokantayhteyden hoitava Repository -luokka oli siis näinkin yksinkertainen:

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface ShoppingListRepository extends JpaRepository<Item, Long>{

}
```

Kuva 14. Spring Boot -sovelluksen repository

Tässä vaiheessa puuttui vielä frontend -toiminnallisuus, mutta backendin toiminta voitiin todentaa Chromen apuohjelmalla Advanced REST Client (ARC). ARC:lla saatiin lähetettyä oikeanlaisia pyyntöjä, joilla pystyttiin varmistamaan että koodi ja yhteys tietokantaan toimii.

Jpa:n Hibernate loi Item -tietokantataulun automaattisesti ja generoi Id:t tauluun sovelluksen käynnistyessä.

Tässä vaiheessa huomattiin, että tietokantaa varten on oleellista tehdä validoinnit tietokannan muodostavassa Java -luokassa, jotta kanta ei heti alkuun mene kuraksi. Item -luokkaan lisättiin validointiannotaatiot.

6.2.3 Pääluokka

ShoppingListApp –niminen pääluokka on luokka, joka sisältää Main -metodin ja varsinaisesti ajaa sovelluksen. Tämä luokka on annotoitu SpringBootApplication –annotaatiolla.

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class ShoppingListApp {

            public static void main(String[] args) {
                        SpringApplication.run(ShoppingListApp.class, args);
            }
}
```

Kuva 15. Spring Boot -sovelluksen pääluokka

6.2.4 Frontend ja haasteet

Spring Boot tarjoaa frontendin tekemiseen valmiita Thymeleaf -pohjia, joille olisi ollut valmis tuki saatavilla helposti, ja esimerkkejä saatavilla runsaasti. Koska haluttiin rakentaa nimenomaan yhden sivun RESTful sovellus, jouduttiin tässä vaiheessa pureutumaan hieman opinnäytetyön pääfokuksesta poikkeavaan asiaan – Ajaxiin ja jQueryyn. JavaScript, jQuery ja Ajax eivät olleet tuttua asiaa ennestään, joten tässä vaiheessa törmättin ensimmäisiin haasteisiin ja käytettiin hyvän verran aikaa näiden ymmärtämiseen.

Sovelluksen frontend –toimintalogiikka koodattin kokonaisuudessaan app.js -tiedostoon (liite 1). Sivun latauduttua haetaan kaikki tietokannassa olevat asiat näkyviin. Koska pyynnöt tehdään Ajaxilla, sivulla lisätty asia näkyy listalla heti ja poistettu asia poistuu listalta heti ilman erillistä päivityspainiketta tai sivun uudelleenlatausta.

6.3 Play -sovellus

Play -sovelluksen aloitus jätettiin hieman suunnitelmaa myöhempään, koska sen ajateltiin olevan helppo. Tuoreessa muistissa oli edellinen projekti, jossa käytettiin Play 2 -sovelluskehystä ja sen kanssa ei silloin ollut suurempia ongelmia.

6.3.1 Asennus ja aloittamisen vaikeus

Kuten Spring Boot, myös Play tarjosi useita vaihtoehtoja sovelluskehyksen asentamiseen. Asennuksen olisi voinut tehdä joko lataamalla ja purkamalla pakatun tiedoston itse tai tekemällä uuden projektin graafisen käyttöliittymän, Activator UI:n, kautta. Koska komentorivityöskentely tuntui luontevammalta, aloitettiin projekti asentamalla Lightbend Activator, joka sisälsi Simple Build Toolin. Ohjeista huolimatta Play asennettiin Homebrewillä, ja uusi Java -projekti luotiin Activatorilla:

```
$ activator new shoppinglist play-java
```

Kuva 16. Uuden Java -projektin luonti Playn Activator -työkalulla

Itse koodaustyö tehtiin Eclipsellä. Play tarjosi valmiin ratkaisun Eclipse -rakenteen luomiseen, kunhan Playn plugins.sbt –tiedostoon lisäsi sbteclipse -liitännäisen. Tämä jälkeen projektin Eclipse -rakenteen luonti onnistui komennolla ”Eclipse” ja projekti voitiin tuoda Eclipseen normaalilla Import –toiminnolla.

Tässä vaiheessa alkoi olla selvää, että Play vaikuttaa hieman haastavammalta kuin Spring Boot.

Oletusprojekti toimi hyvin. Siitä rohkaistuneena lähdettiin muokkaamaan oletusprojektia oman projektin tarpeisiin kirjojen ja internetin avulla.

6.3.2 Määritykset ja riippuvuudet

Play sisältää kolme määritystiedostoa – plugins.sbt, build.sbt ja application.conf. Tietokantayhteyden luomiseksi kahteen näistä piti lisätä PostgreSQL -riippuvuudet. Viimeisin oikea tieto näistä löytyi Playn uusimman version dokumentaatiosta – kirjojen malleja kokeilemalla mikään ei toiminut.

6.3.3 Reittirajapinnat

HTTP -pyyntöjen käsittelyä varten lisättin routes –tiedostoon JSON rajapinnat:

```
GET     /shoppinglist      controllers.ShoppingListController.listItems
POST    /shoppinglist      controllers.ShoppingListController.addItem
DELETE  /shoppinglist/:id  controllers.ShoppingListController.deleteItem(id: Long)
```

Kuva 17. Play -sovelluksen JSON -rajapinnat

Huomattavaa näissä oli, että mikäli yksikin näistä poluista oli vähänkään väärin, sovellus ei suostunut kääntymään.

6.3.4 Tietokantayhteyden muodostaminen

Play käyttää JDBC yhteyden hoitamiseen Hikari Connection Poolia.
Dokumentaation suosittelema PostgreSQL -ajuri ei kuitenkaan suoraan toiminut Hikarin kanssa, joten build.sbt –tiedostoon piti päivittää uudempi ajuri.

Tässä vaiheessa piti luoda Activatorissa Eclipsen projektitiedostot uudestaan, koska Eclipse ei huomaa muutoksia build.sbt -tiedostossa. Tämä asia tuli eteen useammankin kerran sovellusta tehdessä, ja ongelman ratkaisemiseksi vaihtoehtona olisi ollut toisen IDE:n asentaminen.

Koska Hibernate toimi Spring Bootin kanssa, oletettiin sen toimivan tässäkin. Automaattinen tietokantataulun generointi ei kuitenkaan lähtenyt toimimaan. Muutaman tunnin tutkimisen tuloksena saatiin selville, että Playn Evolutions -kirjasto ei vaikuta toimivan Hibernaten kanssa. Spring Boot käytti omaa, laajennettua JPA -toteutusta, kun taas Playlla ei ole Hibernaten kanssa vastaavaa ratkaisua. Mikäli haluaisi käyttää

Hibernatea, pääsisi käyttämään Javan omaa JPA:ta, jolla ei suoraan ole valmista metodia esim. tietokannasta poistamiselle. Tämän johdosta päädyttiin käyttämään Playn paremmin tukemaa Ebeansia.

Ebeansin käyttöönottoa varten määriteltiin Ebean -liitännäiset plugins.sbt, build.sbt, sekä application.conf -tiedostoihin. Lisäksi Item -luokka piti saada käyttämään Ebeansin Model -luokkaa.

Tietokantayhteyttä varten tarvittavat määritykset oli tässä vaiheessa kunnossa, mutta sovellus ei osannut käyttää olemassaolevaa taulua. Playn Evolution olisi halunnut luoda uuden Item -taulun koska Spring Bootin luomassa taulussa oli hieman erilainen tietotyyppi ID:n osalta. Taulun luonti ei kuitenkaan onnistunut, koska sen niminen oli jo olemassa. Periaatteesta tavoitteena oli käyttää samaa tietokantaa ja samaa taulua molemmilla sovelluksilla, joten vaihtoehtoja oli tässä vaiheessa kolme:

1. Olemassa olevian taulujen droppaus, jolloin annetaan Playn luoda taulut. Katsottaisi, toimisiko Spring Boot tuolla taululla.
2. Selvitettäisi, miten Playn saa käyttämään olemassa olevaa taulua, eikä luo taulua uudestaan Evolutionsilla.
3. Rakennettaisi sovelluksille omat taulut.

Toinen vaihtoehto olisi vienyt aika paljon aikaa, ja sen jättämisen toiseksi vaihtoehdoksi perusteltiin sillä, että Spring Boot tuntui olevan joustavampi asioissa. Aloitettiin siis ensimmäisestä vaihtoehdosta, ja se toimi täysin halutusti. Spring Boot vaikutti olevan joustavempi taulurakenteen suhteen.
Nyt saatiin tallennettua tietokantaan ostettavia tavaroita – ARC:n avulla!

6.3.5 Kontrolleri toimintakuntoon

Backend –logiikka koodataan myös Play:ssa kontrolleriin. Tämän sovelluksen kontrolleri koodattiin käyttämään Play MVC:n omaa Controller -luokkaa:

```
import models.Item;
import play.libs.Json;
import play.mvc.*;
import java.util.List;
import com.avaje.ebean.*;
import com.fasterxml.jackson.databind.JsonNode;

public class ShoppingListController extends Controller{

private static Finder<Long, Item> find = new Finder<Long, Item>(Item.class);

    @BodyParser.Of(BodyParser.Json.class)
    public Result addItem() {
            JsonNode json = request().body().asJson();
            if(json == null) {
                 return badRequest("Ei sopiva muoto!");
            }
            Item item = Json.fromJson(json, Item.class);
            Ebean.save(item);
            return ok(Json.toJson(item));
    }

    public Result deleteItem(Long id) {
            Ebean.createSqlUpdate("DELETE FROM item WHERE id=" + id).execute();
            return ok(Json.toJson(id));
    }

    public Result listItems() {
            List<Item> items = find.all();
            return ok(Json.toJson(items));
    }
}
```

Kuva 18. Play -sovelluksen kontrolleri

Playn Controller -luokka ei ollut CRUD -toimintojen osalta aivan yhtä pitkälle jalostettu kuin Spring Bootin vastaava. Lisääminen sujui vaivatta, mutta poistamiselle ei ollut valmista metodia. Poistamisen osalta oltiin valinnan edessä – lisätäänkö Item -luokkaan ”find” metodin jossa haettaisi ensin poistettava ID kannasta, kuten useissa esimerkeissä oli tehty, vai kirjoitetaanko SQL -lause suoraan tämän kontrollerin deleteItem -metodiin. Päädyttiin kirjoittamaan SQL -lause, koska find -metodia ei tarvittaisi tässä sovelluksessa mihinkään muuhun, ja poistamisen yhteydessä sovelluksella on ID jo tiedossa joten sitä on turha hakea tietokannasta uudestaan. Mikäli esim. ostoksen nimen perusteella haku olisi tullut toteutukseen, olisi erillinen find -metodi todennäköisesti kannattanut tehdä.

Tietokannassa olevan datan listaukseen käytettiin Playn omaa Finder -luokan find -metodia.

Huomattiin, että mikäli vastaanotettava JSON on virheellistä muotoa, Controller ei välitä tästä, vaan jää ikilooppiin. Controllerin BodyParserissa on kuitenkin mahdollista määrittää erikseen, että sisääntuleva data on muotoa JSON – epäselväksi jäi, mitä hyötyä tästä määrityksestä on, mikäli virheellinen JSON ei aiheuta virheilmoitusta.

7 Sovelluskehysten erot ja samankaltaisuudet

Play ja Spring Boot ovat molemmat periaatteiltaan aika samanlaisia – sovelluskehyksiä nopeaan kehitykseen. Playlla tuntuu olevan vahvemmat mielipiteet asioista, kun Boot on helpommin muokattavissa omiin tarkoituksiin sopivaksi ilman suurempia haasteita.

Vaikka käyttöasteita ja käyttäjämääriä on suoraan mahdoton vertailla, Spring (MVC ja Boot) vaikuttaa olevan selkesti suosituin Java -sovelluskehys. Play taas vaikuttaa olevan käytetyin silloin kun ohjelmointikielenä on Scala.

Spring Bootin eduksi on luettava, että sitä käsittelevää kirjallisuutta löytyy huomattavasti enemmän kuin Playta käsittelevää aineistoa. Tämä johtunee siitä, että Springillä on sovelluskehittäjien keskuudessa jo laaja käyttäjäkunta ja tuote on suhteellisen tunnettu. Playn nopea ja sovelluskehyksen perusrakenteita muuttava versiokehitys puolestaan aiheuttaa sen, että jo vuoden takaiset dokumentit ja kirjallisuus saattaa olla aivan käyttökelvotonta.

Vaikka Spring Boot on näppärä ja nopea käyttää, sitä ei kuitenkaan voi käyttää pelkästään omana itsenään. Vaikka Spring Bootin sanotaan olevan itsenäinen sovelluskehys, Spring Framework vuotaa selkeästi läpi ja aiheuttaa sen, että Spring Bootia ymmärtääkseen on ymmärrettävä ainakin jonkun verran perinteistä Springiä. Spring Bootissa on myös melko paljon konepellin alla olevaa piilotettua toiminnallisuutta, kuten automaattiset konfiguraatiot, joka monimutkaistaa sovelluskehyksen nopeaa sisäistämistä ja käyttämistä.

Playn vahvuutena mutta myös ongelmana on Scala ja Scalan näkyminen sovelluskehyksen toiminnassa. Play on helppo asentaa ja ottaa käyttöön nopeassa ja yksinkertaisessa projektissa, mutta Simple Build Toolin määritykset vaativat sitten enemmän vaivaa etenkin Scalaa osaamattomalta kehittäjältä. Playn parhaimmat ominaisuudet pääsisivät esille Scalalla ohjelmoiduissa sovelluksissa. Javalla on vaikeampi tai jopa mahdoton tuottaa sellaista sovellusta, joka hyödyntäisi Playn ominaisuuksia, kuten blokkaamattomuutta ja asynkronisuutta.

Niin Play kuin Spring Boot ovat ns. full stack -sovelluskehyksiä. Play vaikuttaa kuitenkin selkeästi olevan enemmän pelkkä ketterä web sovelluskehys, joka on suunnattu nimenomaan funktionaaliseen ohjelmointiin Scalalla. Spring Boot on selkeästi näistä kahdesta oikeampi sovelluskehys Java -maailmaan tai perinteisempään ohjelmistokehitykseen. Playn koko rakenne on suunniteltu tukemaan blokkaamatonta

ohjelmointitapaa, kun Spring ei ota asiaan kantaa - eikä sen oikeastaan tarvitsekaan koska se nojaa täysin JavaEE -maailmaan.

Sovelluskehyksiä voi verrata myös käyttöönoton ja kehittämistyön näkökulmasta. Play sisältää ”hot reload” -ominaisuuden, joka käytännössä sallii koodin muuttamisen lennosta, ja uusien ominaisuuksian käyttöönotto vaatii pelkän selaimen virkistämisen. Play toimii tässä erityisen nopeasti siksi, että se käynnistää pelkästään classloaderin aina uudestaan, eikä koko Java -virtuaalikonetta. Spring Bootissa on vastaavanlainen toiminto, joka antaa tehdä muutoksia koodiin sovelluksen ollessa käynnissä - tämä toiminta vaatii tiettyjä lisämäärityksiä ja on hitaampaa kuin Playn toiminto. Kehittäjän työ on siis Playlla selkeästi nopeampaa.

Sovelluskehykset tarjoavat erilaisia käyttöönottovaihtoehtoja sovelluksilleen. Play tarjoaa useampia vaihtoehtoja mutta suosittelee pakattua ZIP -tiedostoa. Spring Boot -sovelluksesta saa tehtyä joko WAR- tai JAR -tiedoston. Molemmissa on omanlaisensa tuki sille, että sovelluksen saa helposti siirrettyä pilvipalveluun.

Molemmat sovelluskehykset käyttävät testauksessa jUnit -kehystä. Spring Boot tukee myös Mockitoa tietynlaisten testien ajamiseen. Yksikkötesteistä Play tarjoaa useita valmiita malleja projektin asennuksen mukana tulleissa esimerkkitiedostoissa. Spring Bootissa ei vastaavaa ole, vaan kehittäjän vastuulla on tietää miten yksikkötestit rakennetaan.

Ala olevassa taulukossa on vertailtu näiden sovelluskehysten osia:

	Play 2	Spring Boot
Projektinluonti- ja hallintatyökalu	Simple Build Tool	Maven/Gradle
ORM	Ebeans (Hibernate)	Hibernate
Testaustuki	jUnit	Mock, jUnit
Tietoturvamoduuli	Core Security Module	Spring Security
Oletuspalvelin	Netty	Tomcat
Komentorivityökalu	Activator	CLI, Actuator
JSON -tuki	Jackson	Jackson

Vanhin tuettu Java	1.8	1.7
HTTP reititysmääritys	Omassa routes -tiedostossa	Suoraan kontrollerissa

8 Pohdinta

Opinnäytetyön alussa olin vakuuttunut, että nämä sovelluskehykset, kuten kaikki sovelluskehykset, ovat melko samankaltaisia. Tutkiminen ja työskentely näiden sovelluskehysten kanssa sai kuitenkin huomaamaan, että kyseessä on täysin eri maailman sovelluskehykset. Opinnäytetyön aikana ymmärsin, että sovelluskehyksiä on todella erilasia, ja pelkästään niinkin kapealla alalla kuin nopean kehityksen web-sovelluskehysten keskuudessa erot voivat olla huomattavia.

Odotusten vastaisesti, Play ei ollut helppo. Playta markkinoidaan erityisen nopean käyttöönoton kehyksenä, ja ainakaan itseä tuo ei tämän työn perusteella vakuuttanut. Osa sovelluskehyksen toiminnasta olisi vaatinut todella syvällistä ymmärrystä ohjelmoinnista ja sovellusten sisäisestä toiminnasta, jotta siitä olisi saanut irti kaiken mahdollisen. Näin pienen ja yksinkertaisen sovelluksen rakentaminen Playn kanssa olisi saanut olla helpompaa.

Spring Boot puolestaan lunasti lupauksensa helposta ja nopeasta sovelluskehityksestä. Bootin vaikeimmaksi osuudeksi koin automaattiset konfiguraatiot, koska niiden toiminnan ymmärtämiseksi olisi pitänyt ymmärtää paljon enemmän Spring Frameworkia. Tämän pienen sovelluksen luomisessa ei kuitenkaan haitannut se, ettei pellin alla tapahtuvaa toiminnallisuutta heti eikä täydellisesti ymmärtänyt.

Opinnäytetyötä tehdessä pääsin syventämään tietoani sovelluskehyksistä ja sovelluskehityksestä. Työn jälkeen tiedän paremmin, mihin sovelluskehyksiä voi käyttää ja projektista riippuen osaan valita oikeanlaisen kehyksen. Aihe oli mielenkiintoinen, mutta niin laaja että tuntui että tästä olisi voinut kirjoittaa paljon enemmänkin. Opinnäytetyön aiheeksi olisi riittänyt syvällinen tutustuminen yhteen sovelluskehykseen - tässä jäi sellainen tunnelma, etten päässyt tarpeeksi syvälle sisään kumpaankaan. Toisaalta, sain paljon muuta arvokasta tietoa jota pystyn hyödyntämään työssäni.

Kumpikaan näistä sovelluskehyksistä ei ehkä olisi ollut tällaiseen työhän täysin ihanteellinen valinta. Rakennettaessa todella pientä Java -pohjaista sovellusta valitsisin tämän tiedon valossa ehkä jonkun muun sovelluskehyksen. Spring Boot olisi muuten oivallinen valinta, mutta Spring Frameworkin painolasti tuntuu turhan raskaalta tällaiseen.

Nämä nopean kehityksen sovelluskehykset ovat ajankohtaisia etenkin mikropalveluiden yleistyessä. Spring Boot on pari vuotta vanha, ja koska sovelluskehyksellä on jo vakaa pohja, sen käyttö lisääntynee. Play 2 on vuoden vanhempi, mutta vaikka funktionaalisten

ohjelmointikielten käyttö yleistyykin, Play saattaa jäädä vain Scala -kehittäjien työkaluksi. Pitäisin todennäköisenä, että Spring laajentaisi jonkun sovelluskehyksen tukemaan täysin funktionaalista ohjelmointia. Koska Javalla ei tällä hetkellä pysty tekemään täysin funktionaalista ohjelmointia, tämä vaatisi muutoksia Javaan tai Springin perusteisiin, joten sellainen sovelluskehys ei aivan lähitulevaisuudessa ole valmis.

Lähteet

Antonov, A. 2015. Spring Boot Cookbook. Packt Publishing Ltd. United Kingdom.

Bernhardt, M. 2016. Reactive Web Applications. Manning Publications Co. USA.

Gutierrez, F. 2016. Pro Spring Boot. Apress Media, USA.

Hämäläinen, P. 2016. Mikropalvelut nousivat hypen huipulle – mitä hyötyä niistä on?. Luettavissa: http://www.tivi.fi/Kaikki_uutiset/mikropalvelut-nousivat-hypen-huipulle-mita-hyotya-niista-on-6534379. Luettu: 18.10.2016

Newman, S. 2015. Building Microservices. O'Reilly Media. USA.

Petrella, A. 2013. Learning Play! Framework 2. Packt Publishing Ltd. United Kingdom.

Play 2016a. Introducing Play 2, Built for asynchronous programming. Luettavissa: https://www.playframework.com/documentation/2.5.x/Philosophy. Luettu: 20.9.2016.

Play 2016b. Installing Play. Luettavissa: https://www.playframework.com/documentation/2.5.x/Installing. Luettu: 18.9.2016.

Play 2016c. Anatomy of a Play application. Luettavissa: https://www.playframework.com/documentation/2.5.x/Anatomy. Luettu: 2.10.2016.

Play 2016d. Play 2.5 Migration Guide. Luettavissa: https://www.playframework.com/documentation/2.5.x/Migration25. Luettu: 1.10.2016.

Play 2016e. Play Change Log. Luettavissa: https://www.playframework.com/changelog. Luettu: 1.10.2016.

Play 2016f. Play 2.5.X Documentation. Luettavissa: https://www.playframework.com/documentation/2.5.x/Home. Luettu: 19.10.2016

Play 2016g. Play Framework. Luettavissa: https://www.playframework.com/. Luettu: 14.10.2016

Play 2016h. The Application Secret. Luettavissa: https://www.playframework.com/documentation/2.5.x/ApplicationSecret. Luettu: 25.10.2016

Play 2016i. Deploying your application. Luettavissa: https://www.playframework.com/documentation/2.5.x/Deploying. Luettu: 25.10.2016

Sargent, W. 2015. Play in Practice. O'Reilly Media. USA.

Spring 2016a. Building a RESTful web-service. Luettavissa: https://spring.io/guides/gs/rest-service/. Luettu: 30.8.2016.

Spring 2016b. Spring Boot. Luettavissa: http://projects.spring.io/spring-boot/. Luettu: 29.8.2016

Spring 2016c. 16. Auto-configuration. Luettavissa: http://docs.spring.io/spring-boot/docs/current/reference/html/using-boot-auto-configuration.html. Luettu: 19.10.2016

Van der Mersch, V. 28.1.2016. Building a REST API in Java and Scala Using Play Framework - Part 1. Luettavissa: http://nordicapis.com/building-a-rest-api-in-java-scala-using-play-framework-2-part-1/. Luettu: 15.10.2016.

Walls, C. 2016. Spring Boot in Action. Manning Publications Co. USA.

Wayner P. 2015. 7 reasons why frameworks are the new programming languages. Luettavissa: http://www.infoworld.com/article/2902242/application-development/7-reasons-why-frameworks-are-the-new-programming-languages.html . Luettu: 18.10.2016

Wikipedia 2016a. MVC-arkkitehtuuri. Luettavissa: https://fi.wikipedia.org/wiki/MVC-arkkitehtuuri. Luettu: 2.10.2016.

Wikipedia 2016b. Boilerplate code. Luettavissa: https://en.wikipedia.org/wiki/Boilerplate_code. Luettu: 1.10.2016

Wikipedia 2016c. REST. Luettavissa: https://fi.wikipedia.org/wiki/REST. Luettu: 2.10.2016.

Wikipedia 2016d. HTTP. Luettavissa: https://fi.wikipedia.org/wiki/HTTP.

Luettu: 2.10.2016.

Wikipedia 2016e. Play Framework. Luettavissa: https://en.wikipedia.org/wiki/Play_Framework. Luettu: 18.9.2016.

Wikipedia 2016f. Convention over configuration. Luettavissa: https://en.wikipedia.org/wiki/Convention_over_configuration. Luettu: 25.9.2016.

Wikipedia 2016g. Web framework. Luettavissa: https://en.wikipedia.org/wiki/Web_framework. Luettu: 4.10.2016.

Woods, D. 2014. Exploring Micro-frameworks: Spring Boot. Luettavissa: https://www.infoq.com/articles/microframeworks1-spring-boot. Luettu: 2.9.2016.

Liitteet

Liite 1. Spring Boot -sovelluksen app.js.

```
$(function() {
	console.log ('connection established here. you may proceed.');

	//lisää asioita ostoslistalle ja näyttää lisätyn asian sivulla
	$("#add").click(function(){
		var item = $("#itemname").val();
		var amount = $("#itemamount").val();

		$.ajax({
			url: "/shoppinglist",
			contentType: "application/json",
			data: JSON.stringify({"name": item, "amount": amount}),
			type: "POST",
			dataType: "json"
		})
		.done(function(item) {
		 $("tbody").append(shoppingListRow(item));
		})
		.fail(function(error) {
			console.log(error); /
			/TODO: erilaisia käsittelyjä eri erroreille
		});
	});

	//sivun latautuessa haetaan kaikki tietokannassa olevat asiat näkyviin
	$.ajax({
		url: "/shoppinglist",
		contentType: "application/json",
		type: "GET",
		dataType: "json"
	})
	.done(function(itemlist) {
		itemlist.forEach(function(item) {
			$("tbody").append(shoppingListRow(item));
		})
	});

	function shoppingListRow (item) {
		return '<tr><td class="list">' + item.name +
		'</td><td  class="listamount">' + item.amount +
		'</td><td><button class="delete" data-id="' +
		item.id + '">X</button></td></tr>'
	}

	//poistetaan tietyn id:n omaava asia tietokannasta
	$("tbody").on("click", ".delete", function(event) {
		var dataid = $(event.target).attr("data-id");

		$.ajax({
			url: "/shoppinglist/" + dataid,
			type: "DELETE"
		})
		.done(function(id) {
			$(".delete[data-id=" + id + "]").closest("tr").remove();
		});
	});
});
```

Liite 2. Linkki Spring Boot -sovellukseen GitHubissa

https://github.com/katya450/SpringBootDemo

Liite 3. Linkki Play -sovellukseen GitHubissa

https://github.com/katya450/Play2Demo