Liikuntalajitietokannan suunnittelu ja toteutus mikropalveluna

Ammattikorkeakoulututkinnon opinnäytetyö

Visamäki, Tietojenkäsittelyn koulutusohjelma

Syksy, 2018

Antti Pohja

TIIVISTELMÄ

Tietojenkäsittelyn koulutusohjelma
Visamäki

Tekijä	Antti Pohja	**Vuosi** 2018
Työn nimi	Liikuntalajitietokannan suunnittelu ja toteutus mikropalveluna	
Työn ohjaajat	Lasse Seppänen, Juha Wiberg	

TIIVISTELMÄ

Tämän opinnäytetyön on tilannut Ambientia Oy. Ambientia järjestää jokavuotisen liikuntatapahtuman, joka on suunniteltu motivoimaan työntekijöitä työpäivän aikana tapahtuvaan liikuntaa. Tapahtumaa varten kehitetty sovellus on vanhentunut ja se kaipaa päivitystä. Opinnäytetyön lopputuloksena kehitetty mikropalvelu on osa tätä uudistusta.

Mikropalvelu tarjoaa rajapinnan, jonka kautta voidaan käsitellä tietokantaan tallennettuja liikuntalajeja ja liikuntatyyppejä. Ohjelmointikielenä on käytetty Javaa ja sen sovelluskehystä Spring Bootia. Mikropalvelu on suuniteltu toimimaan Ambientian ylläpitämällä OpenShift-alustalla. Rajapintaan tehdään kutsuja http-metodeilla, joiden avulla voidaan luoda, lukea, päivittää ja poistaa tietoja.

Opinnäytetyön lopputuloksena syntynyt sovellus sisältää kaikki määritellyt toiminnallisuudet. Työn tekijä oppi paljon mikropalveluista, rajapinnan kehittämisestä, sekä yleisesti työssä käytetyistä työkaluista. Sovelluksen jatkokehityskohteita ovat esimerkiksi hallintakäyttöliittymän toteuttaminen ja sovelluksen asetusten hallinnan kehittäminen OpenShift-ympäristössä.

Avainsanat Mikropalvelut, rajapinnat, Spring Boot, OpenShift

Sivut 42 sivua, joista liitteitä 5 sivua

Degree Programme in Business Information Technology
Visamäki

Author Antti Pohja **Year** 2018

Subject Developing a Database Mircoservice for Physical Exercises and Exercise Types

Supervisors Lasse Seppänen, Juha Wiberg

ABSTRACT

This thesis was commissioned by Ambientia Oy. Ambientia organizes a yearly exercise event for its employees. The goal of the event is to motivate the employees to perform physical exercises during the work day. An old web application that is used in the event needs updating. The result of this thesis is a microservice which is a part of the application upgrade.

The microservice offers an API to an exercise database. The new exercise application will be able to access the exercises and exercise types stored in the database through the API. The microservice is programmed with Java and its framework Spring Boot. The microservice is designed to be run on OpenShift which is run on Ambientia's premises. The API is accessed by using http-methods and it is used to create, read, update and delete exercises and exercise types.

The result of this thesis is an application that has all the required functionalities. The author gained a lot of knowledge about microservices, API development and all the tools that were utilized during the development. Further development ideas include a user interface for managing the exercises and exercise types and improving handling of application preferences in OpenShift environment.

Keywords Microservices, APIs, OpenShift, Spring Boot

Pages 42 pages including appendices 5 pages

SISÄLLYS

1 JOHDANTO 4

2 MIKROPALVELUT 5

2.1 Mikropalvelujärjestelmän edut 5
2.2 Vertailu muihin samankaltaisiin malleihin 6

3 OPENSHIFT 8

3.1 OpenShiftin versiot 8
3.2 Arkkitehtuuri 8
3.2.1 Docker 9
3.2.2 Kontit 10
3.2.3 Kubernetes 11

4 OHJELMOINNIN TYÖKALUT 12

4.1 Spring ja Spring Boot 12
4.2 PostgreSQL 13

5 PROJEKTI 15

5.1 Tietokannan suunnittelu 16
5.2 Sovelluksen käyttötapaukset 16
5.3 Sovelluksen rajapintojen suunnittelu 17
5.4 Sovelluksen rakenne 18

6 TOTEUTUSVAIHE 20

6.1 Java-sovellus 20
6.1.1 Spring Bootin käyttöönotto 20
6.1.2 Sovelluksen JPA-entiteetit 22
6.1.3 Entiteettien hallinta 23
6.1.4 Controller-luokka 24
6.1.5 Service-luokat 25
6.1.6 Sovelluksen rajapinta 26
6.2 Sovelluksen vienti OpenShiftiin 27
6.2.1 PostgreSQL OpenShiftissä 28
6.2.2 Java-sovellus OpenShiftissä 28

7 JOHTOPÄÄTÖKSET JA POHDINTA 32

7.1 Prosessin eteneminen 32
7.2 Tavoitteiden saavuttaminen ja opitut asiat 33
7.3 Saatu palaute 33
7.4 Jatkokehitys 33

LÄHTEET 35

LIITTEET 38

1 JOHDANTO

Opinäyteyön tilaajana toimii Ambientia Oy, joka on hämeenlinnalainen ohjelmistoyritys. Ambientia kehittää mm. Atlassian-, Liferay- ja verkkokauppatoteutuksia. Työntekijöitä on raportin kirjoitushetkellä noin 170.

Ambientialla on jokavuotinen henkilöstön liikuntahaaste, UFTC (Ultimate Functional Training Challenge). Liikuntahaaste motivoi henkilöstöä liikkumaan työpäivän aikana ja tuo esiin liikunnallista elämäntapaa. Haastetta varten on kehitetty websovellus, johon käyttäjät kirjautuvat ja tallentavat tekemiään liikuntasuoritteita. Sovellus on kehitetty aikoinaan hyvin nopeasti, eikä esimerkiksi käytettävyyttä ole juuri otettu huomioon. Käyttöliittymä ja taustateknologia ollaan uudistamassa modernimmaksi ja opinnäytetyö on osa uudistusta. Uusi UFTC-sovellus rakennetaan OpenShift-alustalle ja se on suunniteltu toimimaan mikropalveluarkkitehtuurin mukaisesti.

Tässä työssä kehitetty mikropalvelu tarjoaa rajapinnan, josta liikuntalajeja ja niitä kuvaavia liikuntatyyppejä voidaan hakea ja muokata. Työssä oli tarkoituksena selvittää, miten lajitietokannalle toteutetaan mikropalvelu ja millainen rakenne sille tulisi suunnitella. Yksi työn tavoitteista oli myös kehittää tekijän ammattitaitoa, sekä laajentaa yrityksessä tietämystä työssä käytetyistä teknologioista.

Liikuntalajitietokanta toteutettiin mikropalveluarkkitehtuurin mukaisesti, eli palvelun tarkoitus ja vastuualue on hyvin tarkasti rajattu. Tavoitteena oli, että sovelluksen tarjoamia palveluita pystytään hyödyntämään mahdollisimman laajasti mistä tahansa sovelluksesta REST-rajapinnan kautta. Mikropalvelumalli sopi hyvin opinnäytetyön aiheeksi, sillä yksittäisen palvelun tehtävä on tarkkaan rajattu.

Liikuntalajitietokannan kehittäminen oli tekijälleen mielenkiintoista ja tarjosi mahdollisuuden oppia paljon uutta Java-ohjemoinnista, mikropalveluista, rajapinnoista ja OpenShift-alustasta. Opinnäytetyön myötä uusi UFTC-sovellus on askeleen lähempänä toteutumista.

2 MIKROPALVELUT

Mikropalvelut ovat kehittyneet monen ohjelmistokehitykseen liittyvän toimintamallin ja teknologian seurauksena. Näitä ovat esimerkiksi Domain-driven design, jatkuva julkaisu (Continuous delivery), skaalattavat virtualisointialustat ja autonomiset tiimit. (Newman 2015, 1.) Mikropalvelut ovat nimensä mukaisesti pieniä kokonaisuuksia ja keskittyvät tekemään yhtä asiaa hyvin. (Newman 2015, 2.)

Mikropalvelu on autonominen yksikkönsä, kuten esimerkiksi jokin käyttöjärjestelmän prosesseista. Ne kommunikoivat toistensa kanssa verkkokutsujen välityksellä, jotta ne on helppo pitää irrallisena toisistaan. Mikropalveluilla on jonkinlainen API, jonka kautta tiedonsiirto tapahtuu toisiin palveluihin ja käyttäjälle. Kultaisena sääntönä mikropalvelua kehitettäessä voidaan pitää, että muutos palvelussa ei saisi aiheuttaa muutostarpeita minnekään muualle. (Newman 2015, 3.)

2.1 Mikropalvelujärjestelmän edut

Mikropalveluarkkitehtuurin hyödyksi on havaittu nopeampi sovellusten julkaisutahti ja uusien teknologioiden omaksuminen. Mikropalveluarkkitehtuurissa uuden teknologian kuten ohjelmointikielen tai tietokannan käyttöönotto voidaan tehdä pienessä mittakaavassa. Kun järjestelmä koostuu lukuisista mikropalveluista, muutoksen voi tehdä jossain vähemmän tärkeässä palvelussa, jolloin mahdolliset haittavaikutukset jäävät pieneksi. Monoliittiarkkitehtuurissa vastaava muutos olisi massiivinen rupeama. (Newman 2015, 4.)

Mikropalvelujärjestelmien vikasietokyky on oikein rakennettuna suurempi, kuin perinteisempien järjestelmien. Jos monoliittijärjestelmässä tapahtuu virhe, usein se rikkoo koko järjestelmän. Mikropalveluissa voidaan virhetilanteessa poistaa virhetilassa oleva palvelu pois käytöstä. Suunnitteluvaiheessa täytyy kuitenkin tarkoin miettiä millaisia virhetilanteita voi syntyä, miten ne käsitellään ja miten ne vaikuttavat loppukäyttäjään. (Newman 2015, 5.)

Järjestelmän resurssien skaalaus voidaan tehdä mikropalvelukohtaisesti, jolloin paljon resursseja tarvitseville mikropalveluille voidaan määritellä enemmän resursseja (Newman 2015, 5 - 6). Monoliittien skaalaus on myös mahdollista, mutta vaatii useita palvelimia, joilla ajetaan samaa sovellusta. Sovellukset toimivat klusterissa, ja todennäköisesti myös niiden

tarvitsema tietokanta on klusteroitu. Monoliittiklusterit ovat hankalia ylläpidettäviä ja päivitettäviä, joten sovellusten pilkkominen pieniin osiin helpottaa niiden pitämistä ajan tasalla. (Ramgir & Samoylov 2018, 99-100.)

Mikropalvelujärjestelmässä muutosten käyttöönotto on riskittömämpää, kuin monoliittijärjestelmissä. Monoliittijärjestelmässä koko sovelluksen käyttöönotto joudutaan suorittamaan alusta, jotta pienikin muutos saadaan tehtyä. Kun muutoksia kasaantuu useampia, voivat virheen negatiiviset vaikutukset olla huomattavia. Mikropalveluissa virhe on helpompi kohdistaa tiettyyn palveluun. Tällöin voidaan palata nopeasti ja helposti aiempaan versioon. (Newman 2015, 6.)

Mikropalvelujärjestelmiä kehitettäessä työ on helpompi jakaa kehittäjien ja ryhmien kesken (Newman 2015, 7). Kehittäjien rekrytointi ja heidän tehtäviensa jakaminen on helpompaa, sillä teknologia ei ole mikropalveluarkkitehtuurissa enää rajoitteena samalla tavalla. Eri teknologiat toimivat rajapintojen välityksellä yhdessä. (Ramgir & Samoylov 2018, 100.)

Hajautettujen järjestelmien ja palvelukeskeisen arkkitehtuurin yhtenä pohja-ajatuksena on toiminnallisuuksien käyttö vapaasti rajapintojen kautta. Mikropalveluiden ajatuksena on myös tarjota toiminnallisuuksiaan vapaasti käyttöön eri tarkoituksiin. (Newman 2015, 7.)

Ramgir & Samoylov (2018, 101) ohjeistavat kirjassaan mikropalvelun kehittämistä siten, että sen lähdekoodin pitäisi olla pienempi, kuin palvelukeskeisen arkkitehtuurin sovelluksen lähdekoodin. Mikropalvelu täytyy voida ottaa myös käyttöön itsenäisesti ja mikropalveluilla on useimmiten omat tietokantansa. Mikropalvelun täytyy olla tilaton ja samojen toiminnallisuuksien täytyy palauttaa aina sama tulos. Lisäksi pitäisi pystyä tarkistamaan palvelun tila tarvittaessa.

Mikropalvelujärjestelmän osan korvaaminen paremmin toimivalla versiolla on paljon pienempi urakka, kuin vastaavassa monoliittijärjestelmässä saman muutoksen tekeminen olisi. Muutos on helpompi suunnitellla ja toteuttaa. Vanhassa ja suuressa järjestelmässä muutos jäisi todennäköisesti tekemättä, koska se olisi liian riskialtista ja työlästä. (Newman 2015, 7.)

2.2 Vertailu muihin samankaltaisiin malleihin

Monet mikropalveluarkkitehtuurin eduista ovat saavutettavissa myös muilla menetelmillä. Mikropalvelujärjestelmän edut syntyvät ensisijaisesti siitä, että se koostuu pienistä osasista, mikä mahdollistaa lisää tapoja ratkaista ongelmia. (Newman 2015, 9.)

Palvelukeskeisessä arkkitehtuurissa useat eri palvelut toimivat yhdessä tuottaakseen erilaisia toiminnallisuuksia. Ne kommunikoivat verkon välityksellä prosessin sisäisten metodikutsujen sijaan. Arkkitehtuurin pohja-ajatus on sinänsä pätevä ja monet sen ongelmista ovat johdettavissa muihin asioihin. Esimerkiksi viestintäprotokollien, ulkoisten väliohjelmistojen, huonosti ohjeistetun palvelun pilkkomisen tai niiden väärän pilkkomisen ongelmat mielletään usein palvelukeskeisen arkkitehtuurin ongelmiksi. (Newman 2015, 8.)

Palvelukeskeisessä arkkitehtuurissa ei selkeästi määritellä, miten esimerkiksi suuri järjestelmä pilkotaan pienempiin osiin. Yhteys todelliseen maailmaan jää usein puutteelliseksi ja palvelut naitetaan liian tiukasti toisiinsa yhteen. (Newman 2015, 8.) Mikropalveluita voidaankin ajatella palvelukeskeisen arkkitehtuurin muotona, joka on syntynyt tosimaailman käyttötapausten pohjalta. (Newman 2015, 9.)

Sovelluksissa varsin yleinen menetelmä on jaettujen kirjastojen käyttäminen. Kirjastojen avulla voidaan jakaa toiminnallisuuksia kehitystiimien ja palveluiden kesken. Tällaisissa kirjastoissa on kuitenkin rajoitteita verrattuna mikropalveluihin. Näitä ovat usein esimerkiksi saman ohjelmointikielen käyttövaatimus ja samalla alustalla toimiminen. Kirjaston käyttöönotossa joudutaan koko sovellus käynnistämään uudelleen. Jaettuja kirjastoja kannattaa kuitenkin käyttää palveluissa, jotta voidaan hyödyntää kerran tehtyä koodia. (Newman 2015, 9.)

Monet ohjelmointikielet mahdollistavat moduulien käytön. Moduulit ovat jaettuja kirjastoja monipuolisempia siinä mielessä, että niitä voidaan ottaa käyttöön ja muuttaa prosessin ollessa päällä. Moduulit periaatteessa mahdollistavat hajautetut palvelut yhden monoliittisen prosessin sisällä, mutta usein ne ovat liian tiukasti naitettuja muuhun ohjelmistoon. (Newman 2015, 10.)

Mikropalveluarkkitehtuuri tarjoaa paljon etuja, mutta se ei silti ole ratkaisu kaikkiin ongelmiin. Hajautettujen järjestelmien yleiset ongelmakohdat pätevät myös mikropalvelujärjestelmiin. Vaatii myös uusien ajattelutapojen omaksumista siirryttäessä muista arkkitehtuureista. Yrityksen toimintaympäristö ja sovelluksen käyttötarkoitus asettavat omat vaatimuksensa mikropalveluarkkitehtuurin hyödyntämistä pohdittaessa. (Newman 2015, 11.)

3 OPENSHIFT

OpenShift on avoimen lähdekoodin alustapilvi, eli siellä sovellukset ja palvelut pyörivät konteissa, joille voidaan määritellä resursseja tarpeen mukaan, sekä skaalata niitä kuormituksen mukaan. OpenShift on Red Hatin tuote ja Red Hat hallinnoi alustan kehitystä. OpenShift on ollut saatavilla vuodesta 2011 ja siitä on tullut suosittu alusta verkkosovelluksille ja -palveluille. OpenShiftin uusin 3. versio pohjautuu Docker-kontteihin ja Kubernetes-teknologiaan. (Shipley & Dumpleton 2016, ix.)

OpenShiftiin voi luoda esimerkiksi websovelluksia ja niiden taustapalveluita. Ohjelmointikieliä ei rajoiteta ja ainoa vaatimus onkin, että sovellus toimii kontissa. (Dumpleton 2018, 1.)

3.1 OpenShiftin versiot

OpenShiftistä on tällä hetkellä saatavilla neljä versiota: Online, Dedicated sekä Container Platform. OpenShift Online on Red Hatin omassa pilvessä toimiva palvelu, josta on saatavilla ilmainen Starter Plan, jossa on rajoituksia esimerkiksi resursseissa ja tuessa. Se on suunnattu yksityiseen opiskeluun, kokeiluihin sekä kehitykseen. Pro plan on maksullinen ja suunnattu ammattimaiseen toimintaan. Sen hinta riippuu tarvittavista resursseista. (Red Hat 2018a.) OpenShift Dedicated on korkean saatavuuden palvelu, jossa koko klusteri on yhdelle asiakkaalle dedikoitu ja Red Hatin ylläpitämä. Palvelu hostataan Amazonin tai Googlen pilvipalvelualustoilla. (Red Hat 2018b). OpenShift Container Platform tarjoaa asiakkaalle mahdollisuuden ajaa OpenShiftiä itse hallinnoimassaan ympäristössä Red Hatin tukipalveluiden piirissä. (Red Hat 2018c.)

3.2 Arkkitehtuuri

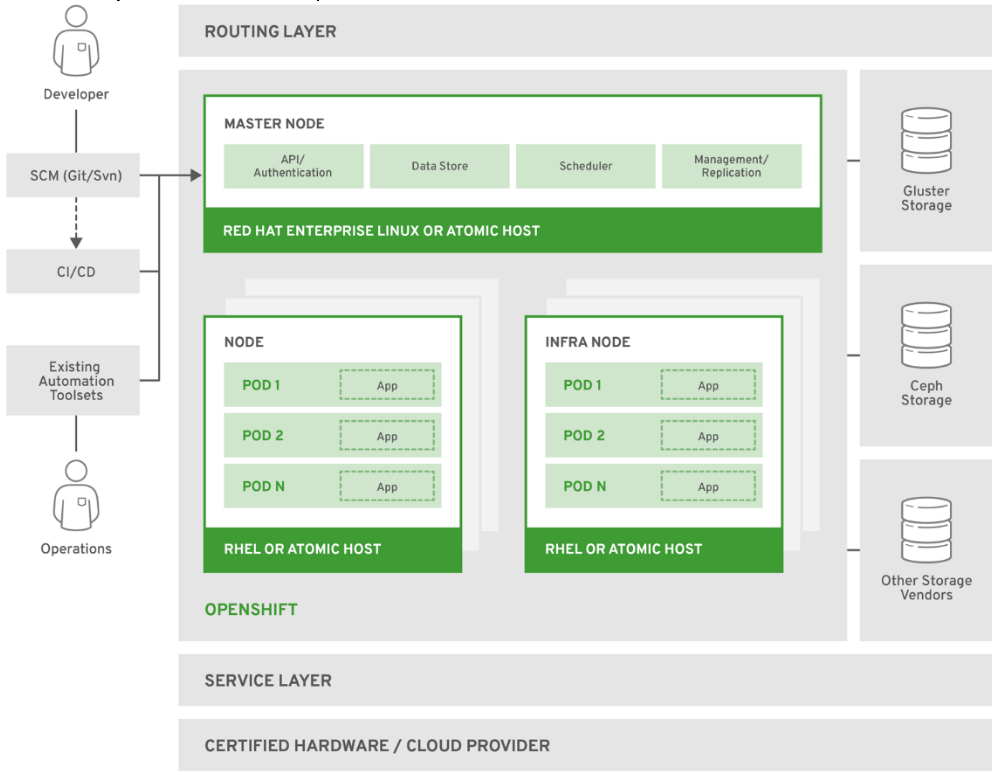
Kuvassa 1 on kuvattu OpenShiftin yleinen arkkitehtuuri. Vihreällä reunustetut elementit kuuluvat Kubernetes-klusteriin. Kubernetes-klusteri muodostaa OpenShiftin ytimen ja OpenShift lisää sen ympärille järjestelmän, joka tarjoaa sovellusten ja koko alustan ylläpitoa ja hallintaa helpottavia ominaisuuksia.

Esimerkiksi OpenShiftissä sovelluksen lähdekoodista voi luoda ja kustomoida erilaisia koontiversioita. Lähdekoodien koontiversiot paketoidaan OpenShiftissä sovelluskuviksi, jotka voi ottaa käyttöön podeissa (Red Hat 2018d.) Podissa toimivalle sovellukselle voidaan tarpeen mukaan skaalata

uusia podeja, jolloin sovelluksen käytössä olevat resurssit kasvavat. Lisäksi OpenShift tarjoaa kehitystiimien ja käyttäjien hallinnan. Kuvan 1 pohjalla näkyy Service layer, joka merkitsee verkkoinfrastruktuuria, eli laitteistoa tai pilveä, jossa OpenShift toimii.

Kubernetes-klustereissa on aina vähintään yksi Master-komponentti, joka orkestroi kuvassa 1 sen alla esitettyjen muiden noodien toimintaa, sekä vuorottaa podien suoritusta noodeissa. (Red Hat 2018e.) Kubernetesista ja klusterista kerrotaan lisää luvussa 3.2.3.

Kuvan 1 yläreunassa kuvattu routing layer tarjoaa sovelluksille yhteydet OpenShiftin ulkopuolelle. Alareunan service layer puolestaan on fyysinen infrastruktuuri, jossa OpenShift toimii. Kehittäjälle OpenShift mahdollistaa jatkuvat sovelluspäivitykset ja ylläpidolle puolestaan työkalut alustan hallintaan. (Red Hat 2018f.)



Kuva 1. OpenShiftin arkkitehtuurin yleiskatsaus (Red Hat n.d.).

OpenShift Origin on vapaasti saatavilla ja sille ei tarjota hostauspalvelua, vaan käyttäjän on itse hankittava tarvittavat palvelinresurssit. RedHatin kaupalliset tuotteet pohjautuvat Originiin ja Origin saa uusimmat päivitykset ensimmäisenä. (Red Hat 2018g.)

3.2.1 Docker

Docker on avoimen lähdekoodin työkalu, joka mahdollistaa sovelluksen tai palvelun paketoimisen sovelluskuvaksi. Sovelluskuva voidaan suorittaa

isäntäjärjestelmässä kontissa, eli voidaan sanoa, että kontti on sovelluskuvan instanssi. Docker eroaa virtuaalikoneista siten, että Docker-kontti ei sisällä käyttöjärjestelmää, vaan isäntäjärjestelmän käyttöjärjestelmään tuodaan kontissa sovelluksen tarvitsemat riippuvaisuudet. Tämä menetelmä pienentää paketin kokoa ja suorittaminen nopeutuu huomattavasti. (Opensource.com 2018.)

Docker sisältää kaksi selvästi erillistä komponenttia: Docker Enginen ja Docker Hubin. Docker Enginen avulla voidaan nopeasti ja kätevästi ajaa kontteja. Docker Hub puolestaan tarjoaa julkisia konttikuvia ladattavaksi, joita kehittäjät voivat hyödyntää säästääkseen aikaa ja vaivaa. (Mouat 2015.)

OpenShiftissä voi suorittaa mitä tahansa Docker-sovelluskuvia tai Dockerfilen perusteella luoda uusia uusia sovelluskuvia (Docker Hub 2018). Dockerfile on tekstitiedosto, jossa on Dockerin komentorivin komentoja, jotka suoritetaan sovelluskuvaa rakennettaessa (Docker 2018). OpenShift tarjoaa myös Source-to-Image (S2I) ominaisuuden, jossa uusi sovelluskuva luodaan versionhallinnasta noudetun lähdekoodin ja valmiin sovelluskuvan yhdistelmästä (Docker Hub 2018). S2I-kuvalle on tiettyjä vaatimuksia ja valmiita kuvia on saatavilla esim. Node.js, Python ja PHP -sovelluksille (OKD, 2018). Kuvassa 1 on vihreällä rajattuna esitetty noodit, sekä niiden sisälllä podit, jossa Docker-kontit toimivat.

3.2.2 Kontit

Konttiteknologia on ollut jo pitkään olemassa, ja ensimmäinen sen täydelliseen hallintaan keskittynyt projekti oli Linux Containers (LXC), joka sai alkunsa vuonna 2008 (Mouat 2015). Se teki konteista helpommin lähestyttäviä. Konttien käyttöönotto ja sovellusten ajaminen konteissa saattoi kuitenkin vaatia suurtakin vaivannäköä (Dumpleton 2018, 2). Vuonna 2013 Docker teki konteista valtavirtaa tuomalla mukaan helposti levitettävät järjestelmäkuvat ja hyvän käyttöliittymän.
Kontti sisältää sovelluksen ja kaikki sen riippuvaisuudet (esim. käyttöjärjestelmän kirjastot ja sovellukset). OpenShift on suunniteltu konttien kehittämiseen ja käyttöönottoon (Shipley & Dumpleton 2016, 1).

Konttien lähtökohta on, että kehittäjä voi käyttää valitsemiaan työkaluja ja olla silti varma, että sovellus toimii kontin ansiosta oikein missä tahansa ympäristössä. Ympäristön ylläpidossa puolestaan voidaan keskittyä verkon toimivuuteen, resursseihin sekä palvelun saavutettavuuteen, sen sijaan, että tarvitsisi huolehtia esimerkiksi sovelluksen ympäristön konfiguroinnista. (Mouat 2015.)

Konteilla on lukuisia etuja perinteisiin virtuaalikoneisiin nähden. Kontin voi käynnistää lähes välittömästi ja se käyttää suoraan isäntäjärjestelmän resursseja ja kontit poistavat suoritusympäristön vaihtelusta aiheutuvia

ongelmatilanteita. Ne eivät myöskään vaadi laitteistolta runsaasti resursseja, jolloin yksittäinen kehittäjä voi omassa järjestelmässään emuloida hajautettua järjestelmää. Kontti mahdollistaa sovelluksen levittämisen ja helpon asentamisen toisiin järjestelmiin, eikä näistä asioista tarvitse kehitysvaiheessa siten huolehtia. (Mouat 2015.)

3.2.3 Kubernetes

Kubernetes on Googlen kehittämä järjestelmä, joka huolehtii konteissa sijaitsevien palveluiden resursoinnista, kuten verkkoyhteyksistä ja levytilasta. Kubernetes tarjoaa mahdollisuudet myös palveluiden konfigurointiin ja automatisointiin. Kubernetes syntyi, kun sitä edeltävä projekti siirrettiiin avoimeen lähdekoodiin vuonna 2014. (The Linux Foundation 2018.) OpenShift käyttää Kubernetesiä konttien orkestrointiin, eli Kubernetes tarjoaa mekanismit niiden käyttöönottoon, ylläpitoon ja skaalaukseen (Red Hat 2018h).

Kubernetes toimii klustereissa ja niissä on yksi tai useampia master-komponentteja, joiden alla puolestaan toimii noodeja. Noodi sisältää podeja, joissa konttiin pakatut sovellukset toimivat. Master-komponentti hallitsee klusterin noodien aikataulutusta ja huolehtii uusien podien käynnistymisestä. Noodien sisällä on järjestelmä, joka ylläpitää siellä pyöriviä podeja ja huolehtii että kontit puolestaan pyörivät podeissa. Kubernetes tukee useita konttijärjestelmiä, esimerkiksi Dockeria. (The Linux Foundation 2018a.)

Master sisältää API-palvelimen, joka validoi ja konfiguroi dataa podeille, palveluille ynnä muille rajapintaa käyttäville kohteille. Masterin komponenteista etcd persistoi masterin tilan, johon muut komponentit pyrkivät seuraamalla etcd:tä. Controller Manager Server seuraa etcd:tä komponenttien replikointiin tulleiden muutosten varalta ja APIn kautta pakottaa tilan muutokset voimaan. (The Linux Foundation 2018b.)

4 OHJELMOINNIN TYÖKALUT

Luvussa esitellään Java-sovelluksessa käytetty Spring-sovelluskehys, ja tarkemmin Spring Boot. Lisäksi sovellus on läheisessä kytköksessä tässä luvussa esiteltyyn PostgreSQL-tietokantaan. Työkaluihin on otettu mukaan myös Twelve-Factor App-metodologia, joka on toiminut löyhänä ohjenuorana sovelluskehityksessä.

4.1 Spring ja Spring Boot

Spring on ohjelmistokehys, jonka osa Spring Boot on. Spring on alun perin kehitetty vaihtoehdoksi Java Enterprise Editionille (Java EE). (Walls 2016, 1-2.) Spring sai alkunsa vuonna 2002, kun Rod Johnson teki kirjan Expert One-on-One J2EE Design and Development, jota varten Johnson kehitti myös ohjelmistokehyksen. Aluksi ohjelmistokehyksen nimi oli Interface21. Kaksi kirjan lukijaa, Juergen Holler ja Yann Caroff, yllyttivät Johnsonia tekemään ohjelmistokehyksestä avoimen lähdekoodin projektin. Vuoden 2002 lopulla ohjelmistokehyksen nimeksi valikoitui Yannin ehdotuksesta Spring. Nimi kuvasi ohjelmistokehyksen uudistavaa ja raikastavaa vaikutusta Java EE:lle. (Johnson 2006.)

Springin olennainen ajatus on riippuvuuksien injektointi, dependency injection ja ohjelmoinnin keskittyminen bisneslogiikkaan, eli siihen mitä sovelluksella halutaan tehdä, ja siirtyminen pois taustajärjestelmien vaatimasta logiikasta. Java EE:n Enterprise JavaBeanien toiminnallisuus voidaan saavuttaa perinteisillä Javan olioilla, joka on kevyempi ratkaisu. (Walls 2016, 2.)

Alun perin riippuvuuksien injektointi toteutettiin Springissä XML-tiedostoissa. Ohjelmointi yksinkertaistui, mutta sen oheen tuli huomattava määrä XML-muotoon kirjoitettavaa konfiguraatiota, jossa määriteltiin riippuvuudet luokkien välillä. (Walls 2016, 2.)

Spring 2.5 toi helpotusta konfiguraation kirjoittamiseen Java-annotaatioiden muodossa. Yksinkertaisilla annotaatioilla saatiin aikaan se, että paljosta XML-konfiguroinnista voitiin luopua. Spring 3.0 puolestaan toi mukaan Java-pohjaisen konfiguraation, eli konfiguraatio voidaan kirjoittaa Java-koodina, mikä teki konfiguroinnista sujuvampaa. (Walls 2016, 2.)

Spring Boot sisältää lisää helpotuksia sovelluskehittäjille. Se sisältää automaattisen konfiguroinnin monien Spring-sovellusten toiminnallisuuksille.

Jos classpathiin lisää H2-tietokannan kirjaston, Spring Boot konfiguroi tietokannan automaattisesti käyttöön ja kirjaston Beanit ovat valmiin injektoitavaksi sovelluksen Beaneissa. (Walls 2016, 4-5.)

Spring Boot hyödyntää starter-riippuvuuksia, jotka sisältävät kattavan määrän kirjastoja. Riippuvuuksien hallinta yksinkertaistuu, kun saman tyyppisisssä sovelluksissa usein käytetyt riippuvuudet sisältyvät starter-paketteihin. Starter-riippuvuudet on myös nimetty sovelluksen tarpeen mukaan, esimerkiksi *web, security* tai *jpa*. Näin kehittäjän on helppo valita oikea starter-riippuvuus omien tarpeidensa mukaan. (Walls 2016, 5).

Näiden lisäksi Spring Boot tarjoaa komentoliittymän, jolloin Groovy-skriptejä voi suorittaa suoraan komentoriviltä ilman sovelluksen kääntämistä. Komentoliittymä pystyy hakemaan yleisimmät riippuvuudet automaattisesti starter-pakettien avulla ilman import rivejä, sillä se tunnistaa skriptissä käytetyt luokat. (Walls 2016, 6).

Viimeinen Sprin Bootin lisäämä osanen on Actuator, jonka avulla voi sovelluksen käydessä tarkkailla sen ominaisuuksia, kuten käytössä olevat Beanit, autokonfiguraation tila, ympäristömuuttujat, HTTP-kutsut ja niin edelleen. Actuator mahdollistaa ssh-yhteyden ottamisen sovellukseen, jolloin sovellukselle voi antaa komentoja sen tilan tutkimiseksi. (Walls 2016, 6-7).

Spring Boot ei sisällä itsessään mitään ominaisuuksia, kuten JPA:ta tai palvelinta. Spring Bootin avulla niitä voi kuitenkin hyödyntää riippuvuuksien ja automaattisen konfiguroinnin kautta saumattomasti. Spring Boot siis helpottaa Springin käyttöä automatisoimalla konfigurointia, joka olisi muuten kehittäjän tehtävä. (Walls 2016, 7).

4.2 PostgreSQL

PostgreSQL on oliopohjainen relaatiotietokantajärjestelmä, joka käyttää SQL-kyselykieltä. Se sai alkunsa Kalifornian yliopistossa Berkeleyssä POSTGRES-projektina, jota johti Michael Stonebraker. POSTGRES kehitys alkoi vuonna 1986. (The PostgreSQL Global Development Group 2018a.)

Vuonna 1994 Andrew Yu ja Jolly Chen julkaisivat POSTGRESin pohjalta avoimen lähdekoodin projektin nimeltä Postgres95. Postgres95 lisäsi järjestelmän tehokkuutta ja helpotti ylläpitoa. Suuria muutoksia olivat muun muassa PostQUEL-kyselykielen korvaaminen SQL-kielellä ja psql-sovelluksen käyttöönotto kyselyjen suorittamiseen komentoriviltä. Postgres95 nimi muutettiin vuonna 1996, ja nimeksi tuli PostgreSQL, joka toi esiin sekä nimen historian, että tuen yleiselle SQL-kielelle.
(The PostgreSQL Global Development Group 2018b.)

PostgreSQL tukee esimerkiksi kompleksisia kyselyitä, viitevaimia sekä näkymiä. Myös käyttäjäkohtaiset laajennukset kuten uusien datatyyppien ja funktioiden lisääminen, indeksointimenetelmät sekä proseduraalisten ohjelmointikielien käyttö ovat mahdollisia. (The PostgreSQL Global Development Group 2018a.) PostgreSQL pyrkii noudattamaan SQL-standardia, joskin jotkin ominaisuudet tai arkkitehtuurin vaatimat ratkaisut voivat poiketa standardista (The PostgreSQL Global Development Group 2018c.)

Kirjoitushetkellä PostgreSQL on neljänneksi suosituin tietokantajärjestelmä DB-Engines -sivuston mukaan ja ainoa neljästä ensimmäisestä, joka on kasvattanut suosiotaan viimeisen vuoden aikana. Lista kerätään hakukoneiden avulla etsien mainintoja, käytetyimpiä hakusanoja, työpaikkailmoituksia ja esimerkiksi twiittejä aiheeseen liittyen. (Solid IT 2018.)

5 PROJEKTI

Opinnäytetyön toimeksiantajalla on käytössään vanha ja nopeasti kehitetty liikuntaharjoitesovellus. Sitä käytetään jokavuotisessa UFTC-rupeamassa. UFTC tulee sanoista Ultimate Functional Training Challence ja osallistujat kirjaavat tekemänsä harjoitteet sovellukseen. Sovelluksen käytettävyys ja ylläpidettävyys vaativat päivittämistä. Sovelluksen tarkoitus on motivoida yrityksen henkilöstöä liikkumaan työpäivän aikana.

Kyseessä ei ole yritykselle kriittinen sovellus, mutta sen kehittäminen on henkilöstön hyvinvointia tukevaa toimintaa. Sovellusta ollaan uudistamassa modernimmaksi ja opinnäytetyön tuotteena syntyvä lajitietokanta on osa uudistusta. Projektin tavoitteena on ollut myös kehittää työn tekijän ammattitaitoa. Projektissa käytetyt teknologiat ovat työn tilaajalla laajasti käytössä, joten tekijän kokemuksen lisääminen hyödyttää organisaatiota laajemminkin.

Lajitietokannan tarkoitus on hyvin rajattu. Sen ajatus on tarjota UFTC-sovellukselle rajapinta, josta voidaan hakea lajit, joita tarvitaan. Lajien lisäksi lajitietokanta sisältää lajien tyypit. Esimerkiksi työmatkapyöräily voi sisältää tyypitykset ulkoilma, aerobinen ja saavutettava. Vaikka lajitietokannan ensisijainen tarkoitus on palvella UFTC-sovellusta, niin mikropalvelumallin perusajatuksen mukaisesti sitä voidaan käyttää muistakin sovelluksista. Esimerkiksi jos joku haluaa käyttää lajilistaa jossain täysin toisessa sovelluksessa, on se saatavilla rajapintaa hyödyntämällä.

OpenShift tarjoaa erinomaiset puitteet mikropalveluille. Samassa OpenShift-ympäristössä toimivien sovellusten on helppo hyödyntää toistensa rajapintoja tai muita palveluita. Sovelluksien palvelut voidaan avata myös ympäristön ulkopuolelle. Alusta huolehtii palveluiden välisestä verkkoliikenteestä, sekä niiden tarvitsemista resursseista.

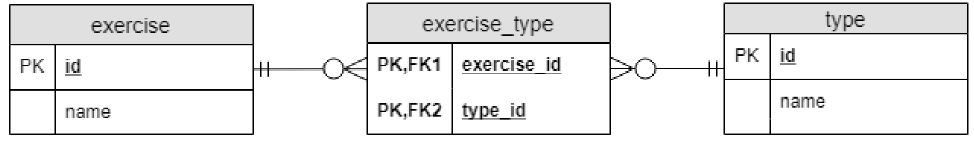
Projektin idean selkiydyttyä varsinaisessa suunnittelussa keskityttiin arkkitehtuurin valintaan, sekä siihen mitä kaikkea valittu arkkitehtuuri vaatii. Mikropalveluarkkitehtuuri valittiin sen ajankohtaisuuden takia ja työn tilaajalla on käytössään teknologioita, jotka tukevat arkkitehtuurin valintaa. Opinnäytetyön kannalta mikropalveluarkkitehtuurin eduksi katsottiin myös aiheen rajaamisen helppous ja luonnollisuus. Suunnittelmavaiheessa on mietitty alustavasti tietokannan rakenne, sovelluksen käyttötapaukset sekä sovellusrajapinta.

5.1 Tietokannan suunnittelu

Liikuntalajien tallentaminen tietokantaan on yksi UFTC-sovelluksen tarpeista. Esiin nousseista kysymyksistä suurin oli mitä kaikkea tietokantaan okeasti tarvitsee tallentaa. Aluksi suunnittelin kantaa siten, että sinne tallennetaan lajien nimet, niistä saatavat pisteet, kuvaukset ja ylipäänsä kaikki mahdollinen, mitä tietoa sovelluksen täytyy saada liikuntalajeihin liittyen. Mikropalveluarkkitehtuuri kuitenkin ohjasi tätä alun käsitystä rajatumpaan suuntaan: Liikuntalajitietokannan ei tarvitse käsittää mitään muuta, kuin lista lajeista, sekä mahdollisesti niihin liittyvää metatietoa.

Lajitietokanta päätettiin rakentaa siten, että siellä on kaksi taulua. Yksi taulu sisältää listan liikuntalajeista ja toinen sisältää lajeihin liitettävää metatietoa. Metatietotaulu voidaan käsittää tagivalikoimana, joita voidaan liittää liikuntalajeihin tarpeen mukaan. Metatietotaulu sisältää liikuntalajeja kuvaavaa tietoa, esimerkiksi: *ulkoliikunta*, *sisäliikunta, työmatkaliikunta*, *kevyt*, *raskas*, *joukkueliikunta* ja niin edelleen.

Liikuntalajilla voi olla monta tagia, ja puolestaan tagi voi olla liitetty moneen liikuntalajiin. Tällöin kahden taulun välille syntyy monesta moneen -yhteys, jolloin tietokantaan tarvitaan yhteyden luomista varten vielä kolmas taulu. Lajiin ei välttämättä kuitenkaan tarvitse liitää metatietoa ja metatieto voi olla olemassa ilman, että sitä on liitetty yhteenkään lajiin, jolloin kyseessä ei ole välttämätön yhteys. Kuvassa 2 on esitetty tietokannan rakenne.



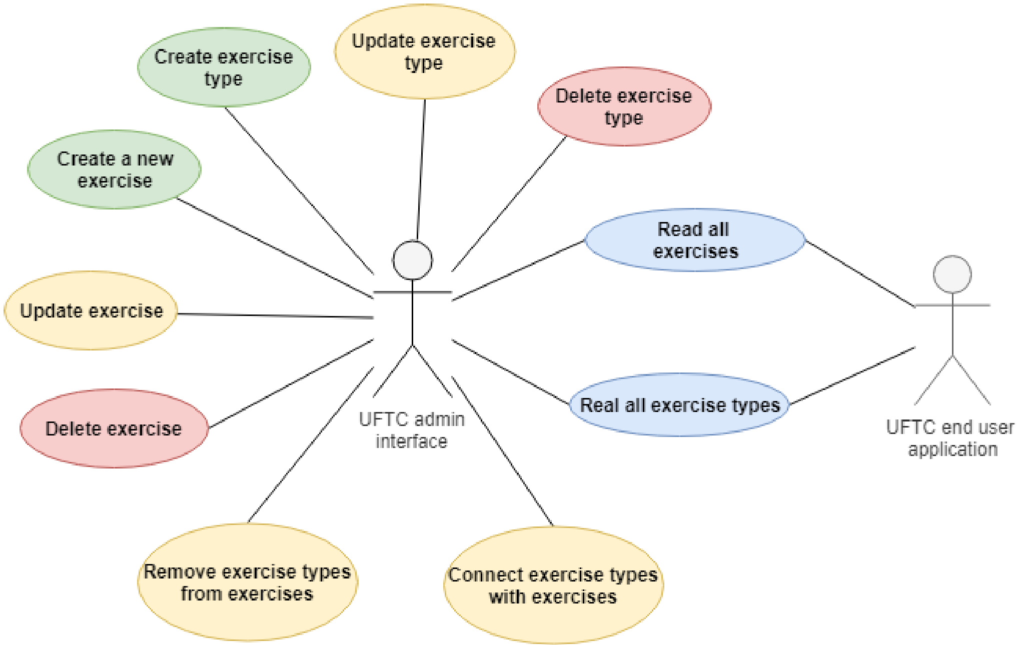
Kuva 2. Tietokannan rakenne.

Sovelluksen ohjelmoinnissa käytetty Hibernate-ohjelmistokehys osaa luoda sovelluksen tarvitsemat tietokantataulut sovelluksessa määritettyjen entiteettien ja niiden yhteyksien perusteella. Tästä johtuen tietokannan käyttöönotossa tarvitsee vain asentaa tietokantaohjelmisto ja luoda sinne tietokanta, johon sovelluksella on käyttöoikeus. Kehittäjän ei tarvitse tietokantaan tehdä muutoksia manuaalisesti, vaan sovellus osaa itse tehdä tarvittavat toimenpiteet tietokannassa. Hibernate on JPA:n implementaatio.

5.2 Sovelluksen käyttötapaukset

Käyttötapauksia suunniteltaessa pohdinta kohdistetiin siihen, millaisia tarpeita UFTC-sovelluksella on tietokantapalvelua kohtaan. UFTC-

sovelluksen tarpeet tulisivat kohdistumaan todennäköisimmin lajitieto-kannan ylläpitoon, sekä lajien ja metatietojen välittämiseen loppukäyttä-jälle. Näistä saatiin muodostettua UML-malli käyttötapauksista (Kuva 3). Käyttötapaukset on ajateltu sovelluksen todennäköisten näkymien ja siellä suoritettavien toimenpiteiden näkökulmasta.



Kuva 3. Liikuntalajitietokannan käyttötapaukset.

Käyttötapauksista valtaosa keskittyy lajitietokannan hallintaan. Sovelluksen täytyy tarjota mahdollisuus luoda, muokata ja poistaa liikuntalajeja. Lisäksi lajien tyypeille tarvitaan samat mahdollisuudet. Lajien ja tyyppien yhdistäminen olisi voitu jättää lopullisen UFTC-sovelluksen huoleksi, mutta siitä päätettiin tehdä lajitietokannan ominaisuus.

Liikuntalajien ja metatiedon yhteys on todennäköisesti muuttumaton riippumatta niitä hyödyntävästä sovelluksesta. Tyypit ovat myös selvästi liikuntalajeja kuvaavaa aineistoa, jolloin on loogista ja perusteltua sisällyttää myös lajien ja tyyppien yhteydet tietokantapalveluun. Sen sijaan esimerkiksi liikuntalajista saatava pistemäärä ei ole liikuntalajia kuvaavaa informaatiota, joten se jätettiin lajitietokannasta pois.

5.3 Sovelluksen rajapintojen suunnittelu

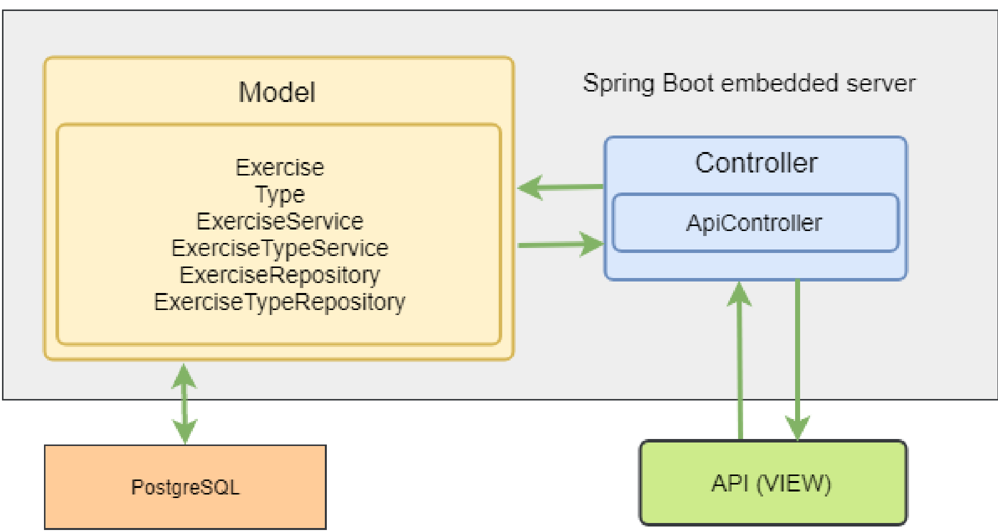
Aiheenvalinnan selkiydyttyä jo vaiheessa päätettiin, että mikropalveluille tyypillisesti lajitietokannan yhteys sovelluksen ulkopuolelle toimii REST-rajapinnan välityksellä. Käyttötapausten perusteella rajapintoja tarvittiin lajien ja tyyppien luontiin, muokkaukseen ja poistoon, sekä vielä lajien ja tyyppien yhteyksien luomiseen ja poistoon.

Lähtökohtaisesti lajien hallintaan tarjotaan yksi osoite ja tyyppien hallintaan toinen osoite. Sovelluksen juuriosoitteen perään muodostetaan polku haluttuun resurssiin. Esimerkiksi liikuntalajien tapauksessa juuriosoitteen jälkeen tuleva osa on */exercises*. Jos puolestaan halutaan käsitellä yhtä tiettyä liikuntalajia, polun perään laitetaan lajin tunniste, *id.* Esimerkiksi */exercises/14*. Liikuntalajiin linkitetyt tyypit löytyvät vielä tämän polun alta osoitteesta */exercises/14/types*.

Käytettävät http-metodit olisivat GET, POST, PUT ja DELETE. Näillä voidaan suorittaa kaikki sovellukselta vaadittavat toimenpiteet.

5.4 Sovelluksen rakenne

Sovelluksen rakenne suunniteltiin noudattamaan tyypillistä MVC-mallia. Sovellus ei varsinaisesti sisältäisi kuitenkaan näkymää, vaan Controller tarjoaa rajapinnan, jota voi käyttää http-metodien avulla. Rajapinta voidaan toki lukea näkymäksi siinä missä html-sivukin. Kuvassa 4 on kuvattu sovelluksen arkkitehtuuri ja Java-luokat. Suunnitteluvaiheessa kuvan kaikkia luokkia ei vielä tiedetty, mutta MVC-malli säilyi muutostenkin jälkeen selkeänä.



Kuva 4. Sovelluksen MVC-arkkitehtuuri.

Sovelluksen rakennetta ei suunniteltu alussa kovin pitkälle, vaan rakennetta muokattiin tarpeen mukaan työn edetessä. Alussa tiedettiin, että tarvitaan Java-luokat liikuntalajeille ja lajityypeille. Näiden lisäksi tarvittiin luokka, joka hallinnoi REST-rajapintaa, sekä luokat tietokantayhteyden hallintaan. Kehitysvaiheessa ilmeni tarve myös Service-luokille liikuntalajeja ja tyyppejä varten, koska REST-rajapintaa hallinnoiva luokka alkoi sisältää liikaa lajien ja tyyppien käsittelyyn liittyvää logiikkaa. Lajeja ja tyyppejä käsittelevä logiikka haluttiin siirtää omiin luokkiinsa.

Sovelluksen rakenteessa tapahtui muutoksia myös testien kirjoittamisen yhteydessä. Moni testi vaati metodien korjailua, ja esimerkiksi edellisessä kappaleessa mainittujen Service-luokkien luominen helpotti sovelluksen toimintojen testaamista.

6 TOTEUTUSVAIHE

Opinnäytetyön työstäminen alkoi samoin, kuin kovin moni muukin ohjelmointiin liittyvä projekti, jossa tutustutaan uusiin teknologioihin: runsaalla internethaulla ja yksinkertaisiin tutoriaaleihn perehtymällä. Opinnäytetyön käytännön osuuteen saatiin hyvä lähtölaukaus työn tilaajan järjestämästä koulutuspäivästä. Päivän aikana rakennettiin perusrakenteeltaan hyvin samankaltainen sovellus käyttäen samoja teknologioita, mistä oli apua kehitystyön käynnistymisessä.

6.1 Java-sovellus

Sovelluksen ohjelmoinnissa käytetystä Javasta oli jo ennestään kokemusta, joten sen käyttöön päädyttiin jo senkin vuoksi. Kielen perusrakenne oli tuttu, joten suurin opettelu oli Springin ja JPA-kirjastojen käytössä. Lisäksi työpaikalla käytetty ohjelmointikieli on hyvin laajalti Java.

6.1.1 Spring Bootin käyttöönotto

Spring Boot on osa Spring ohjelmistokehystä. Sovelluksessa päädyttiin käyttämään Spring Bootia lähinnä sen vuoksi, että se on työn tilaajalla laajasti käytössä ja soveltuu hyvin mikropalvelun toteuttamiseen. Vaihtoehtoja Spring Bootille löytyy runsaasti, esimerkiksi Node.js-, Ruby- ja Python-kirjastoina. Spring Bootin puolesta puhuivat sen käyttäminen työn tilaajalla, Java-ohjelmoinnin tuttuus, sekä laajasti internetistä löydettävä tukimateriaali.

Springin verkkosivuillaan tarjoama Spring Initializr on työkalu, jolla on helppo luoda pohja Spring Boot sovellukselle. Opinnäytetyön kirjoitushetkellä työkalu on saatavilla osoitteessa https://start.spring.io/. Spring Initializr antaa luoda Maven- tai Gradle-projektin, ja ohjelmointikielenä voi käyttää Javaa, Groovya tai Kotlinia. Opinnäytetyössä valittiin käyttöön Maven ja kieleksi Java. Työssä käytettiin Spring Bootin versiota 2.0.2.

Spring Initializrissa voi määrittää projektin metatietoja tarkemmin niin halutessaan. Kuvassa 5 on esimerkki tietokantasovelluksessa käytetyistä metatiedoista. Metatietojen lisäksi sovellukseen voidaan määrittää valmiiksi Springin Java-kirjastoja, joita sovelluksessa halutaan käyttää. Valikoimassa on kaikki modernin websovelluksen kehitykseen vaadittavat kirjastot, esimerkiksi tietokantayhteyksien, rajapintojen ja käyttöliittymän kehittämiseen. Valitut kirjastot lisätään automaattisesti Mavenin POM.xml tai

Gradlen gradle.build -tiedostoihin. Kun kirjastot on määritelty näissä kokoonpanotiedostoissa, Maven tai Gradle noutaa tiedostot Mavenin repositoriosta.

Project Metadata
Artifact coordinates
Group
net.ambientia
Artifact
uftcexercises
Name
uftcexercises
Description
Exercise database service for UFTC application
Package Name
net.ambientia.uftcexercises
Packaging
Jar
Java Version
8

Kuva 5. Tietokantapalvelun Spring Initializr -määrityksiä.

Sovelluksen kehityksen lomassa kirjastoja lisättiin ja poistettiin tarpeen mukaan. Lisääminen oli yksinkertaista, sillä IntelliJ IDEA ilmoitti, jos kaivattua resurssia ei löytynyt. Jos resurssi jäi tarpeettomaksi, jäi se helposti myös POM.xml -tiedostoon ylimääräisenä roikkumaan.

Yhtenä Springin kätevimpänä ominaisuutena pidin @Autowired-annotaatiota. Yksinkertaistettuna annotaatio injektoi tämänhetkiseen Java beaniin muita beaneja. Koodilistauksen 1 esimerkissä luokka ExerciseService on määritelty @Service-annotaatiolla. @Service on yksi @Component-annotaation erikoistuneista stereotyypeistä, jolla on oma roolinsa. @Component, @Service, ynnä muut mahdollistavat luokkien automaattisen löytymisen Javan classpathin kautta. Jotta @Autowired toimisi, täytyy luokat olla määritelty esimerkiksi @Component-annotaatiolla, sillä @Autowired injektoi luokat juurikin classpathin perusteella. Springissä on muitakin vastaavia annotaatiota, joita käytetään eri tyyppisille luokille.

Koodilistauksen 1 esimerkissä on määritelty kolme muuttujaa, *exerciseRepository*, *exerciseTypeRepository* ja *exerciseTypeService*. Luokan *ExerciseService* konstruktorin yläpuolella on @Autowired-annotaatio, jonka vuoksi parametrina annetut luokat injektoidaan kuva luokkaan. Konstruktoria ei siis kutsuta missään, vaan parametreista luodut oliot ovat käytettävissä luokassa. Lisäksi *ExerciseRepository* ja *ExerciseTypeRepository* ovat

rajapintoja, joten @Autowired poistaa tarpeen luoda rajapinnoista implementaatiot.

```
@Service
public class ExerciseService {

    private ExerciseRepository exerciseRepository;
    private ExerciseTypeRepository exerciseTypeRepository;
    private ExerciseTypeService exerciseTypeService;

    @Autowired
    public ExerciseService(ExerciseRepository exerciseRepository,
                           ExerciseTypeRepository exerciseTypeRepository,
                           ExerciseTypeService exerciseTypeService) {
        this.exerciseRepository = exerciseRepository;
        this.exerciseTypeRepository = exerciseTypeRepository;
        this.exerciseTypeService = exerciseTypeService;
    }
}
```
Koodilistaus 1. Spring @Autowired-annotaatio.

6.1.2 Sovelluksen JPA-entiteetit

Minkä tahansa MVC-malliin pohjautuvan Java-sovelluksen perustana ovat luokat, jotka kuvaavat tietokannassa esiintyviä tauluja. Opinnäytetyössä käytetty Hibernate osaa luoda luokkien perusteella tietokantaan tarvittavat taulut ja niiden väliset yhteydet, jolloin ohjelmoijan tarvitsee huolehtia niistä ainoastaan lähdekoodin puolella.

Sovelluksen tietokannan rakenne on kuvattu kuvassa 2. Tietokantaa varten tarvitsi luoda JPA-entiteettiluokat. Liikuntaharjoitteelle luotiin luokka *Exercise* ja liikuntatyypeille *Type*. Koska kummassakaan ei voi olla duplikaattinimiä, oli aluksi suunnitelmana pitää entiteetin nimi myös sen id:nä. Kehitystyön edetessä huomattiin, että numeerisen id-tietueen puuttuminen johti ongelmiin REST rajapinnan ja repositorioiden kanssa. Havaitiin, että oli yksinkertaisesti helpompaa pitää myös numeerinen id mukana, sillä käytetyt kirjastot ja sovelluskehykset oli selvästi suunniteltu niitä silmällä pitäen.

Koodilistauksessa 2 näkyy *Exercise*-luokan määrittelyjä. *Id* on automaattisesti generoitu kokonaisluku, joka tulee PostgreSQL-tietokannan sequence-taulusta. Liikuntaharjoitteen nimi, *name*, on String-muotoinen ja määritelty annotaatiolla @NaturalId, joka on Hibernaten tapa merkitä entiteetin luonnollinen tunniste, kuten uniikki nimi. Nimi on määritelty pakolliseksi. Lisäksi harjoitteessa on vielä *types*, jonka tyypiksi on määritelty Set-kokoelmaluokka. *Types* sisältää liikuntatyypit, jotka liikuntalajiin on mahdollisesti liitetty.

Liikuntalajien ja liikuntatyyppien yhdistely oli eniten päänvaivaa tuottanut asia entiteettejä käsiteltäessä. Koodilistauksen 2 @ManyToMany-an-

notaatio tarkoittaa, että lajien ja tyyppien välillä on monesta moneen -yhteys. Tietokantaa tarvitaan tällöin kolmas taulu, jossa yhteys on kuvattu. Lähdekoodissa kolmatta taulua ei kuvata omana entiteettinään, vaan Hibernate luo taulun tietokantaan @ManyToMany- ja @JoinTable-annotaatioiden perusteella. Koodilistauksessa 2 @JoinTable-annotaatiossa määritellään liitostaulun nimi *exercise_type*, käsiteltävän luokan id:tä kuvaava tietue *exercise_id*, sekä vastapuolen luokan id:hen viittaava *type_id*. Luokassa Type ei määritellä lainkaan liitostaulua, vaan siellä @ManyToMany saa lisämäärityksen *mappedBy = "types"*, joka viittaa Exercise-luokan kenttään *types*.

Exercise- ja Type-luokissa on lisäksi määritelty @JsonIgnoreProperties-annotaation avulla tarpeettomia tietoja, joiden ei haluttu päätyvän rajapinnan kautta haettavaan JSON-dataan.

```
@Entity
@JsonIgnoreProperties({"hibernateLazyInitializer", "handler"})
public class Exercise {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    private Long id;

    @NaturalId(mutable = true)
    @Column(nullable = false)
    private String name;

    @ManyToMany(
            cascade = {CascadeType.MERGE, CascadeType.PERSIST},
            fetch = FetchType.EAGER)
    @JoinTable(name = "exercise_type",
            joinColumns = @JoinColumn(name = "exercise_id"),
            inverseJoinColumns = @JoinColumn(name = "type_id"))
    @JsonIgnoreProperties("exercises")
    private Set<Type> types = new HashSet<>();
```

Koodilistaus 2. Esimerkki sovelluksen JPA-entiteetin kentistä.

6.1.3 Entiteettien hallinta

Sovelluksessa on kaksi rajapintaluokkaa, *ExerciseRepository* ja *ExerciseTypeRepository*, jotka ovat JpaRepositoryn laajennoksia. JpaRepository on rajapintaluokka, jonka avulla voidaan tehdä kyselyitä tietokantaan. JpaRepositorylla onnistuvat kaikki tarvittavat CRUD-toimenpiteet.

```java
@Repository
public interface ExerciseRepository extends
        JpaRepository<Exercise, Long> {
    boolean existsByName(String name);
    Exercise getByName(String name);
    @Query(value = "SELECT nextval('hibernate_sequence')",
            nativeQuery = true)
    Long getNextSeriesId();
}
```

Koodilistaus 3. Esimerkki sovelluksen repository-rajapintaluokasta.

Koodilistauksessa 3 on esitetty *ExerciseRepository*-rajapintaluokka. JpaRepositorysta laajennetut rajapinnat toimivat perustarkoituksessaan ilman erikseen määritettyjä ominaisuuksia. Lajitietokantasovellusta varten molempien entiteettien repository-rajapintaan luotiin kuitenkin metodit *existsByName*, *getByName* ja *getNextSeriesId*. Ensimmäinen tarkistaa onko tietokannassa riviä haetulla nimellä ja toinen palauttaa nimen perusteella Exercise-olion. Kolmatta metodia käytettiin sovelluksen yksikkö- ja integraatiotesteissä, joissa oli tarve tietää millä id:llä seuraava taulun rivi luodaan.

6.1.4 Controller-luokka

Sovelluksen ulospäin suuntautuvan rajapinnan palveluista huolehtii *ApiController*-luokka, joka on merkitty Springin @RestController-annotaatiolla. @RestController-luokissa määritellään rajapinnan polut, sekä niistä saatavat vastaukset. Koodilistauksessa 4 on esitetty liikuntalajien resurssin osoite ja sen http:n GET-kutsuun vastaava metodi. @GetMapping kertoo Springille, että sen perässä määritettyyn polkuun tulevat GET-kutsut käsitellään alla olevan metodin mukaan. */exercises*-polkuun lähetetään GET-kutsu, ja koodilistauksen metodi hakee repositoriosta kaikki liikuntaharjoitteet ja lisää ne palautettavaan listaan. Spring huolehtii listan serialisoinnista JSON-muotoon, joka voidaan lukea asiakassovelluksessa.

```java
@GetMapping("/exercises")
public List<Exercise> getExercises() {
    List<Exercise> listToReturn = this.exerciseRepository.findAll();
    listToReturn.sort(Comparator.comparingLong(Exercise::getId));
    return listToReturn;
}
```

Koodilistaus 4. Esimerkki ApiControllerin resurssin käsittelystä.

Koodilistauksessa 5 on esimerkki /exercises-polkuun tehdyn GET-kutsun vastauksem JSON-sisällöstä. JSON-data sisältää listan liikuntalajeista, jotka on kukin esitetty JSON-objektina, sekä niihin liitetyt liikuntatyypit. Esimerkkilistauksessa vastauksena on saatu vain yksi liikuntalaji, johon on liitetty kaksi liikuntatyyppiä.

```json
[
  {
    "id": 3,
    "name": "biking",
    "types": [
      {
        "id": 1,
        "name": "outdoors"
      },
      {
        "id": 2,
        "name": "commute"
      }
    ]
  }
]
```

Koodilistaus 5. Esimerkki GET-metodilla saadun vastauksen JSON-sisällöstä.

ApiControllerissa määritellään kaikki polut, joiden avulla voidaan tehdä lajitietokantaan hakuja ja muutoksia. Liikuntaharjoitteiden ja niiden tyyppien käsittelyssä ApiControllerin ja repositorioiden välillä hyödynnetään service-luokkia.

6.1.5 Service-luokat

Sovelluksen kehityksen keskivaiheilla ApiController-luokka sisälsi kaiken logiikan, joka tarvittiin rajapinnasta tulevien kutsujen ja datan käsittelyyn. ApiController kutsui suoraan repository-luokkia, mikä ei ole toivottavaa vähänkään monimutkaisempaa logiikkaa vaatitissa toimenpiteissä. Erillisten service-luokkien luominen helpottaa myös yksikkötestien kirjoittamista. Tähän tehtiin muutos, sillä controllerin pitäisi huolehtia ainoastaan rajapinnan toiminnoista, eikä puuttua datan jatkokäsittelyyn.

Liikuntaharjoitteille luotiin service-luokka *ExerciseService* ja liikuntatyypeille *ExerciseTypeService*. Näihin luokkiin siirrettiin logiikkaa joistakin ApiControllerin metodeista. Esimerkiksi ExerciseService-luokkaan luotu metodi *addTypesToExercise* sisältää null-arvojen tarkistuksia, joukon käsittelyn for-silmukassa sekä virheentarkistuksia. Metodi on nähtävillä koodilistauksessa 6.

```
public void addTypesToExercise(Long id, Set<Type> typeSet)
        throws NotFoundException {
    String notFoundTypes = "";
    if (exerciseRepository.findById(id).isPresent()) {
        Exercise exercise = exerciseRepository.findById(id).get();
        for (Type type : typeSet) {
            try {
                if (exerciseTypeRepository.
                        findById(type.getId()).isPresent()) {
                    exercise.addType(exerciseTypeRepository.
                            findById(type.getId()).get());
                    exerciseRepository.save(exercise);
                    exerciseTypeService.toString();
                } else {
                    notFoundTypes += id + " ";
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
        if (notFoundTypes.length() > 0) {
            throw new NotFoundException(
                    "Could not add type(s): " + notFoundTypes);
        }
    } else throw new NotFoundException(
            "No exercise exsits with id " + id);
}
```

Koodilistaus 6. ExerciseService-luokan metodi addTypesToExercise.

Erillisten service-luokkien olemassaolo helpottaa myös sovelluksen jatkokehitystä, kun kaikki entiteettien käsittelyyn vaadittava koodi ei olekaan ApiControllerissa. Service-luokkia voidaan tällöin käyttää muissa sovelluksen luokista.

6.1.6 Sovelluksen rajapinta

Liikuntalajitietokanta voisi olla aivan hyvin oma taulunsa UFTC-sovelluksen omassa tietokannassa, mutta se rajoittaisi tietokannan tietojen käyttöä. Mikropalvelun olennainen ajatus on, että se tarjoaa rajapinnan, jonka avulla tietoa siirretään sen ja muiden sovellusten välillä.

Rajapinnan resurssit ovat saavutettavissa osoitteista, joihin voi tehdä http:n CRUD-metodien kutsuja. Rajapintaliittymiin otetaan yhteys URL:n avulla. Esimerkiksi paikallisessa kehitysympäristössä liikuntalajit voidaan hakea GET-kutsulla osoitteesta http://localhost:8080/exercises. Sovelluksessa on mukana README.md-tiedosto, jossa on tarkempi kuvaus rajapinnan käytöstä. Taulukossa 1 on esitetty resurssien osoitteet, tuetut http-metodit, sekä lyhyt kuvaus eri kutsujen roolista.

Osoite	Tuetut http-metodit	Kuvaus
/exercises	GET POST DELETE	GET palauttaa kaikki liikuntalajit tyyppeineen. POST lisää liikuntalajeja. DELETE poistaa yksittäisen tai useampia liikuntala-jeja.
/exercises/{id}	GET DELETE	GET palauttaa id:tä vastaavan liikuntalajin tyyppei-neen. DELETE poistaa id:tä vastaavan liikuntalajin.
/exercises/{id}/types	GET PUT DELETE	GET palauttaa tiettyyn liikuntalajiin liitetyt tyypit. PUT lisää tai muokkaa liikuntalajin tyyppejä. DELETE poistaa tyyppejä liikuntalajista, mutta ei kokonaan.
/types	GET POST DELETE	GET palauttaa kaikki liikuntatyypit. POST lisää liikuntatyyppejä. DELETE poistaa yksittäisen tai useampia tyyppejä.
/types/{id}	GET DELETE	GET palauttaa yksittäisen liikuntatyypin. DELETE poistaa yksittäisen liikuntatyypin.
/types/{id}/exercises	GET	GET palauttaa kaikki liikuntalajit, joihin tyyppi on liitetty.

Taulukko 1 Sovelluksen resurssien osoitteet ja tuetut http-metodit.

Liikuntalajit ja -tyypit palautetaan resurssista riippuen joko yksittäisenä JSON-objektina tai JSON-objektilistana. Liikuntalajin JSON-objektista on esimerkki koodilistauksessa 5. Tietoja lisättäessä tai päivitettäessä lähetetään data myös JSON-muodossa. Useimmat http-metodit palauttavat http-statuskoodin, sekä JSON-datana objektin, johon kutsu on kohdistunut. Osa palauttaa JSON-datan sijasta vain tekstimuotoisen informaation siitä, mitä tehtiin.

6.2 Sovelluksen vienti OpenShiftiin

Opinnäytetyössä käytettiin työn tilaajan omaa OpenShift-alustaa. Alustassa oli saatavilla sovelluskuva tietokannalle, sekä S2I-kuva Java-sovellukselle. Sovelluskuvat noudetaan Git-repositoriosta. Paikallisesti OpenShiftin saa pyörimään Minishiftin avulla, jonka avulla saa pystyyn yhden noodin OpenShift-klusterin.

OpenShiftiin luotiin uusi projekti tietokantasovellusta varten. Projektin nimeksi annettiin yksinkertaisesti UFTC. Vakiona Minishiftissä oli saatavilla joitakin sovelluskuvia, mutta Java-sovelluksen tarpeisiin ei ollut sopivaa saatavilla. Sovelluskuvia on runsaasti saatavilla esimerkiksi GitHubissa. Niitä on mahdollista luoda myös itse ja tähän löytyy oppaita internetistä.

6.2.1 PostgreSQL OpenShiftissä

PostgreSQL-tietokannalle löytyi valmis sovelluskuva, jonka käyttöönotto vaati muutaman hiirenklikkauksen, sekä tarvittavien asetusten määrittämisen. Sovelluksen *application.properties*-tiedostossa oli määritelty tietokannan osoitteeksi *db:5432/uftcexercisedb*. OpenShiftissä tietokantapalvelun nimeksi annettiin myös *db* Java-sovelluksen määrityksen mukaisesti, jolloin sovellus sai tietokantaan suoraan yhteyden. Järkevämpi, tai ainakin oikeampi tapa toteuttaa tietokantayhteyden määritys, olisi käyttää sovellukselle OpenShiftissä määriteltäviä ympäristömuuttujia, jolloin esimerkiksi tietokantapalvelun osoite tai tietokannan nimi eivät olisi kovakoodattuna lähdekoodiin. Tätä ei opinnäytetyöhön käytetyn ajan puitteissa kuitenkaan toteutettu, vaan se jää jatkokehityskohteeksi.

Tietokantapalveluun määriteltiin myös tietokannan nimi *uftcexercisedb*, mutta tietokantaa ei kuitenkaan luotu automaattisesti. OpenShiftin podeihin saa tarvittaessa ssh-yhteyden, tai hallintapaneelista löytyvä komentorivi myös mahdollistaa podilla suoritettavat tehtävät. Kumpikin käyttää samaa komentotulkkia, joka ei käytettävyydeltään ole kummoinen, mutta ajaa asiansa. Komentoriviltä kirjauduttiin PostgreSQL-palveluun sisään ja sinne luotiin tarvittava käyttäjätunnus ja tietokanta manuaalisesti. Kaiken kaikkiaan tietokannan pystytys oli OpenShift-ympäristössä erittäin vaivatonta.

6.2.2 Java-sovellus OpenShiftissä

Työn tilaajan OpenShiftissä oli paljon tarjolla runsaasti sovelluskuvia, sillä niitä oli sinne ajan mittaan tuotu, joten siellä sovelluksen käyttöönotto oli vaivattomampaa. Minishiftissä Java-sovellukselle ei ollut kuitenkaan sopivaa sovelluskuvaa valmiina, vaan sellainen täytyi tuoda OpenShiftiin. Tarve oli S2I-kuvalle, joka tarjoaa sovelluspohjan, johon sovelluksen lähdekoodi lisätään erikseen. Sovelluskuva oli jo onneksi tiedossa, joten tarvitsi enää vain etsiä se.

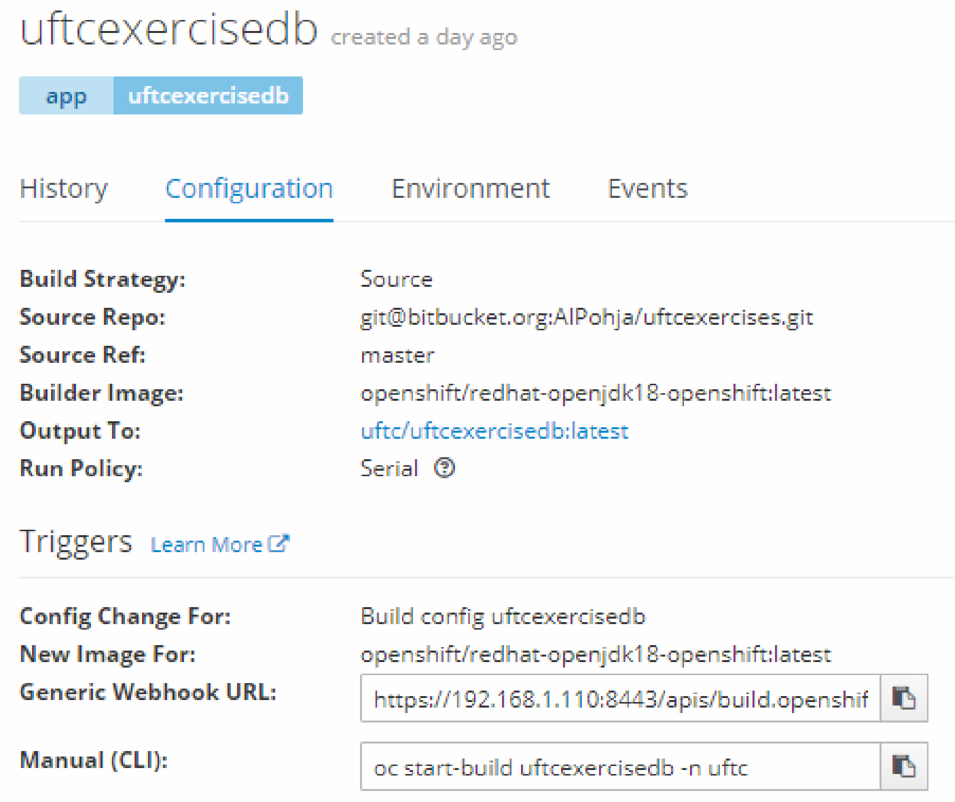
S2I-sovelluskuvan saa helposti tuotua OpenShiftiin koodilistauksen 7 kaltaisen JSON-tiedoston avulla. Esimerkin JSON-tiedostossa määritellään Docker-repositorion osoite, josta Docker-kuva noudetaan. S2I-kuvan tapauksessa tärkeää on *annotations*-elementin *tags*-alielementin määrite *builder*, joka kertoo OpenShiftille, että tätä sovelluskuvaa käytetään S2I-kuvana. Kun kuva otetaan käyttöön projektissa, määritetään sille vielä erikseen osoite, josta sovelluksen lähdekoodi noudetaan.

```
{
  "kind": "ImageStream",
  "apiVersion": "v1",
  "metadata": {
    "name": "redhat-openjdk18-openshift"
  },
  "spec": {
    "dockerImageRepository": "registry.access.redhat.com/redhat-openjdk-
18/openjdk18-openshift",
    "tags": [
      {
        "name": "1.0",
        "annotations": {
          "description": "OpenJDK S2I images.",
          "iconClass": "icon-jboss",
          "tags": "builder,java,xpaas",
          "supports":"java:8,xpaas:1.0",
          "sampleRepo": "https://github.com/jboss-openshift/openshift-
quickstarts",
          "sampleContextDir": "undertow-servlet",
          "version": "1.0"
        }
      }
    ]
  }
}
```

Koodilistaus 7. S2I-sovelluskuvan JSON-määritys OpenShiftiin tuontia varten.

Kuvassa 6 on esitelty uftcexercisedb-sovelluksen konfiguraatio. *Source Repo* -kohdassa on osoite Bitbucketin repositorioon, josta lähdekoodi noudetaan. Lähdekoodin hakemista varten on luotu ssh-avainpari, joista yksityinen osa tallennetaan OpenShiftiin ja julkinen osa repositorion pääsyavaimiin. Avaimen avulla lähdekoodin haku onnistuu helposti. OpenShiftiin voi tallentaa salaisuuksia (*Secrets*), kuten juurikin ssh-avaimia tai palveluiden käyttäjätunnuksia. Niitä ei silloin säilytetä selkokielisenä palvelussa.

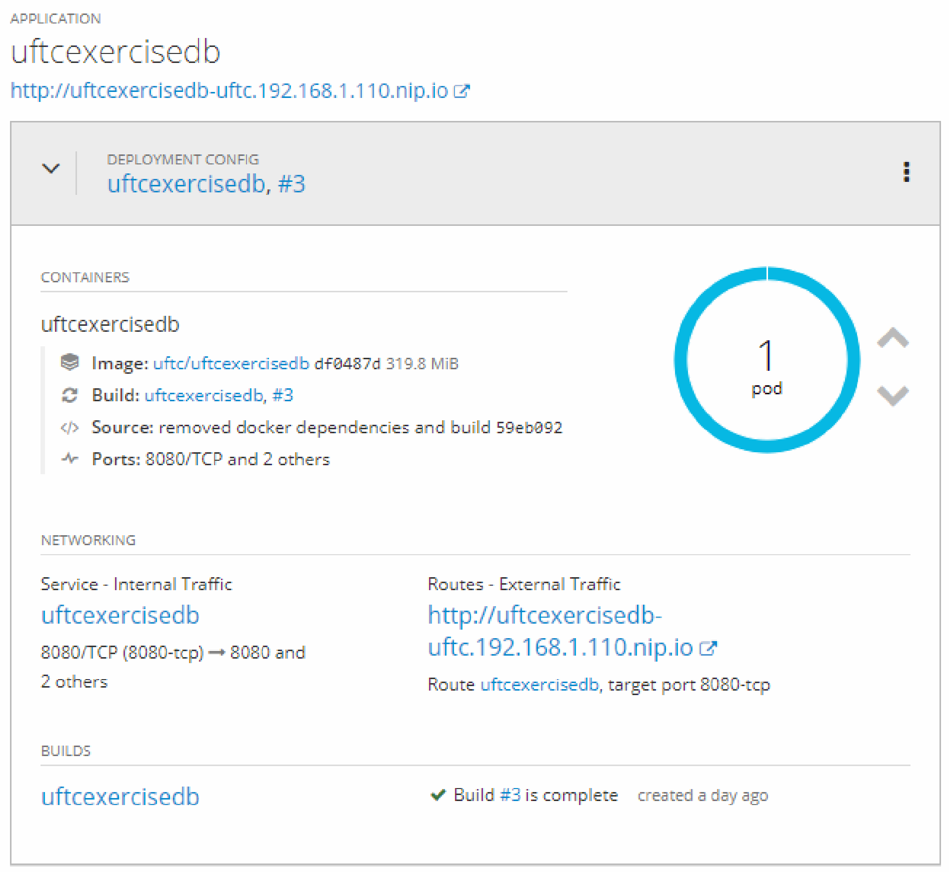
Kuvan 6 *Source Ref* viittaa repositorion kehityshaaraan, josta lähdekoodi noudetaan. OpenShiftiin voi tarvittaessa luoda esimerkiksi testi- ja laadunvalvontaympäristöt, joihin lähdekoodi noudetaan eri kehityshaaroista. *Builder Image* kertoo sovelluksen kääntämisessä käytettävän S2I-sovelluskuvan. *Output To* on käännöksen tuloksena syntyvän sovelluskuvan nimi, joka sisältää lopullisen sovelluksen, joka voidaan ottaa käyttöön OpenShiftissä. *Triggers*-osiossa määritellään mikä laukaisee uuden sovelluskuvan kääntämisen. Laukaisijana voi olla esimerkiksi lähdekoodin tai sovelluskuvan muutos.



Kuva 6. Sovelluksen build-asetuksia OpenShiftissä.

Projektin etusivulla OpenShift näyttää yhteenvedon oleellisesta tiedosta. Esimerkin yhteenvedosta voi nähdä kuvassa 7. Sovelluksen yleistilan näkee oikealla olevasta ympyrästä, jonka keskellä ilmaistaan myös kuinka monta podia sovelluksella on käytössään. Sininen väri tarkoittaa, että sovellus toimii normaalisti ja esimerkiksi punainen ilmaisee virhettä. Yleisnäkymässä on listattu myös sovelluksen kontin tiedot, jossa näkyy esimerkiksi käytetty sovelluskuva, sekä mistä lähdekoodin versiosta sovellus on käännetty. Alempana näkyy OpenShiftin sisäinen verkkonimi sekä mahdollinen ulospäin näkyvä osoite, josta sovellukseen pääsee käsiksi ulkoverkosta. Lajitietokantasovellukselle on tässä esimerkissä määritetty ulkoinen osoite, jotta sen toimintaa on ollut helppo testata. Lopullisessa käyttötarkoituksessaan se on saavutettavissa vain OpenShiftin sisältä.

Sovellus vaati toimiakseen yhteyden tietokantaan, joten tietokanta käynnistettiin ensin. Sovelluksen asetuksissa oli määritetty tietokannan yhteysosoite ja kun sovellusta käynnistettiin, toimi tietokantayhteys saman tien. Ensimmäisten epäonnistuneiden yritysten jälkeen todettiin, että PostgerSQL-kantaan täytyi luoda manuaalisesti oikeanniminen tietokanta, sekä käyttäjätunnukset, joilla on sinne tarvittavat oikeudet. Kun tietokanta oli sovelluksen vaatimassa kunnossa, lähti sovelluksen podi käyntiin hienosti ja se alkoi vastaamaan sille määritettystä osoitteesta.



Kuva 7. OpenShiftin yleisnäkymä sovelluksen tilasta ja tiedoista.

7 JOHTOPÄÄTÖKSET JA POHDINTA

Opinnäytetyön prosessi on ollut varsin pitkä johtuen etupäässä elämäntilanteesta, mutta se on viimeisen vuoden aikana edennyt tasaisesti. Työn, perheen ja opinnäytetyön yhdistäminen on ollut ajoittain hankalaa, mutta onneksi joustoa on löytynyt, kun sitä on vaadittu. Tässä luvussa kerron miten prosessi eteni, miten opinnäytetyön tavoitteet täyttyivät ja mitä työn aikana opittiin. Lisäksi tuon esiin työhön perehtyneiltä tai tutustuneilta saatua palautetta, sekä aivan lopuksi vielä esittelen mahdollisia jatkokehityskohteita.

7.1 Prosessin eteneminen

Opinnäytetyön aiheenvalinta lienee aina ongelmallista ja niin tässäkin tapauksessa. Vaihdoin aihetta kokonaan kertaalleen, mutta lopulliseen aiheeseen olin koko työn ajan tyytyväinen. Valittu aihe kiinnosti minua ajatuksen tasolla ja myös tekemisen osalta, mikä loi tarvittavan motivaation työn jatkuvalle etenemiselle, vaikka siinä kestikin melko kauan.

Sovellukseen käytettävät teknologiat tarjottiin minulle valmiina ehdotuksena, mikä helpotti työn käynnistymistä. Minun ei tarvinnut käyttää aikaa sen pohtimiseen, mikä teknologiapino olisi sopiva OpenShiftissä toimivalle mikropalvelulle, vaan sain toimivaksi todetut työkalut käyttööni. Teknologiat tukivat myös ammatillista kehittymistäni, sillä niitä tulisin mitä todennäköisemmin käyttämään töissä tulevaisuudessa.

Mikropalveluista, OpenShiftistä ja Springistä löytyi kirjallista materiaalia hyvin. Lisäksi teokset olivat laadukkaita ja hyvin kirjoitettuja. Sen sijaan PostgreSQL-tietokannan osalta jouduin turvautumaan hyvin pitkälti internetlähteisiin, koska kunnollista kirjamateriaalia ei työn tarpeisiin tahtonut löytyä. Löydetyt kirjat keskittyivät enimmäkseen tietokannan hallintaan ja kyselyiden tekemiseen perustiedon sijasta. Lähdemateriaalin puute ei koitunut ongelmaksi, koska opinnäytetyötä ajatellen tietokannan rooli oli taustapalvelu. Työn aikana sinne ei tarvinnut tehdä oikeastaan lainkaan kyselyitä manuaalisesti. Riitti, että tietokantaan loi kannan ja tarvittavat käyttäjätunnukset, jonka jälkeen Java-sovellus hoiti loput.

Suunnittelutyö oli varsin suoraviivaista, vaikka sovelluksessa tapahtuikin muutoksia työn aikana. Muutokset ovat tavallista ohjelmointityössä, joten osasin olla stressaamatta niistä. Sovelluksen perusarkkitehtuurin rakennuspalikat eivät kuitenkaan kokeneet merkittäviä muutoksia työn edetessä. Esimerkiksi tietokanta, entiteettiluokat ja rajapinta toteuttivat lopulta ne ominaisuudet, jotka niihin oli suunniteltu.

Kehitystyössä hyödynsin runsaasti internetin tutoriaaleja. Niistä oli paljon hyötyä erityisesti silloin, kun aloin kehittämään uutta ominaisuutta. Olennainen osa tiedonetsintää olivat myös esimerkiksi Springin ja OpenShiftin dokumentaatio. Moneen ongelmatilanteeseen ratkaisu, tai ainakin polku ratkaisuun löytyi kysymysten ja vastausten pohjalta Stackoverflow-sivustolla. Stackoverflowsta löysin myös parempia tapoja tehdä työn alla olevia asioita.

Kaikista sujuvinta tekeminen oli silloin, kun sain keskittyä ohjelmointityöhön tai sen dokumentoimiseen. Tietoperustan kirjaaminen raporttiin lähdemateriaalin pohjalta oli henkisesti työläintä vaihetta. Käytännön osuuden raportointi, sekä tämän yhteenvedon tekeminen olivat mielekästä kirjoittamista, jolloin tekstiäkin syntyi jouhevasti.

7.2 Tavoitteiden saavuttaminen ja opitut asiat

Työn tavoitteena oli rakentaa UFTC-sovelluksen tarpeisiin sopiva mikropalvelu, joka tarjoaa sovelluksen käyttöön tietokannan liikuntalajeista. Lisäksi tavoitteena oli, että mikropalvelu toimii OpenShiftissä. Näihin molempiin tavoitteisiin päästiin, vaikka palvelua ei vielä raportin kirjoitusvaiheessa oltu otettu käyttöön. Lajitietokanta kuitenkin tukee UFTC-sovelluksen uudistusprosessin etenemistä.

Henkilökohtaisena, ja myös työnantajan tavoitteena opinnäytetyöprosessissa on ollut tekijän ammatillinen kehittyminen ohjelmoijana, sekä työpaikalla käytössä olevien teknologioiden haltuunotto. Työn aikana opin runsaasti Java-ohjelmoinnista ja erityisesti Spring-sovelluskehyksestä. Lisäksi yksikkö- ja integraatiotestien kirjoittaminen oli uutena asiana erittäin hyödyllistä ja sai ajattelemaan ohjelmointiprosessia uudelta kantilta.

7.3 Saatu palaute

Tein kehitystyötä varsin itsenäisesti ja ainoastaan kertaalleen keskustelin ohjaajien kanssa työn etenemisestä. Tuossa vaiheessa palaute oli sen suuntaista, että asioita oli selvästi mietitty ennen toteutusta. Lisäksi sain rohkaisevaa palautetta jo kirjoittamistani testeistä ja lisävinkkejä niiden kirjoittamiseen. Työn keskivaiheilla saatu palaute oli arvokasta ohjaamaan kehitystyötä parempaan suuntaan. Samassa opinnäytetyöryhmässä olleilta opiskelijoilta tuli myös palautetta ja sieltä nousi monesti esiin asioita, joita ei itse ollut tullut ajatelleeksi. Opponentilta tuli väliseminaarissa myös rakentavaa palautetta työstä ja hyviä muutos- ja lisäysehdotuksia.

7.4 Jatkokehitys

Sovelluksen jatkokehityskohteista kiireellisin on OpenShiftin ympäristömuuttujien käyttöönotto tietokantayhteyden määrittelyssä, sillä kehitys-

vaiheen tapa tietokantayhteyden tietojen tallennukseen application.properties-tiedostoon ei ole erityisen turvallinen tai joustava. Muutos tullaan toteuttamaan todennäköisesti sovelluksen käyttönoton yhteydessä.

Sovelluksen rajapinnan dokumentointi on tehty manuaalisesti ja se on sovelluksen README.md-tiedostossa. Tiedoston sisältö on esitelty liitteessä 1. Dokumentointiin voisi ottaa tulevaisuudessa käyttöön Swaggerin, joka on suunniteltu rajapintojen automaattiseen dokumentointiin. Swagger luo rajapinnasta interaktiivisen näkymän, jonka avulla rajapinnan käytön periaatteet selviävät nopeasti. (SmartBear, 2018)

Sovellusta olisi mahdollista laajentaa hallintakäyttöliittymällä, jota käytettäisiin liikuntalajien ja -tyyppien hallintaan. Tämä voidaan toteuttaa esimerkiksi osana UFTC-sovellusta, tai siihen voidaan rakentaa myös kokonaan oma palvelunsa. Tällaisen palvelun kehittäminen voisi sopia opinnäytetyön aiheeksi.

LÄHTEET

Docker (2018). Docker Documentation, Dockerfile reference. Haettu 2.5.2018 osoitteesta https://docs.docker.com/engine/reference/builder/.

Docker Hub (2018). Openshift/origin-base, Full Description. Haettu 2.5.2018 osoitteesta https://hub.docker.com/r/openshift/origin-base/.

Dumpleton, G. (2018). Deploying to OpenShift. Sebastopol, Kalifornia, USA: O'Reilly Media.

Johnson, R. (2006). Spring Framework: The Origins of a Project and a Name. Blogijulkaisu 9.11.2006. Haettu 15.9.2018 osoitteesta https://spring.io/blog/2006/11/09/spring-framework-the-origins-of-a-project-and-a-name/.

The Linux Foundation (2018a). Kubernetes Components. Haettu 8.9.2018 osoitteesta https://kubernetes.io/docs/concepts/overview/components/.

The Linux Foundation (2018b). What is Kubernetes? Haettu 8.9.2018 osoitteesta https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.

Mouat, A. (2015). Using Docker. Sebastopol, Kalifornia, USA: O'Reilly Media.

Newman, S. (2015). Building Microservices. Sebastopol, Kalifornia, USA: O'Reilly Media.

OKD (2018). OKD Documentation, Source-to-Image (S2I), Using Images. Haettu 2.5.2018 osoitteesta https://docs.okd.io/latest/welcome/index.html.

Opensource.com (2018). What is Docker? Haettu 2.5.2018 osoitteesta https://opensource.com/resources/what-docker.

The PostgreSQL Global Development Group (2018a). PostgreSQL Documentation. What is PostgreSQL? Haettu 15.9.2018 osoitteesta https://www.postgresql.org/docs/current/static/intro-whatis.html.

The PostgreSQL Global Development Group (2018b). Haettu 15.9.2018 osoitteesta https://www.postgresql.org/docs/current/static/history.html.

The PostgreSQL Global Development Group (2018c). About. Haettu 15.9.2018 osoitteesta https://www.postgresql.org/about/.

Red Hat (n.d.). OpenShift Architecture Overview. Haettu 15.9.2018 osoitteesta https://docs.openshift.com/enterprise/3.0/architecture/index.html.

Red Hat (2018a). OpenShift Product Pricing. Haettu 15.9.2018 osoitteesta https://www.openshift.com/pricing/index.html.

Red Hat (2018b). OpenShift Dedicated. Haettu 15.9.2018 osoitteesta https://www.openshift.com/products/dedicated/.

Red Hat (2018c). OpenShift Container Platform. Haettu 15.9.2018 osoitteesta https://www.openshift.com/products/container-platform/.

Red Hat (2018d). OpenShift Documentation, Builds and Image Streams. Haettu 16.9.2018 osoitteesta https://docs.openshift.com/container-platform/3.5/architecture/core_concepts/builds_and_image_streams.html.

Red Hat (2018e). OpenShift Documentation, Kubernetes Infrastructure. Haettu 16.9.2018 osoitteesta https://docs.openshift.com/enterprise/3.0/architecture/infrastructure_components/kubernetes_infrastructure.html.

Red Hat (2018f). OpenShift Documentation, Overview. Haettu 15.9.2018 osoitteesta https://docs.openshift.com/container-platform/3.10/architecture/index.html.

Red Hat (2018g). OpenShift Origin, OKD: The Origin Community Distribution of Kubernetes README.md. Haettu 15.9.2018 osoitteesta https://github.com/openshift/origin/blob/master/README.md.

Red Hat (2018h). Kubernetes infrastructure, Infrastructure components. Haettu 8.9.2018 osoitteesta https://docs.openshift.com/enterprise/3.0/architecture/infrastructure_components/kubernetes_infrastructure.html.

Ramgir, M & Samoylov, N. (2018). Java: High-Performance Apps with Java 9. Packt Publishing, 2018.

Richardson, C. & Smith, F. (2016). Microservices From Design to Deployment. San Francisco, Kalifornia, USA: Nginx Inc.

Shipley, G. & Dumpleton, G. (2016). OpenShift for Developers, A Guide for Impatient Beginners. Sebastopol, Kalifornia, USA: O'Reilly Media.

SmartBear (2018). Documenting Your Existing APIs: API Documentation Made Easy with OpenAPI & Swagger. Haettu 13.12.2018 osoitteesta https://swagger.io/resources/articles/documenting-apis-with-swagger/.

Solid IT (2018). DB-Engines Ranking. Haettu 15.9.2018 osoitteesta https://db-engines.com/en/ranking.

Walls, C. (2016). Spring Boot in Action. New York, USA: Manning Publications Co.

Wiggins, A. (2017). The Twelve-Factor App. Haettu 15.9.2018 osoitteesta https://12factor.net/.

Liite 1/1

SOVELLUKSEN README.md-tiedosto

UFTC Exercise database API

The UFTC exercise database microservice stores and provides all the exercises for the UFTC application, or any other application that might need them.

Database

The database structure is really simple. There's an exercise table and a type table. Exercise table contains all the exercises and the type table all the exercise types. Types can be thought as tags that describe the exercise, eg. "outdoors", "commute", "physical". The idea is to be able to filter exercises based on tags if you're looking for a certain type of workout.

Table names are `exercise`, `type` and the join table is `exercise_type`. These are created by the application. The `exercise` and `type` tables have `id` and `name` fields. There's a many-to-many connection between exercises and types which is handled in the join table. The tables are created and managed by the application so all data modifications should be done by API calls.
The application can use for example PostgreSQL or MySQL and a database needs to be created and the datasource info defined in `application.properties`. For example:

```
spring.datasource.url=jdbc:postgresql://localhost:5432/uftcexercisedb
spring.datasource.username=postgres
spring.datasource.password=postgres
```

The application has been tested with PostgreSQL but it may work with other databases as well.

API

All the API paths below are in relation to the applications base url. For example the GET request for all exercises should be made at `http://localhost:8080/exercises`

Exercise access points

All exercises

```
/exercises
```

Supported HTTP methods

Method	Accepts	Produces
GET	-	application/json
POST	application/json	application/json
DELETE	application/json	application/json

GET sample response

```
[
    {
        "id": 3,
        "name": "biking",
        "types": [
            {
                "id": 1,
                "name": "outdoors"
            },
            {
                "id": 2,
                "name": "commute"
            }
        ]
    }
]
```

POST sample

```
[
    {
        "name": "biking"
    },
    {
        "name": "running"
    }
]
```

POST sample to update exercise names

```
[
    {
        "id": 1,
        "name": "newName"
    }
]
```

DELETE sample

```
[
    {"id": 18 },
    {"id": 19 },
    {"id": 20 },
    {"id": 21 }
]
```

Single exercise

Supported HTTP methods

Method	Accepts	Produces
GET	-	`application/json`
DELETE	`application/json`	`application/json`

`/exercises/{id}`

POST sample to update exercises

```
[
    {
        "id": 1,
        "name": "jogging"
    }
]
```

Types of a single exercise

`/exercises/{id}/types`

Supported HTTP methods

Method	Accepts	Produces
GET	-	`application/json`
PUT	`appplication/json`	`appplication/json`

Associating types with an exercise

`/exercises/{id}/types`

Supported HTTP methods

Method	Accepts	Produces
PUT	`application/json`	`application/json`

PUT sample with type ids to associate with an exercise

```
[
    {
        "id": 8
    },
    {
        "id": 11
    },
    {
        "id": 14
    }
]
```


All exercise types

```
/types
```

Supported HTTP methods

Method	Accepts	Produces
GET	-	`application/json`
POST	`application/json`	`application/json`
DELETE	`application/json`	`application/json`

GET sample response


```
[
    {
        "id": 1,
        "name": "outdoors"
    },
    {
        "id": 2,
        "name": "commute"
    }
]
```


POST sample to create new types


```
[
    {
        "name": "outdoors"
    },
    {
        "name": "atwork"
    }
]
```


POST sample to update types


```
[
    {
        "id": 2,
```

```
        "name": "outdoors"
    },
    {
        "id": 5,
        "name": "atwork"
    }
]
```


Single type

```
/types/{id}
```

Supported HTTP methods

Method	Accepts	Produces
GET	-	`application/json`
DELETE	`application/json`	`application/json`

GET response for single type


```
{
    "id": 1,
    "name": "outdoors"
}
```


All exercises associated to a type

```
/types/{id}/exercises
```

Supported HTTP methods

Method	Accepts	Produces
GET	-	`application/json`

Additional notes

- You can add types along with an exercise in a JSON object, but you will get an error if one of the types exists in the database already. It is recommended to add them separately and associate them after that.
- It's possible to add, update and delete exercises and types in a batch