Java-ohjelmistoprojekti hyödyntäen Spring-kehystä

Oskari Lehtonen

Haaga-Helia ammattikorkeakoulu

Amk-opinnäytetyö

2021

Tietojenkäsittelyn tutkinto

Tiivistelmä

Tekijä(t)
Oskari Lehtonen

Tutkinto
It-tradenomi

Raportin/Opinnäytetyön nimi
Java-ohjelmistoprojekti hyödyntäen Spring-kehystä

Sivu- ja liitesivumäärä
34 + 20

Tämän opinnäytetyön motivaationa on ollut palvelinpään ohjelmoinnin tason nostaminen nykyisen työmarkkinatilanteen vaatimalle tasolle. Tämän lisäksi työ auttaa ohjelmistoprojektin kokonaisuuden hahmottamisessa, sekä laadukkaan koodin tuottamisessa. Työn päätavoite on tekijän osaamisen syventäminen.

Tässä opinnäytetyössä on luotu pikajuoksijalle kohdennettu harjoituspäiväkirja Java web -projektina. Työssä on hyödynnetty Spring Boot -sovelluskehystä Java-koodin tuottamisen helpottamiseksi. Tekijä on tehnyt tarvittavat vaatimusmäärittelyt sekä suunnittelut ennen itse lähdekoodin kirjoittamista, jotta työtapa kuvaisi paremmin oikeita työolosuhteita. Työssä on tämän lisäksi paneuduttu tapoihin, miten pystytään tuottamaan helposti luettavaa, laadukasta sekä hyviin tapoihin perustuvaa koodia.

Työn ei ole tarkoitus jäljitellä ohjelmistoprojektia täysimittaisesti, vaan työtä on rajattu useammasta kohdasta. Projektissa ei ole projektiryhmää eikä se noudata mitään projektihallinnan suuntauksista. Ohjelma keskittyy pelkästään palvelinpuolen koodiin ja siitä on rajattu pois testaus, käyttöliittymän teko sekä autentikointi.

Tietoperustaan on koottu projektin sekä ohjelmoinnin osalta keskeisimmät asiat. Tietoperustassa ei vertailla vastakkaisia tapoja suorittaa tehtäviä tai käydä aihe alueita läpi laajasti, vaan tietoperusta on projektissa käytettyjen tapojen tukena.

Tekijä on tässä työssä käynyt ohjelmistoprojektiaan läpi vaiheittain, samalla kertoen, miksi on päätynyt tiettyihin valintoihin ja millaisilta tehdyt valinnat ovat tuntuneet. Tekijän kokemukset ja päätelmät ovat havaintoja, joita työn aikana on tullut esiin. Kokonaisuutena työtä kannattaa tarkastella enemmän tekijän kokemuksien näkökulmasta, jotta työn todellinen tarkoitus tulee ymmärretyksi.

Asiasanat
Ohjelmistokehitys, Java, Spring Boot, palvelinohjelmointi

Sisällys

1 Johdanto

Koska työpaikalla vaaditaan ammattitaitoa, työkokemusta ja vuorovaikutustaitoja, tulee jokaisen työnhakijan hankkia kokemus itselleen valitsemallaan tavalla. Oma motivaationi ja idea tähän projektiin tuli työhakuprosessin kautta, kun hakuprosessista tuli kieltävä vastaus. Miksi sain kieltävän vastauksen? Satojen hakijoiden tasaisuudesta on vaikea erottautua, jos takana ovat vain koulussa tehdyt projektit. Jos ei työkokemusta ole opiskelujen ohella kertynyt, tulee omien projektien olla tasokkaita, jotta työpaikan saaminen olisi mahdollista. Vaihtoehtona on myös tehdä opinnäytetyö, joka mukailee oikeaa projektia, ja samalla tuo hyvää kokemusta tekijälle tulevia työnhakuja ajatellen. Tämä opinnäytetyö on yksi esimerkki tällaisesta.

Työn aiheena on luoda palvelinpään koodi Javalla, hyödyntäen Spring Boot -sovelluskehystä. Työn tarkoitus on edistää omaa oppimistani ohjelmistoprojekteissa ja sen kokonaisuuden hahmottamisessa. Tässä projektissa pystyn luomaan jotain alusta loppuun niin, että ohjelmalla on tietyt ominaisuudet ja kriteerit. Työhön kuuluvan rakennettavan sovelluksen on tarkoitus helpottaa pikajuoksijan harjoittelun seurantaa jokapäiväisessä harjoittelussa. Työn päänäkökulmana on kuitenkin itse ohjelmointi ja ohjelmistoprojektin suorittaminen. Työn valmistuessa tuotoksena ei ole valmis tuote, vaan valmistettavan sovelluksen yksi osa. Työhön on sisällytetty luokat, tietokantayhteys vaadittavilla CRUD-ominaisuuksilla sekä REST-rajapinta. CRUD sekä REST sisältävät tietojen lukemisen, lisäämisen, muokkaamisen sekä poistamisen. CRUD hoitaa kanssakäymisen tietokannan kanssa, kun taas REST HTTP-pyyntöjen kanssa. Ohjelmoinnista pois on jätetty käyttöliittymän teko, autentikointi sekä sovelluksen testaus siihen tarkoitetuilla metodeilla. Sovelluksen testaus on suoritettu tietokantakyselyiden kautta. Olen tähän dokumenttiin kartoittanut laadukkaan koodin luomiseen tarvittavia käytäntöjä. Laadukkaan koodin osalta olen tukeutunut Robert C. Martinin kirjoittamaan Clean Code -myyntimenestykseen sekä Steve McConnellin vanhaan, mutta sitäkin arvostetumpaan Code Complete 2 -käsikirjaan.

Työ rajataan ohjelmoinnin osalta palvelinpuolen koodiin. Käyttöliittymää ei tässä projektissa tehdä, vaan sovelluksen toimivuus pystytään tarkistamaan Postman-sovelluksen kautta. Postman pystyy lähettämään HTTP-pyyntöjä käyttäjäystävällisessä muodossa, ja pyyntöjen vastaukset ovat luettavissa välittömästi. Sovellusta on jatkossa mahdollista kehittää eteenpäin valmiimmaksi versioksi, mutta tässä opinnäytetyössä keskitytään vain ohjelman palvelinpuolen koodiin ja sen tuomiin haasteisiin.

Tämän lisäksi olen tuonut tähän dokumenttiin pätkiä omasta koodistani, joista kerron omia mietteitäni ja päätelmiä perustuen ennalta oletettuihin standardeihin. Käyn läpi omaa

ohjelmointiurakkaani ja kerron, mikä oli haastavaa ja mikä helppoa. Tämän avulla pystyn tuomaan esille omia osaamisalueitani sekä mielipiteitäni.

Ohjelmistoprojektin osalta työssä on käsitelty suunnittelua ja ohjelmointiin valmistautumista. Tarkoituksena on jäljitellä karkealla tasolla oikeaa ohjelmistoprojektia, kuitenkin siten, että vain yksi ihminen suorittaa projektin alusta loppuun. Suunnitteluun ja ohjelmointiin valmistautumiseen liittyvät kohdat ovat toteutettu pääpiirteittäin, jotta projektikokonaisuudesta saadaan mahdollisimman laaja kuva, ei niinkään yksityiskohtainen. Projektityöhön liittyvä kokonaisuus kattaa muitakin näkökulmia kuin vain palvelinpuolen ohjelmointia, kuten huomataan vaatimusmäärittelyssä, mikä käsittelee sovellusta isompana kokonaisuutena. Työssä ei ole tarkasteltu mitään projektiryhmän kanssakäymisen näkökulmasta, vaan työ on tehty yksilötyönä, eikä muita osapuolia ole hyödynnetty. Hyvän ohjelmistoprojektin määritelmään olen etsinyt tietoa Sampo Karjalaisen Pro gradu -tutkielmasta, joka tutkii ohjelmistoprojektin onnistumista toimittajan näkökulmasta (Karjalainen 2008). Lisäksi olen käyttänyt Mikko Mäntynevan teoksia, sillä hän on liiketoiminnan kehittämisen yliopettaja, kouluttaja, puhuja ja tietokirjailija. Lisäksi hän on kirjoittanut viisi kirjaa, jotka käsittelevät esimerkiksi kasvua ja projektitoimintaa (Kauppakamari s.a.).

Koska aihe alue on todella laaja, kokonaisuuden hahmottamisseksi tässä opinnäytetyössä keskitytään pääsääntöisesti seuraaviin kysymyksiin.

Kuinka tehdä ohjelmistoprojekti alusta loppuun?
Kuinka rakentaa toimiva palvelinpään koodi Javalla?
Kuinka tuottaa helposti luettavaa ja ymmärrettävää koodia?

Käsitteet

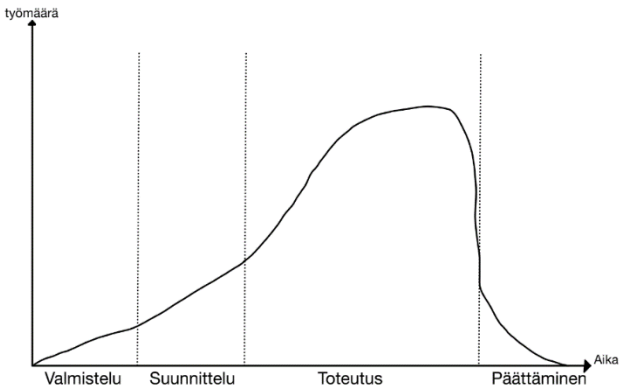
Annotaatio	Antaa lisätietoja ohjelmasta kääntäjälle (GeeksforGeeks 2020).
Back-end	Ohjelmiston koodi, joka ajetaan palvelun palvelimella (Dagmar 2015).
CRUD	Englannin kielen akronyymi sanoille luoda, lukea, päivittää ja poistaa (Altvater 2017).
Docker	Sovelluksien paketoimiseen käytettävä sovellus (Docker s.a.).
Front-end	Ohjelmiston graafisen käyttöliittymän koodi (Dagmar 2015).
Git	Hajautettu versionhallinta järjestelmä (Git s.a.).
HTTP	Selainten ja www-palvelimien tiedonsiirtoprotokolla (Vanhatapio 2020)

IDE	Ohjelmointiympäristö (Integrated Development Environment), joka sisältää editorin koodin kirjoittamiselle, tulkin ohjelmointikielen suorittamiseen ja debuggerin (Johansson 2019).
Java	Ohjelmointikieli (FutureLearn 2021).
JDK	Ohjelmakehityspaketti (Java Development Kit), joka on tarkoitettu Java-ohjelmointiin (Tyson 2020).
JPA	Ohjelmarajapinnan määritys (Java Persistance API), jolla on tarkoitus hallita relaatiotietoja Java-sovelluksissa (Oracle s.a.).
JSON	Kevyt datan tiedonsiirtomuoto (JavaScript Object Notation) (JSON s.a.).
LTS	Pitkän ajan tuki ohjelmistolle (Long Term Support), joihin tehdään esimerkiksi turvallisuuskorjauksia (Freitag 2020).
MVC	Arkkitehtuurimalli (Model, View, Controller), jota käytetään moderneissa web-sovelluksissa (Agile Education Research 2019).
Ohjelmointi	Tietokoneelle tai muulle laitteelle annettavien toimintaohjeiden antaminen. Puhekielessä koodaus. (Jämsen 2016)
Yhden suhde yhteen	Yhden suhde yhteen -assosiaatio (Oracle 2011).
REST	Arkkitehtuurimalli (Representational State Transfer), joka on tarkoitettu ohjelmointirajapintojen toteuttamiseen (Agile Education Research 2019).
Tietokanta	Kokoelma tietoja, johon voidaan suorittaa hakuja ja jonka tietoja voidaan muuttaa (Laaksonen 2020).

2 Ohjelmistoprojektin kulku

2.1 Projektin päätekijät

Jotta ohjelmistoprojektin läpivienti onnistuisi hyvin, tulisi projektissa keskittyä neljään asiaan: ihmisiin, tuotteeseen, prosessiin ja itse projektiin (Karjalainen 2008, 8). Ohjelmistoteollisuudessa tärkeimpänä edellytyksenä onnistuneisiin projekteihin pidetään henkilökuntaa (Karjalainen 2008, 9). Ihmisten kommunikaatio on tullut yhä tärkeämmäksi osaksi ohjelmistokehittäjän työhaastatteluja, kun useimmat yritykset katsovat hakijan soveltuvuutta kommunikaatiotaidoissa (Mayuko 2019).
Tuotteen määritteleminen on projektin kannalta tärkeää, sillä ilman kunnollista määritystä järkevien työmäärien arvioiminen on mahdotonta, eikä realistista projektisuunnitelmaa pystytä rakentamaan. Tuotteen määrittämisellä tulisi selvittää sen vaikutusalue. Vaikutusalue tulisi olla ymmärrettävissä niin asiakkaan, projektin johdon, sekä teknisen osaston osalta. (Karjalainen 2008, 9)
Koska ohjelmistoprojektimalleja on useita erilaisia, tulee projektista vastaavan valita oikea prosessimalli juuri kyseiselle tuotteelle. Tähän vaikuttaa esimerkiksi ohjelmistoyrityksen historia ja yrityksen sisäiset prosessit. Prosessin valinnan jälkeen tuote ja prosessi voidaan sulauttaa yhteen projektisuunnitelmaksi. (Karjalainen 2008, 10)
Oheisessa kuviossa on yksi esimerkki projektin elinkaaren vaiheista ja niihin kohdentuvasta työmäärästä.



Kuva 1. Projektin elinkaari (mukaillen Mäntyneva 2016, 6)

2.2 Projektin toteuttajat

Projektin asettaja tekee projektin käynnistämispäätöksen. Yleensä asettajana toimii projektin tilaaja, joka valitsee projektille ohjausryhmän, hoitaa rahoituksen ja vastaa resurssien riittävyydestä. (Mäntyneva 2016, 21)

Ohjausryhmän tarkoitus on käydä läpi ja hyväksyä projektisuunnitelma. Ohjausryhmä on isossa osassa projektin valvomiseen liittyvissä asioissa, sillä se seuraa projektin toteutumista ja pitää projektin toimintaympäristön yhteistyösuhteita yllä. Ohjausryhmä on myös tukena projektipäällikön päätöksissä. Edellä mainittujen asioiden takia ohjausryhmä on mukana koko projektin käynnissä olon ajan. (Mäntyneva 2016, 21)

Projektipäällikkö johtaa projektin toteutumista. Hän on muun muassa asiantuntija, neuvottelija, asiakasyhteyshenkilö, tilaaja ja tiedottaja. Kun projektisuunnitelma on hyväksytty, projektipäällikön tehtävänä on sen toteuttaminen. Projektilla on hyvä mahdollisuus onnistua, jos projektipäälliköllä on esimerkiksi hyvät viestintä- ja neuvottelutaidot, kyky huolehtia monesta asiasta samanaikaisesti, hallita kokonaisuutta tai johtaa ihmisiä. (Mäntyneva 2016, 31–33)

Yleensä projektille määritellään myös projektisihteeri, joka luo aikatauluja, laatii tarjouskyselyitä sekä vastaa kokousjärjestelyistä. Sihteeri luo myös kokouksista muistiot sekä seuraa kustannuksien kulusta projektin aikana. (Mäntyneva 2016, 21)

Projektiryhmän jäsen on osa tekemisen ydintä. Jokaisella projektiryhmän jäsenellä on oma tehtäväalueensa, jolle on erikoistunut. Hänen on määrä huolehtia omien tehtäviensä tekemisestä ja raportoida tehtävien edistymisestä projektipäällikölle. Jäsenen on tarkoitus ylläpitää ja kehittää omia taitojaan, jotta projektille tuodut hyödyt olisivat mahdollisimman suuret. (Mäntyneva 2016, 21)

2.3 Ohjelmiston elinkaari

Tietokoneohjelmistojen kehittäminen voi olla sen suuruuden ja monimutkaisuuden takia hankala prosessi. Ohjelmistokehityksen historian aikana tutkijat ovat tunnistaneet useita erilaisia osa-alueita, jotka sopivat termiin ohjelmistokehitys. Näitä ovat esimerkiksi ongelmien ratkaisu, ohjelmistoarkkitehtuurin suunnittelu, ohjelmointi ja virheenkorjaus, testaus sekä ylläpito. Yleensä yksin ohjelmoimaan opetelleet eivät ole erotelleet edellä mainittuja asioita, vaan mahdollisesti mieltävät kaikki edellä mainitut asiat ohjelmoinniksi. Epäviralli-sissa projekteissa työskennelleet taas käyttävät ohjelmoinnista termiä rakentaminen, joka on hieman kuvaavampi, vaikka tästäkin termistä puuttuu perspektiivi. (McConnell 2004, luku 1)

Ohjelmistotuotannon vaiheisiin kuuluvat esitutkimus, vaatimusmäärittely, suunnittelu, toteutus, testaus, käyttöönotto sekä ylläpito. Esitutkimuksessa on tarkoitus tuoda esille

yleiset vaatimukset ohjelmistolle, mutta sen ei tule vielä ottaa kantaa siihen, millainen ohjelmiston tulisi olla. Esitutkimus saattaa jatkua myös projektin edetessä, sillä asiakkaan tarpeiden analysointi jatkuu läpi koko projektin ajan. (Kasala 2016, 6–7)

Toisena vaiheena on vaatimusmäärittelyn tekeminen, jossa on tarkoitus kuvata järjestelmälle asetetut tavoitteet (Kasala 2016, 8).Vaatimusmäärittely on yksi ohjelmistojärjestelmien kehitystyön perustehtävistä (Paakki 2011, 1). Samalla se on kuitenkin yksi haastavimmista osista ohjelmistojen kehittämistä (Ruuska 2012, 7), sekä merkityksellisin vaihe ohjelmistoprojektin onnistumisen kannalta (Kasala 2016, 8). Sitä on tehty yhtä kauan kuin itse ohjelmistoja, mutta vaatimusmäärittelyn tyylit ja tehokkuus ovat parantuneet ajan myötä (Paakki 2011, 2).

Huonosti tehty vaatimusmäärittely on yleinen syy ohjelmistoissa ja järjestelmissä oleviin virheisiin, minkä takia tulee mahdollisia viivästyksiä tai lisäkustannuksia, mikä taas johtaa tyytymättömyyteen (Ruuska 2012, 7). Vaatimusmäärittelyä tehtäessä tulee löytää järjestelmävaatimuksien ongelmakohtia ja samalla löytämään sidosryhmien välinen yhteisymmärrys ohjelmistosta (Ruuska 2012, 15).

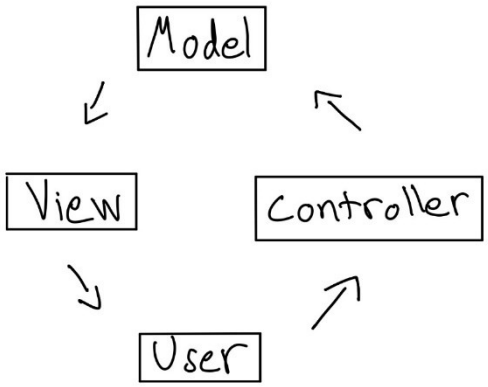
Vaatimusmäärittely tarkoittaa eri ihmisille ja organisaatioille eri asioita (Paakki 2011, 1). Ruuskan (2012, 17) mukaan vaatimusmäärittelyn tulisi sisältää seuraavat asiat:

1. Järjestelmältä vaaditut palvelut ja toiminnot
2. Järjestelmän rajoitukset
3. Yleiset ominaisuudet
4. Muihin järjestelmiin kohdistuvat integrointimääritykset
5. Järjestelmän sovellusalue
6. Järjestelmän kehittämisprosessin rajoitukset.

Kolmantena vaiheena on suunnittelu. Suunnitteluun kuuluu arkkitehtuurisuunnittelu, joka sisältää käyttöliittymän, sovelluslogiikan ja tietorakenteet sekä moduulisuunnittelu, joka sisältää komponentit ja konfiguraation. Suunnitteluvaiheessa suunnitellaan esimerkiksi käyttöliittymä. Tässä vaiheessa asiakas näkee ensimmäistä kertaa visuaalisen osan konkreettisesta tuotteesta ja saattaa jopa luulla, että ohjelmisto on melkein valmis, vaikka toteutusta ei ole edes aloitettu. (Kasala 2016, 10–11)

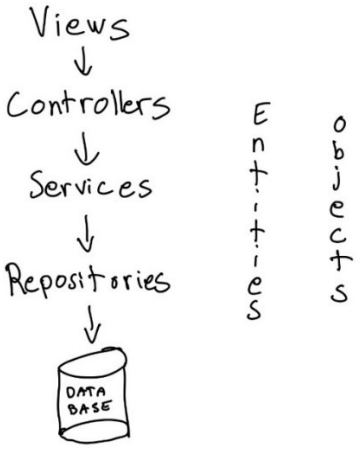
Ennen ohjelmoinnin aloittamista tulee myös valita ohjelmistolle sopiva arkkitehtuurimalli. Sovellukselle ominainen arkkitehtuurimalli helpottaa sovelluksen rakentamista, sillä tietyn tyyppisillä sovelluksilla on tietynlaisia vaatimuksia ja eri arkkitehtuurimallit sopivat paremmin eri sovelluksille. Internetissä yleisesti käytettävä MVC-malli, sekä verkkokaupoissa ja työpöytäsovelluksissa käytettävä kerroksittainen rakennemalli ovat kaksi yleisintä arkkitehtuurimallia World Wide Web -applikaatioiden rakentamisessa. (Mallawaarachchi 2017)

MVC-mallin tarkoituksena on erottaa käyttöliittymä sovelluksesta siten, että käyttöliittymät eivät tee sovelluksen kannalta tärkeitä sovelluslogiikan toimenpiteitä. (Agile Education Research 2019) Dynaamiset ja käyttöliittymäkomponentteja sisältävät web-sovellukset ovat hyviä kohteita käyttää MVC-mallia. Sovelluskehitys nopeutuu, koska koodista saadaan uudelleenkäytettävää. (Räisänen 2013, 7)



Kuva 2. MVC-malli

Kuvan kolme kerrosarkkitehtuuri jakaa sovelluksen selkeisiin vastuualueisiin. Tämä selkeyttää pyyntöjen kulkua sovelluksessa. Puhtaassa kerrosarkkitehtuurissa, jossa kaikki kerrokset ovat olemassa, kutsut eivät ohita kerroksia, vaan ne kulkeutuvat jokaisen kerroksen läpi (Agile Education Research 2019). Kerrosarkkitehtuuri mahdollistaa ohjelmiston eri osien määrittelyn osissa eri käsitetasoilla. Tämän takia kerrokset ei aseta rajoituksia muiden kerroksien käytölle. Jokaisen kerroksen muuttaminen tai toteuttaminen on mahdollista siten, että muihin kerroksiin ei tarvitse edes koskea. (Tverin 2007, 7)



Kuva 3. Kerrosarkkitehtuuri

Neljäntenä vaiheena on toteutus, jossa edellisten vaiheiden merkitys korostuu. Tämä johtuu siitä, että heikosti suunnitellun ohjelmiston vaatimukset ovat haastavia toteuttaa.

Vaatimusmäärittely on tärkeä dokumentti toteutuksen edetessä, sillä sitä voidaan käyttää toteutuksen aikana ohjelmoinnin kulmakivinä. Tämän takia ohjelmoinnista tulee hyvinkin suoraviivaista. Päämäärä on kuitenkin selvä ja yksinkertainen: valmistaa suunnitteluvaiheen mukainen järjestelmä. (Kasala 2016, 12)

Ohjelman rakentaminen on keskeisin toiminta ohjelmistokehityksessä. Rakentamiseen menee projektin koosta riippuen 30–80 prosenttia koko projektiin käytettävästä ajasta. Mikä tahansa asia, joka vie näin ison osan projektin ajasta on painopisteeltään tärkeä. Ohjelmiston vaatimukset sekä arkkitehtuuri on suunniteltu ennen projektia, jotta rakennusvaiheen voisi tehdä tehokkaasti. Järjestelmän testaus tehdään rakennusvaiheen jälkeen, jotta varmistutaan, että valmistus on tehty oikein. (McConnell 2004, luku 1)

Ohjelman rakentaminen on ainoa pakollinen osa projektia. Ihanteellinen ohjelmistoprojekti taas käy tarkasti läpi sovelluksen vaatimuksia ja arkkitehtuuriin liittyviä asioita ennen kuin itse ohjelmointi alkaa, kun taas usein työelämässä edellä mainitut osat jätetään välistä ja siirrytään suoraan ohjelman rakentamiseen. Samalla jätetään pois yleensä testaus, sillä virheitä on mahdollisesti niin paljon, että aika ei vain riitä niiden korjaamiseen. Ohjelmoinnin tärkeydestä kertoo myös se, että vaikka miten huonot esivalmistelut projektissa olisi, ei rakentamista voisi jättää väliin, sillä siitä syntyy itse lopputuote. (McConnell 2004, luku 1)

Viidentenä vaiheena on testaus, missä on tarkoituksena löytää ohjelmasta toteutusvaiheessa syntyneet virheet. Testausmuotoja ovat yksikkötestaus sekä järjestelmätestaus. Yksikkötestauksessa testauksen suorittaa itse ohjelmoija, kun taas järjestelmätestauksessa testaaja on jokin riippumaton taho. (Kasala 2016, 13)

Testauksen jälkeen tulee vaihe kuusi eli käyttöönotto. Yksinkertaisuudessaan se tarkoittaa tuotetun ohjelman julkaisemista eli käyttöönottoa. Käyttöönottoon sisältyy useita kohtia kuten esimerkiksi koekäyttöä, koulutusta sekä varsinainen käyttöönotto. Jotta ohjelmistosta saataisiin kaikki irti, tulee koulutus suorittaa loppukäyttäjille riittävän hyvin. Samalla tarkistetaan, että järjestelmä toimii hyvin uudessa ympäristössä myös siten, että se on yhteensopiva muiden järjestelmien kanssa. (Kasala 2016, 14)

Seitsemäntenä ja samalla viimeisenä vaiheena on ylläpito. Ylläpitoa voidaan hoitaa täydentävänä, korjaavana tai adaptiivisena. Täydentävässä on tarkoitus ohjelmiston eliniän nostaminen, korjaavassa paikataan virheitä, joita ilmenee käytön aikana ja adaptiivisessa ylläpidossa taas joko vaatimukset tai käyttöympäristö ovat muuttuneet. Ylläpito voi olla

joko tuottavaa tai kuluttavaa riippuen sen toteutustavasta. Huonosti hoidettu ylläpito on rasite, kun taas hyvin hoidettu tuo lisäarvoa molemmille osapuolille. (Kasala 2016, 16)

3 Laadukkaan koodin tuottaminen

3.1 Ohjelmointi

Laadukkaan koodin määritelmiä on varmasti yhtä monta kuin on ohjelmoijia. Pääpiirteittäin laadukas koodi on alan ammattilaisten mielestä hyvin jäsenneltyä, toimivaa, tehokasta, hyvin nimettyä sekä minimaalista. Yhtä kaavaa ei siis ole, vaan kokonaisuuden täytyy olla hallinnassa niin projektin valmistautumisesta itse koodin kirjoittamiseen. (Martin 2016, luku 1)

Huono koodi voi hidastaa projektin tekemistä huomattavasti. Jokainen muutos koodiin saattaa rikkoa useampaa osaa kokonaisuudesta. Kokonaisuudesta tulee niin iso ja sotkuinen, että sen ratkaiseminen on käytännössä mahdotonta. Samalla, kun sekasorto koodissa lisääntyy, työntekijöiden tuottavuus laskee. Kun projektiin yritetään lisätä työpanosta, yhä useampi ihminen lisää kokonaisuuden kompleksisuutta luoden ympäristöstä entistä monimutkaisemman. Työntekijöillä ei ole yhteistä säveltä, mikä laskee tuottavuutta entisestään, ja kierre on valmis. (Martin 2016, luku 1)

3.2 Nimeäminen

Nimeämistä on joka puolella sovellusta kirjoittaessa. Täytyy nimetä tiedostot, muuttujat, funktiot, luokat sekä paljon muuta. Koska sitä tehdään paljon, se on parempi tehdä hyvin. Hyvin nimetyn asian tulisi vastata kolmeen kysymykseen; sen tulisi kertoa miksi se on olemassa, mitä se tekee sekä miten sitä käytetään. Jos nimi tarvitsee kommentin selittääkseen mitä se tekee, niin se ei ole tarpeeksi tarkasti nimetty. (Martin 2016, luku 2)

Javassa luokkien nimet tulee kirjoittaa isolla alkukirjaimella, ja niiden pitäisi olla substantiiveja. Luokkien nimet täytyy pitää mahdollisimman yksinkertaisina ja kuvaavina, sekä ilman lyhenteitä, pois lukien yleisesti käytetyt kuten URL tai HTML. Metodien nimeämisessä on totuttu käyttämään verbejä pienellä alkukirjaimella aloitettuna, sekä useammalla sanalla nimetyssä metodissa jokaisen sisäisen sanan ensimmäisen kirjaimen isolla kirjoitettuna, kuten esimerkiksi haeKäyttäjä, luoKäyttäjä tai poistaKäyttäjä. (Oracle 1999)

Muuttujien nimeäminen tapahtuu pienellä alkukirjaimella ja sisäisten sanojen ensimmäinen kirjain isolla kirjoitettuna. Muuttujien nimien tulisi olla lyhyitä, mutta silti kuvaavia, eikä niitä tulisi aloittaa alaviivalla tai dollarimerkillä, vaikka kumpikin tapa on sallittu. Yhden kirjaimen nimiä tulisi muuttujissa välttää, paitsi jos ne ovat niin sanottuja ”pois heitettäviä” muuttujia, kuten esimerkiksi silmukoissa käytettäviä muuttujia. Muuttujien, joiden arvot ei

vaihdu, eli vakioiden nimissä, tulee kaikki kirjaimet kirjoittaa isolla kirjaimella ja useita sanoja sisältävissä nimissä sanojen väliin tulee laittaa alaviiva, kuten esimerkiksi MAX_ARVO. (Oracle 1999)

3.3 Kommentointi

Kommentit eivät ole lähtökohtaisesti hyviä. Oikeanlainen kommentointi muuttaa koodin helposti luettavaan muotoon, kun taas huono kommentti antaa väärää informaatiota ja sekoittaa lukijaa. Jos ohjelmakoodin nimeäminen on tehty tarpeeksi hyvin, ei kommentointia tarvitse kovin paljoa käyttää. (Martin 2016, luku 4)

Martinin (2016, luku 4) mukaan hyviä kommentointiaiheita ovat lakiasioihin liittyvät kommentit, informatiiviset kommentit, selvennykset, varoitukset sekä tekemättömiä komponentteja koskevat kommentit. Huonoja kommentteja hänen mukaansa ovat päiväkirjamaiset, liikaa informaatiota sisältävät ja toistavat kommentit, kirjanmerkkikommentit sekä koodin ulkopuolinen kommentointi.

Paljon informaatiota sisältävät kommentit ovat hyviä, jos joku muu on mahdollisesti käyttää lähdekoodiasi, mutta ei lue sitä kokonaan läpi. Jos olet kirjoittamassa kirjastoa tai kehystä, jota muut kehittäjät käyttävät, tulet tarvitsemaan jonkinlaista ohjelmointirajapinnan dokumentaatiota. Paljon informaatiota sisältävien kommenttien huono puoli on se, että jonkun ulkopuolisen, joka on vastuussa sovelluksen ylläpidosta, on haastavampi lukea sekavaa koodia, joka sisältää pitkiä kommenttipätkiä. (Sourour 2017)

3.4 Rakenne

Tänään kirjoittamasi koodi tulee hyvin suurella todennäköisyydellä muuttumaan seuraavassa julkaisussa. Tämän takia koodin rakenteella on iso merkitys esimerkiksi siihen, että miten helposti muutettava osa löytyy. Luettavuus ja ohjelmointityyli luovat asetelman, joka mahdollistaa ohjelmiston ylläpidettävyyden ja jatkokehityksen. (Martin 2016, luku 5)

Pienemmät tiedostot ovat yleensä helpompia ymmärtää kuin isot tiedostot. Pääsääntöisesti projektien tiedostokoko vaihtelee muutaman rivin mittaisesta koodista noin 500 rivin mittaiseen koodiin. Suurin osa tiedostoko'oista on 100 rivin molemmin puolin. Tiedoston sisältö tulisi rakentaa kuin uutisartikkeli. Otsikon eli tiedostonimen tulisi olla niin kuvaava, että se kertoo lukijalle, ollaanko oikeassa tiedostossa vai ei. Tiedoston yläosassa tulisi olla ohjelman kannalta tärkeitä konsepteja sekä algoritmeja ja mitä alemmas tiedostoa siirrytään, sitä tarkempia ja yksityiskohtaisempia tiedot ovat. Jopa 40 % koodirivin leveydestä sijoittuu 20–60 merkin sisään. Yli 80-merkkisten rivien määrä putoaa huomattavasti

verrattuna lyhyempiin riveihin, sillä kapeammat tiedostot ovat helpompia lukea kuin liian leveät. (Martin 2016, luku 5)

4 Ohjelmistoprojekti

Halusin pitää työn realistisena, joten en lähtenyt ajattelemaan sovellusta liian monimutkaisena. Suurin osa keskittymisestä oli kuitenkin opinnäytetyössä, mikä tarkoitti vastausta kysymyksiin:

Kuinka tehdä ohjelmistoprojekti alusta loppuun?
Kuinka rakentaa toimiva palvelinpään koodi Javalla?
Kuinka tuottaa helposti luettavaa ja ymmärrettävää koodia?

Sovellus on vielä kehitysvaiheessa, mutta toimiva, oikean rakenteen omaava kokonaisuus. Tässä vaiheessa seuraava askel sisältäisi sovelluksen jatkokehitystä ja parantelua kohti valmiimpaa tuotosta.

4.1 Suunnittelu

Suunnittelu lähti projektin osalta liikkeelle itse ideasta. Ideaksi syntyi harjoituspäiväkirja oman intohimoni, pikajuoksun, kautta. Olin suunnitellut harjoituspäiväkirjasovellusta jo pidempään ja tämä tilaisuus opinnäytetyön kanssa oli otollinen tämän projektin aloittamiseen. Koska toimin tässä projektissa yksin, ei projektiryhmää tai projektipäällikköä sen erityisemmin valittu. Tämä johti alussa siihen, että projektin aikatauluttaminen ja suorittaminen täytyi suunnitella ja toteuttaa hyvin alusta alkaen.

Realistisen aikataulun ja suunnittelun myötä pystyin hahmottamaan paremmin projektin suuruuden ja osa-alueet. Osa-alueet koostuivat vaatimusmäärittelyn tekemisestä, opinnäytetyön kirjoittamisesta sekä itse ohjelmoinnista. Varasin jokaiselle osa-alueelle riittävästi aikaa. Suurimman osan aikataulusta täytti ohjelmointi. Tiesin, että koodia kirjoittaessa virheiden määrä voi olla suuri ja niiden ratkaiseminen venyttäisi aikataulua, jos aikataulu olisi liian tiukka.

Suunnittelin aikataulua myös projektin elinkaarta kuvaavan kuvan mukaan (kuva 1). Kuvassa olevaan valmisteluosioon sisältyi tämän projektin kohdalla idean keksiminen ja sen hahmottaminen päässä sekä aikataulun laatiminen. Suunnitteluosioon kuului vaatimusmäärittelyn täyttö ja toteutusvaiheeseen ohjelmointi sekä opinnäytetyön kirjoittaminen. Kaavio kuvasi jälkeenpäin ajateltuna hyvin projektin kokonaisuutta, jopa näin pienessä projektissa.

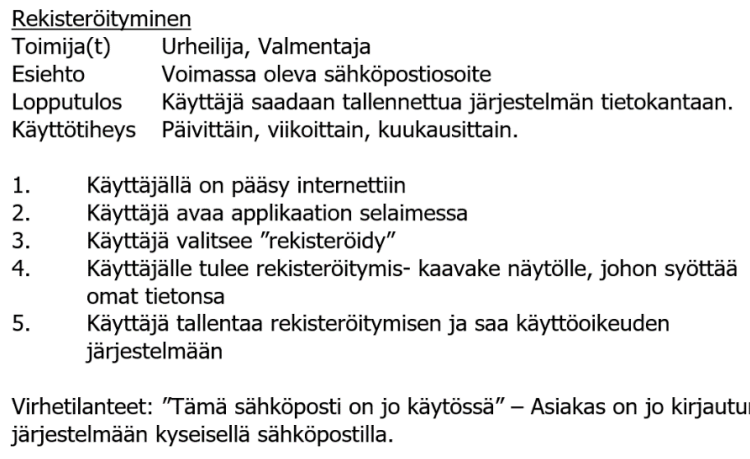
4.2 Vaatimusmäärittely

Vaikka sovelluksen rakenne oli pääpiirteittäin selvä omassa päässäni, oli seuraavana edessä vaatimusmäärittelyn tekeminen (liite 1). Aloitin vaatimusmäärittelyn tekemisen koulukurssilta tehdyn työn pohjalta. Kyseinen pohja on mielestäni kattava ja se sopii hyvin tähän projektiin, sillä iso osa kyseisestä vaatimusmäärittelystä sisältää tietoa juuri palvelinpuolelle. Vaatimusmäärittely ei ole täysin verrattavissa valmistettuun ohjelmaan, vaan vaatimusmäärittely sisältää myös ominaisuuksia isommasta kokonaisuudesta, joita tämänhetkinen versio ohjelmasta ei vielä sisällä.

Vaatimusmäärittelyä tehdessä huomasin, kuinka tarkkaan ohjelmisto piti oikeasti suunnitella. Vaikka ajatus omassa päässäni oli valmiin tuntuinen, ei se siltä tuntunut, kun keräsin tietoja ylös dokumentille. Vaatimusmäärittelyä tehdessä tuli useaan otteeseen muutettua omaa alkuperäistä suunnitelmaa. Suunnitelmat pääsääntöisesti supistuivat, kun omassa mielessäni olleet ajatukset olisivat vaatineet niin ison työmäärän, että yhden ihmisen projektina siinä olisi ollut aivan liian iso työ. Todella yksinkertaisessakin ohjelmassa tietokantataulukoiden ja luokkien hahmottaminen on haastavaa, sillä pienessäkin projektissa syntyvien tietokantataulukoiden määrä on suuri.

4.2.1 Käyttötapaukset

Vaatimusmäärittelyssä olevista käyttötapauksista kertovat esimerkit auttavat hahmottamaan sovelluksen käyttötarkoitusta. Kokosin vaatimusmäärittelyyn eri toimintapolkuja, joita käyttäjät tulevat suorittamaan ohjelmaa käyttäessään. Alla olevassa kuvassa (kuva 4) esimerkkitapaus rekisteröinnistä.



Rekisteröityminen

Toimija(t)	Urheilija, Valmentaja
Esiehto	Voimassa oleva sähköpostiosoite
Lopputulos	Käyttäjä saadaan tallennettua järjestelmän tietokantaan.
Käyttötiheys	Päivittäin, viikoittain, kuukausittain.

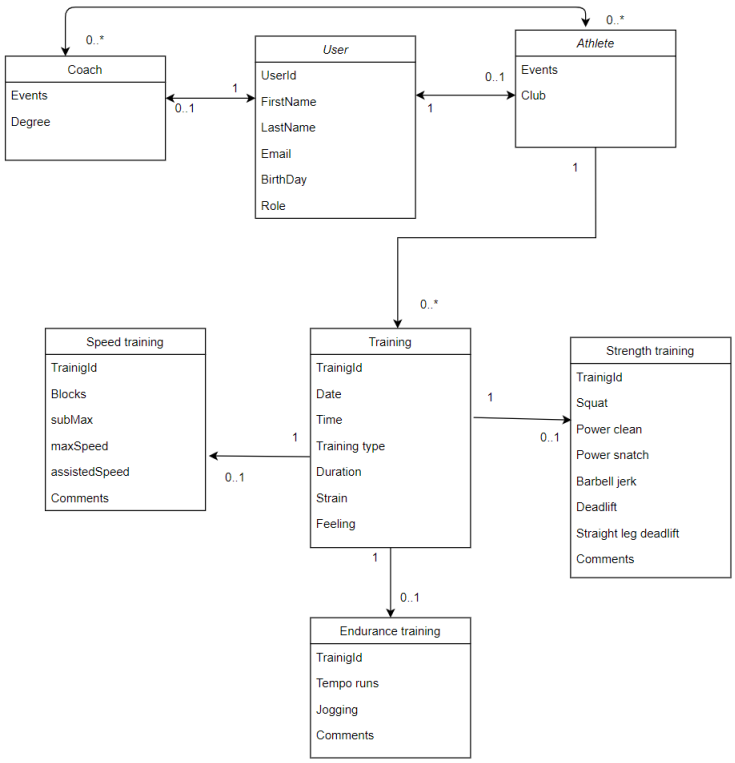
1. Käyttäjällä on pääsy internettiin
2. Käyttäjä avaa applikaation selaimessa
3. Käyttäjä valitsee ”rekisteröidy”
4. Käyttäjälle tulee rekisteröitymis- kaavake näytölle, johon syöttää omat tietonsa
5. Käyttäjä tallentaa rekisteröitymisen ja saa käyttöoikeuden järjestelmään

Virhetilanteet: ”Tämä sähköposti on jo käytössä” – Asiakas on jo kirjautunut järjestelmään kyseisellä sähköpostilla.

Kuva 4. Käyttötapaukset

4.2.2 Säilytettävät tiedot

Relaatiokaavioon tallennetaan järjestelmän tietokantaluokat ja niiden väliset suhteet. Relaatiokaaviosta pystyy tarkistamaan, onko luokkien välillä yhden suhde yhteen, yhden suhde moneen, monen suhde yhteen tai monen suhde moneen. Relaatiokaaviossa olevat nuolet kertovat myös onko luokkien välinen suhde kaksisuuntainen vai yksisuuntainen (engl. bidirectional ja unidirectional). Kaksisuuntainen suhde antaa kummallekin luokalle oikeuden lukea toistensa tietoja, kun taas yhdensuuntainen suhde vain toiseen suuntaan.



Kuva 5. Relaatiokaavio

Iso osa vaatimusmäärittelystä tuli tarpeeseen ohjelmoitaessa. Vaatimusmäärittely oli usein toisessa ikkunassa auki, kun tein Java-luokkia. Relaatiokaaviosta oli suuri apu, kun rakensin ohjelmiston luokkia ja loin tietokantarakennetta. Pystyin katsomaan vaadittavat kentät suoraan vaatimusmäärittelystä, eikä niitä tarvinnut keksiä lennosta tai lisäillä ja poistella jälkikäteen. Tässä tuli ensimmäistä kertaa huomattua kuinka paljon hyötyä vaatimusmäärittelystä oli. Voin vain kuvitella, miten suuri vaikutus on isoissa ja komplekseissa projekteissa.

Jokaisesta luokasta tehtiin tämän lisäksi tarkemmat määritelmät. Liitteessä 2 huomataan, kuinka luokalle annettu yleismääritelmä antaa tarkemman käsityksen siitä, minkä takia

luokka on olemassa. Lisäksi attribuutteihin kerätyt tiedot tekevät ohjelmoinnista paljon helpompaa, kun tekijän ei tarvitse ohjelmakoodia kirjottaessa pähkäillä, mihin muotoon mikäkin attribuutti täytyy tallentaa. Jokaisen attribuutin kohdalla oleva kuvaus auttaa myös hahmottamaan mitä kullakin attribuutilla tarkoitetaan, kuten kuvassa 6 nähdään. Kuvan 6 attribuutit ovat selkeästi nimettyjä, jolloin kuvauksen pystyisi päättelemään nimestäkin, mutta hankalimmissa tapauksissa yhteys ei ole niin selvä. Arvojoukkosarake tekee sovelluksen validoinnista helpomman, kun on suoraan määritelty, mitä arvoja attribuutti voi pitää sisällään. Oletusarvo on taas useimmiten *null*, sillä vaikka tietokantataulukossa arvo ei voisi olla *null*, niin käyttäjän täytyy itse lisätä omat arvonsa kenttiin, sillä ennalta määrättyä arvoa ei voida useassa tapauksessa määrittää. Liitteen 2 role-attribuutissa oletusarvo on urheilija, sillä sovellus on suunnattu urheilijoille, ja tässä tapauksessa oletusarvon määrittäminen on mahdollista.

Attribuutit

nimi	tietotyyppi	pituus	kuvaus	arvojoukko	oletusarvo
Id	Long	1000000	Harjoituksen yksilöivä tunnus	0–9	null
Date	LocalDate	8	Päivä, jolloin harjoitus on tehty	yyyy-mm-dd	null

Kuva 6. Attribuutit

Myös suhteiden, vastuiden, operaatioiden sekä määrätietojen määrittäminen auttaa hahmottamaan luokan tärkeyttä, käytettävyyttä sekä yleisiä vaatimuksia. Operaatiot kohdasta on helppo katsoa vaadittavat metodit, jotta ei tule tehtyä turhaa tai liian vähän työtä, kun taas määrätiedoista saatava hyöty auttaa ohjelmakoodin tehokkuuden parantamiseen. En itse saanut näistä määritelmistä omaan työhön kovin paljoa hyötyä. Ainoat kohdat, jotka pystyin luomaan vaatimusmäärittelyn pohjalta, olivat operaatiot.

Kun vaatimusmäärittely tuli valmiiksi, oli seuraavana edessä itse ohjelman kirjoittaminen. Vaikka vaatimusmäärittelyn aikana oli kova hinku aloittaa ohjelmointi, sain tehtyä koko vaatimusmäärittelyn ennen ohjelmoinnin aloitusta. Vaatimusmäärittely oli auki lähes aina ohjelmaa rakentaessa, kun omia ajatuksia tuli varmistettua dokumentista käsin.

5 Toteutus

5.1 Valmistelut

Ohjelman rakentaminen alkoi ympäristön perustamisella. Koska olin jo aikaisemmin kirjoittanut Javaa omalla tietokoneellani, olin asentanut koneelleni JDK-version 8. Halusin kuitenkin kirjoittaa nykyisen ohjelman versiolla 11, jotta sain muutamia lisäominaisuuksia, kuten esimerkiksi tehokkaampaa muistin käyttöä sekä paremman Docker-tuen. Docker automatisoi sovelluksen julkaisuvaihetta luomalla paketteja, jotta sovellus toimii tehokkaasti eri ympäristöissä, mikä on jatkon kannalta tärkeää. Ehkä tärkeimpänä syynä oli kuitenkin Java 8 -tuen mahdollinen loppuminen lähitulevaisuudessa. Vaikka uusimman Java-version numero on 16, en halunnut hypätä suoraan uusimpaan versioon, koska 64 % Java-ohjelmoijista käyttää vielä Java 8 ja 25 % Java 11 vuonna 2020 (Vermeer 2020.). Lisäksi uusimman version käyttäminen etujoukossa tuntuu tuovan usein haasteita yhteensopivuuden kanssa.

Koska olin kuullut paljon puhetta siitä, että Java 8 on paras versio ohjelmoida Javalla yhteensopivuuteen vedoten, niin päättelin Java 16 olevan epävakaa käyttää. Tämä osoittautui kuitenkin tietämättömyydeksi ja virhearvioksi. Java 16 on vakaa versio, mutta ei pitkän ajan tuen versio eli LTS-versio. Viimeisin LTS-versio Javasta on 11 (Freitag 2020), joten tämä tuki myös päätöstäni tehdä ohjelma JDK 11:sta hyödyntäen. Samalla sain selville syitä, miksi useimmat toimijat eivät halua vaihtaa vanhoista Java versioistaan pois.

Vermeerin (2020) mukaan isoimmat syyt version päivittämättömyyteen ovat:

1. Tämänhetkinen koodi toimii hyvin (51 %)
2. Siirtymisen hinta on liian suuri (35 %)
3. Ei saada yritystä hyväksymään siirtymistä (35 %)
4. Uusissa versioissa ei ole mitään tarvittavia ominaisuuksia (27 %)
5. Jokin muu syy (22 %)
6. Uuden version kadenssi ei sovi (10 %)
7. Uuden version tuen puute (7 %)

Seuraavaksi tuli valita ohjelmointiympäristö, jossa koodia kirjoittaa, eli IDE. Olen aikaisemmin käyttänyt Eclipseä muun muassa koulun tehtävissä ja kotona tehdyissä projekteissa. Muutaman kerran opiskelun aikana törmäsin IntelliJ IDEA:n, joka oli omasta mielestäni ulkoasultaan paljon miellyttävämpi Eclipseen verrattuna, enkä löytänyt syytä, miksi vaihdosta olisi haittaa. Sovelluksesta oli ilmainen ja maksullinen versio. Maksullinen versio maksoi yli sata euroa vuodessa, mutta sain opiskelijana ilmaisen käyttöoikeuden maksulliseen versioon, joka sopi omiin tarpeisiini vallan mainiosti. Maksullisessa versiossa etuna oli paljon hyödyllisiä työkaluja Spring-tuotekehitykseen, jotka olivat iso apu nopeuttaen ohjelman rakentamista huomattavasti. Maksullinen versio pystyi esimerkiksi ennakoimaan

kirjoitettavaa koodia pohjautuen Spring-kehykseen ehdottamalla Springin metodeita sekä yhdistämään Spring-sovelluksen eri osat kokonaisuudeksi.

IDE:n valinnan jälkeen oli aika luoda edellytykset versionhallintaan, jonka hoidin Gitin avulla. Versionhallinnan käyttäminen oli erittäin helppoa IntelliJ IDEA:n kanssa, sillä Gitin sai integroitua IDE:en sisään siten, että ohjelma tunnisti mitä kohtia tiedostoista oli muutettu tai lisätty, minkä jälkeen ohjelmakoodin viimeisimmät versiot oli helppo siirtää järjestelmänhallintatyökaluun ohjelmoinnin yhteydessä. Olen pitänyt kahta eri versioita harjoituspäiväkirjasta: *main* sekä *production*. Lisään valmiin koodin *production*iin ja myöhemmässä vaiheessa vien kokonaisuuksia *main*-haaraan.

Gitin mukana olo motivoi aina valmistamaan osa-alueet loppuun asti. Kun sain siirrettyä valmiin osan koodista Gittiin, niin tiesin saaneeni jotain valmiiksi ja työ liikkui eteenpäin. Samalla toimivat versiot olivat tallessa versionhallinnassa, mikä auttoi mahdollisten ongelmatilanteiden selvittämisessä. Vanhan toimivan version pystyi tarvittaessa hakemaan versionhallinnasta, jos nykyinen versio sisältäisi ohjelman kannalta kriittisiä virheitä.

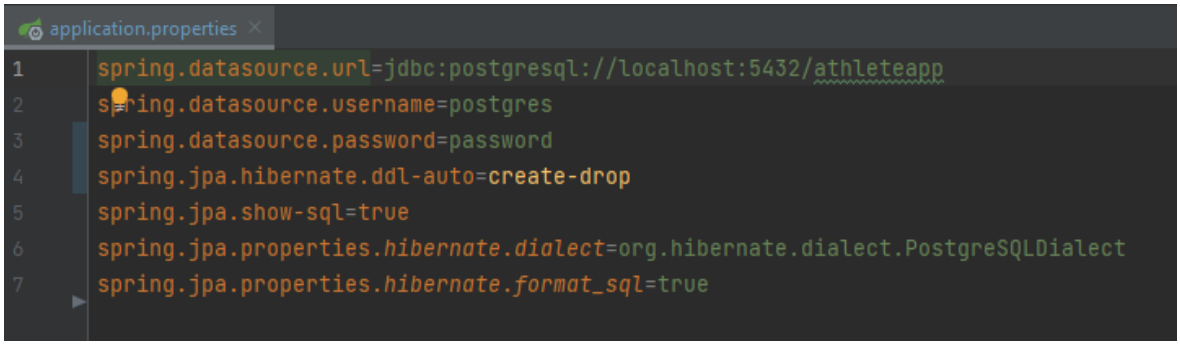
Seuraavaksi täytyi valita sovellukselle tietokanta. Valitsin PostgreSQL-tietokannan, joka oli minulle entuudestaan tuttu ja hyvin toimiva tietokanta. Myös tietokannan sai yhdistettyä IntelliJ IDEA:n käyttöliittymään, josta oli helppo tehdä hakuja tietokantaan ja tarkastella siellä olevia tietoja.

Spring-sovelluksen luominen onnistuu helposti Spring Initialzr web -sovelluksen kautta. Se ei tee ohjelmakoodia, mutta luo Spring-projektin rakenteen, jonka voi lisätä helposti ohjelmointiympäristöön. Initialzr-sovelluksessa pystyy myös lisäämään riippuvuuksia, joita tulee käyttämään sovelluksessa. Esimerkiksi PostgreSQL Driver, jota tarvitaan Java-ohjelman yhdistämiseen Postgre-tietokantaan, voidaan lisätä helposti hiirtä klikkaamalla suoraan sovellukseen ilman erillisiä tiedostojen latauksia useista eri lähteistä.

Nyt oli enää jäljellä Spring-projektin siirtäminen ohjelmistoympäristöön. Se onnistui helposti IntelliJ IDEA:n kautta muutamaa nappia painamalla.

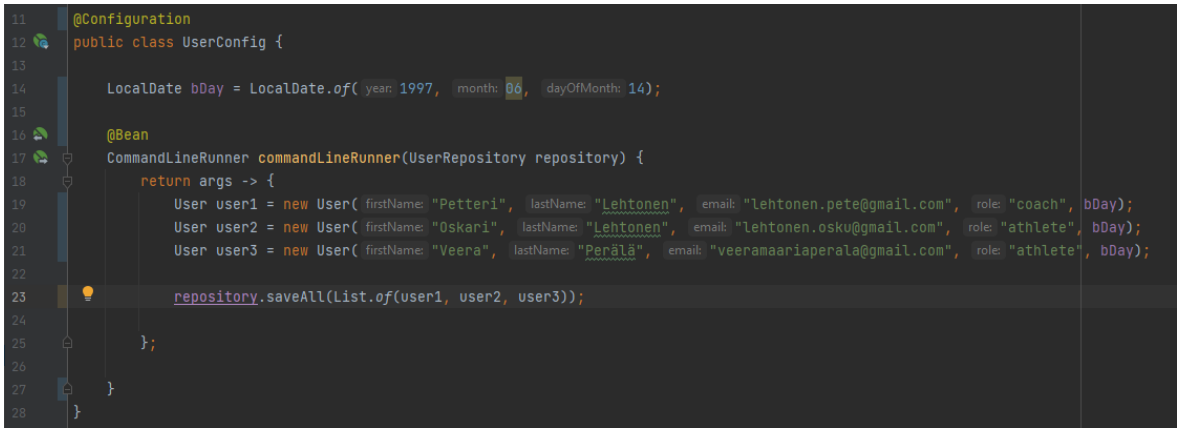
Ennen ohjelmoinnin aloitusta täytyy vielä määritellä *application.properties*-tiedosto. Properties-tiedostoon tallennetaan konfiguroitavia parametreja, jotka vaihtelevat sovelluksen valmistusprosessin eri vaiheissa. Kuvassa 7 nähdään, kuinka kolme ensimmäistä riviä ovat tietokantayhteyden määritystä varten ja neljä seuraavaa ovat sovelluksen ja tietokannan välisiä muita asetuksia. Varsinkin *hibernate.ddl-auto=create-drop* -määritys auttaa sovelluksen kehityksessä, kun aina sovellusta käynnistäessä sovellus poistaa tietokannan

kaikki taulukot ja luo ne uudelleen relaatiokartoitustyökalu Hibernaten avulla. Kaikki tämä tapahtuu automaattisesti sovellusta käynnistäessä. Kyseisen määrityksen avulla sovelluksesta on aina uusin versio toiminnassa myös tietokannan puolelta.



Kuva 7. Sovelluksen määritykset

Halusin poistaa ja luoda tietokantataulukot aina sovelluksen käynnistyessä uudestaan, jotta tietokannan ja sinne siirrettävien tietojen määritykset olisivat aina ajan tasalla. Tämän takia tarvitsin tiedoston, joka myös lisäsi tietoa automaattisesti tietokantaan, jotta minun ei tarvinnut joka kerta käynnistyksen jälkeen itse käydä lisäämässä tietoja tietokantaan. Kuvassa 8 nähdään luokka, joka ajetaan myös jokainen kerta sovelluksen käynnistyttyä.



Kuva 8. Konfigurointi

5.2 Ohjelmointi

Koska halusin tehdä järjestelmän kerrosarkkitehtuurin mukaan, tuli tietojen kulkea sovelluksessa kerroksittain. Kerrosarkkitehtuuri on omasta mielestäni yksinkertainen ja hyvä tapa tehdä sovelluksia sen yksinkertaisen rakenteen vuoksi niin teoriassa kuin myös ohjelmakoodia kirjoittaessa. Kerrosarkkitehtuurin avulla pystyin helposti liikuttamaan dataa sovelluksien eri kerroksien läpi, aina välillä sitä muokaten. Käytin tässä sovelluksessa kuvan 3 mukaista kerrosarkkitehtuuria, jossa data kulkee käyttöliittymän kautta kontrollerille,

sieltä *service*-kerrokseen, josta päästiin *repository*n kautta tietokantaan ja takaisin samaa reittiä.

Olen käyttänyt kommentointia projektissa oman oppimiseni edistämiseksi, vaikka se hieman sotiikin Martinin (2016) hyvien kommenttien käytäntöjä vastaan. Kirjoittamani kommentit ovat toki informatiivisia, varoittavia tai selventäviä, mutta ne ovat ehkä hieman toistavia sekä liikaa informatiiviota sisältäviä, kuten esimerkiksi kuvasta 10 voi nähdä. Halusin kuitenkin helpottaa omaa oppimistani selittämällä asiat hyvin auki, sillä luin tekstiä läpi aina silloin tällöin, kun muokkasin jotain kyseisessä tiedostossa.

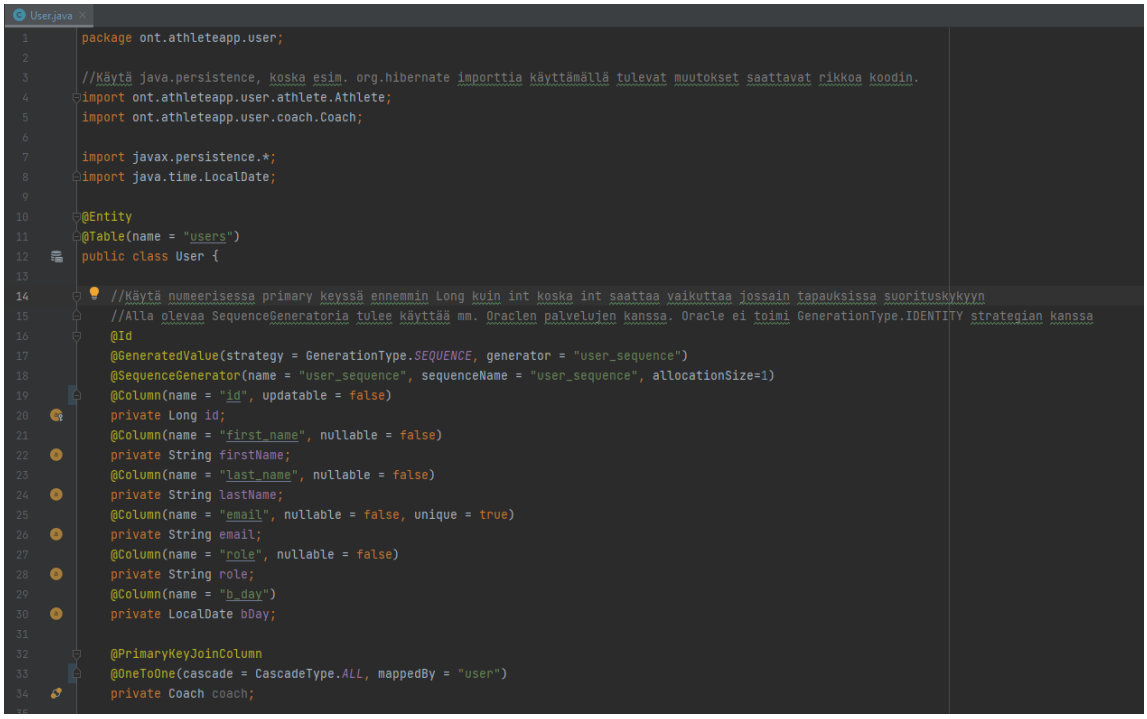
5.2.1 Tietokantaluokat

Ohjelmoinnin aloitin tietokantaluokkien kirjoittamisesta. Aloitin käyttäjistä, sillä se tuntui luonnollisimmalta tavalta aloittaa. Kuvassa 9 on käyttäjäluokan 34 ensimmäistä riviä koodia. Luokka määritellään @Entity-annotaatiolla, jotta JPA on tietoinen kyseisestä luokasta. Normaalissa tapauksessa sovellus löytää luokan nimen kautta yhteyden tietokantataulukoihin. Käyttäjä tapauksessa, nimi *user* on kuitenkin varattu PostgreSQL:n toimesta, minkä takia tietokantataulun nimeä ei suositella samaksi. @Table-annotaatio antaa ohittaa luokalle annetun nimen, jotta sovellus osaa löytää oikean nimisen tietokantataulukon. Tässä tapauksessa nimeksi on annettu *users*.

@SequenceGenerator-annotaatio määrittää *id*-arvon nousun aina, kun uusi käyttäjä luodaan. Ilman kyseistä annotaatiota PostgreSQL nostaa *id*-arvoa 50 yksiköllä, joka on viisikymmentä kertainen verrattuna siihen, että käyttäisi @SequenceGenerator-annotaatiota. Kyseisen annotaation kanssa ensimmäisen käyttäjän *id*-arvo olisi 1 ja seuraavan 2, kun taas ilman sitä ensimmäisen *id*-arvo olisi 1 ja toisen 51. Lisäksi GenerationType.IDENTITY-strategia ei toimi Oraclen tietokantojen kanssa, jos Oraclen versiota ei ole päivitetty versioon 12 C tai uudempaan. Kyseisen strategian on tarkoitus kertoa tietokantapalvelulle, että sen tulee hoitaa *id*-arvon nousu. Tämän takia on turvallisempaa sovelluksen jatkon kannalta, että *id*-arvon päivitys määritellään @SequenceGenerator annotaation avulla.

Tietokantaluokkia kirjoittaessa eteneminen oli suhteellisen suoraviivaista. Pystyin katsomaan suoraan vaatimusmäärittelystä tarvittavat attribuutit ja lisäämään ne luokkaan. Tämän jälkeen tuli lisätä konstruktorit, *get*- ja *set*-metodit sekä *to string* -metodi. Konstruktorit auttavat olioiden luomisessa, kun konstruktori asettaa oikeat arvot kyseisille muuttujille. *Get*-metodi palauttaa haettuja tietoja luokasta, kun taas *set*-metodi lisää asettaa tietoja attribuuteille. *To string* -metodi palauttaa olion string-muotoon kirjoitettuna, jolloin sen tarkasteleminen on helpompaa.

Lisäksi käyttäjällä on yhden suhde yhteen -yhteys valmentaja- ja urheilijaluokkaan, jotta jokainen urheilija ja valmentaja osataan osoittaa käyttäjään. Halusi käyttää myös @PrimaryKeyJoinColumn-annotaatiota, joka luo käyttäjälle sekä urheilijalle/valmentajalle saman *id*-arvon. Tämä tekee tietokannasta siistimmän ja yksinkertaisuuden takia helpottaa työskentelyä tietokannan kanssa.



Kuva 9. Käyttäjäluokka

5.2.2 Repository

Tietokantaluokkien ohjelmoinnin jälkeen oli aika tehdä @Repository-luokka (kuva 10), eli niin sanottu DAO (Data Access Object), joka pitää huolen yhteydestä tietokantaan. Repositorioiden luomisessa on kolme eri repository-vaihtoehtoa, josta valmiita metodeja voidaan ottaa käyttöön. Vaihtoehtoina ovat JpaRepository, jossa on kaikkien muidenkin repositorioiden metodit, PagingAndSortingRepository, jossa on kaikki CrudRepositoryn metodit, mutta ei JpaRepositoryn metodeja, tai CrudRepositoryyn, jossa ei ole muiden kuin oman repositorioiden metodeja. Tämän avulla saadaan käyttöön useita lisäkomentoja, joita kirjastot sisältävät. Käytin sovelluksessa CrudRepositorya, sillä se sisältää kaikki tarvittavat metodit, joita tässä vaiheessa tarvitsin.

Kirjastojen komentojen avulla pystytään luomaan helposti englannin kielellä luettavia kyselyjä. Repository-luokkaan tehtyjä metodien nimet kertovat minkä SQL-kyselyn

repository suorittaa tietokantaan. Kutsu saadaan ajettua yksinkertaisesti kutsumalla luokassa olevia metodeja, joiden parametreina ovat kyselyyn vaadittavat tiedot.

@Query-annotaatio antaa mahdollisuuden tehdä natiivikyselyjä SQL-kielellä. Tämä tulee tarpeeseen monimutkaisemmissa kyselyissä, esimerkiksi kun halutaan liittää useamman taulun tietoja yhteen vastaukseen tietyillä ehdoilla, eli niin sanottu JOIN-kysely. Tässä tapauksessa metodin nimeäminen ei enää vaikuta kyselyyn, vaan repository osaa ajaa @Query-annotaatioon kirjoitetun SQL-lausekkeen. Toinen vaihtoehto edellä mainittua annotaatiota käyttäessä on käyttää JPQL-kyselykieltä. Kyseinen kieli on samantapainen kuin SQL, mutta sen toimintaperiaate perustuu siihen, että se käyttää Java luokkien objekteja muodostaessaan tietokantakyselyjä.

UserRepository.java

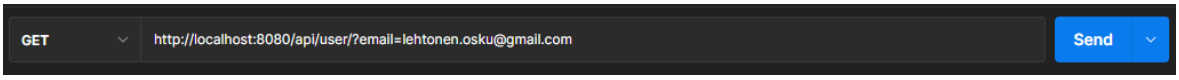
```
1  package ont.athleteapp.user;
2
3  import java.util.Optional;
4
5  import org.springframework.data.jpa.repository.Query;
6  import org.springframework.data.repository.CrudRepository;
7  import org.springframework.stereotype.Repository;
8
9  //@Repository on vastuussa tietokannan yhteydestä(Lisää repository koodiin ja pääset käsiksi tietokantaan)
10 @Repository
11 public interface UserRepository extends CrudRepository<User, Long> {
12
13     /*  Query annotaatio antaa kirjoittaa itse omia JPQL tai natiivi kyselyitä.
14     Tämä ylikirjoittaa metodin nimen, joka ilman annotaatiota osoittaa kyselyn.*/
15
16     /*     Voit käyttää JPQL dokumentaatiosta saatavia avainsanoja (esim And Or LessThan) ja luoda omia
17     kyselyjä ja luoda esim. findUserByFirstNameEqualsAndAgeIsLessThan(String name, Long age)*/
18
19     //    @Query("SELECT s FROM User s WHERE s.email = ?1")
20     Optional<User> findUserByEmail(String email);
21
22 }
23
```

Kuva 10. Repository

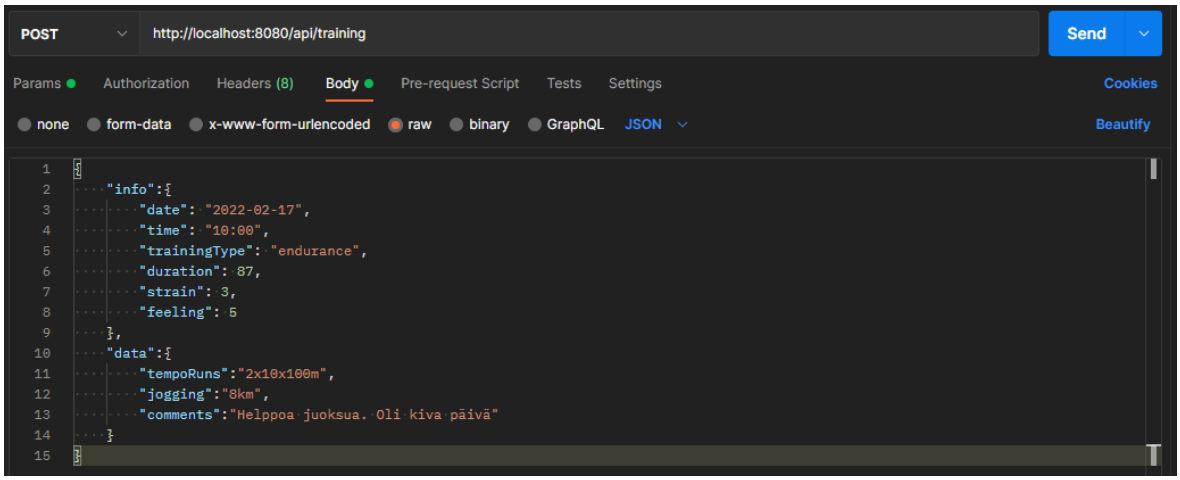
5.2.3 Controller

Controllerin tarkoitus on ohjata tiedonkulkua määrittelemällä URL:n loppuosat, joilla käyttäjä pystyy lähettämään pyyntöjä palvelimelle. Liitteessä 3 nähdään, että määrittely tehdään @RestController-annotaatiolla, joka on esitelty Spring-versiossa 4.0. @RestController-annotaatio eroaa aikaisemmin käytetystä @Controller-annotaatiosta siten, että jokaisessa pyyntöä käsittelevässä metodissa ei tarvitse enää käyttää @ResponseBody-annotaatiota. @ResonseBody-annotaatio määrittää kyseisen metodin. Palautusarvo muodostuu HTTP-pyynnön rungosta. @RestController-annotaation käyttö poistaa lyhyesti sanottuna toistoa koodista.

Muuten controller-luokassa on määritelty @GetMapping-, @PostMapping-, @DeleteMapping- sekä @PutMapping-annotaatiot, jotka määrittävät HTTP-pyynnön tyypin. Käytin käyttäjäluokassa kahta eri tapaa kerätä käyttäjältä tietoja. @RequestBody- sekä @RequestParam-annotaatioita. @RequestBody antaa mahdollisuuden hakea HTTP-pyynnön rungon, josta Spring pystyy poimimaan käyttäjän lähettämän datan JSON-muodossa. @RequestParam taas saa tiedon URI:sta pyynnön mukana tulleesta rungosta. Kuvassa 11 esimerkki @RequestParam käytöstä. Pyyntö käyttää *get*-metodia hakeakseen käyttäjän, jonka sähköpostiosoite on lehtonen.osku@gmail.com. Tässä tapauksessa siirrettävä tieto on nähtävissä URI:ssa. Kuvassa 12 ollaan lisäämässä uutta harjoitusta ja käytössä on @RequestBody-annotaatio. Tällöin URI:ssa ei näy mitään tietoja, vaan kaikki tiedot menevät HTTP-pyynnön rungon mukana. Postman-sovelluksessa HTTP-pyynnön runkoon pystyy lisäämään helposti tietoa testauksen onnistumiseksi.



Kuva 11. @RequestParam



Kuva 12. @RequestBody

Oma kokemukseni oli, että @RequestBodyn käyttö oli helpompaa controller-tasolla sen takia, että pystyin käyttämään samaa metodia, vaikka niistä saatava data olikin erilaista. Tämä täytyi vain ottaa huomioon service-kerroksessa, kun tietoa purettiin JSON-muodosta vaatimusmäärittelyssä määriteltyyn muotoon. @RequestParam käyttö oli taas controller-tasolla hieman jäykempää, mutta service-kerroksessa sen käyttö oli helppoa, kun kaikki pyynnöstä tulleet tiedot olivat tallennettuna jo oikeaan muotoon. Tässä tapauksessa service-tasolla myös tiesi aina mitä tietoa controller-tasolta on tulossa, eikä sitä tarvinnut sen enempää käsitellä.

Sovellusta jatkokehitettäessä tulisi ottaa vain yksi tapa käytäntöön, joko @RequestBody tai @RequestParam. Molempien käyttö samanaikaisesti sovelluksessa on hieman sekavaa ja voi johtaa turhiin virheisiin käyttöliittymän päässä. Kyseistä projektia tehdessä halusin jättää molemmat tavat esille, jotta pystyn esittämään asian jälkikäteen.

5.2.4 Service

Service-kerrokseen sisällytin kaiken bisneslogiikan. @Service-annotaatio määrittelee luokan, jonka tarkoitus on ottaa vastaan controllerista tuleva data ja käsitellä sitä vaadittavalla tavalla. Useimmiten se lähettää käskyn repositoryyn joko hakiessa tai muokatessa tietokannassa olevia tietoja. Luokassa olevat metodit ovat aika yksinkertaisia, eikä Spring-kehyksestä ole tässä vaiheessa service-kerroksessa hyötyä. Poikkeuksena on tietojen päivittäminen. @Transactional-annotaatio, joka nähdään kuvassa 13, muuttaa objektin päivittämistilaan, mikä luo mahdollisuuden hakea tiedot tietokannasta, muuttaa niitä ja lähettää ne muutettuina takaisin.

```
    //@Transactional annotaatio hoitaa tietokannan päivittämisen. Entity menee "manage" tilaan
    @Transactional
    public void updateUser(Long userId, String email, String firstName, String lastName, String role, LocalDate bDay) {
        User user = userRepository.findById(userId).orElseThrow(() -> new IllegalStateException("User with id " + userId + " does not exist"));

            if(firstName != null){
                user.setFirstName(firstName);
            }
            if(lastName != null){
                user.setLastName(lastName);
            }
            if(role != null){
                user.setRole(role);
            }
            if(bDay != null){
                user.setbDay(bDay);
            }

        if(email != null && email.length() > 0 && !Objects.equals(user.getEmail(), email)) {
            Optional<User> userOptional = userRepository.findUserByEmail(email);
            if(userOptional.isPresent()) {
                throw new IllegalStateException("Email already used");
            }
                user.setEmail(email);
        }
    }

}
```

Kuva 13. Päivitä käyttäjä

Kuvassa 14 on esimerkki käyttäjän tallentamisesta tietokantaan. Metodin parametrina on controller-luokasta saatu JSON-data, joka pystytään muuttamaan oikeaan muotoon ja tallentamaan tietokantaan. Tässä vaiheessa tehdään datalle vaadittavat validoinnit, jotta tietokantaan menevät tiedot ovat oikeita, eikä mitään, mikä voisi rikkoa sovelluksen.

Service-kerros on mielestäni paikka, missä pystyy helposti lyhentämään kirjoitetun koodin määrää. Luokkaan tulee kirjoitettua paljon omia ideoita, eikä luokassa pysty noudattamaan suoria ohjeita kuten esimerkiksi tietokantaluokkien konstruktoreja, *get*- sekä *set*-

metodeita tehdessä. Monta kertaa service-luokkaa kirjottaessa huomaa, että omaan bisnes logiikkaan ei löydykään niin suoraa vastausta, kun haluaisi, ja tämän takia joutuu soveltamaan ja tekemään koodista pidemmän ja monimutkaisemman, vaikka saman lopputuloksen olisi voinut mahdollisesti tehdä muutamalla rivillä.

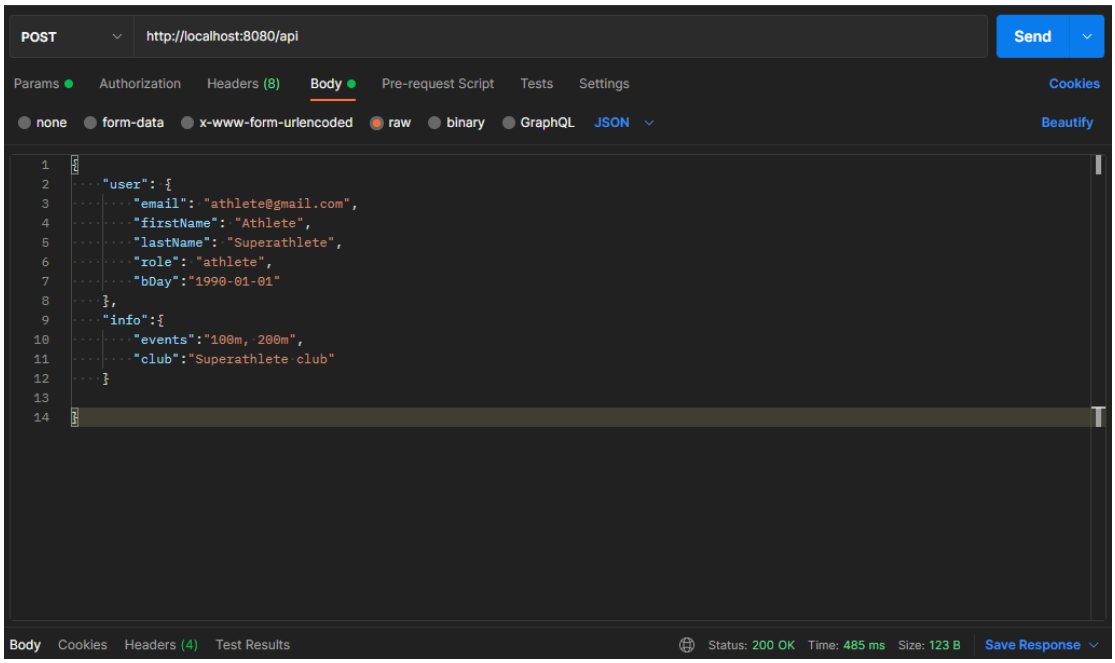
```
40  @   public void addNewUser(ObjectNode json){
41      //      Tallennetaan jsonista haetut tiedot oikeaan muotoon
42              JsonNode userData = json.get("user");
43              JsonNode infoData = json.get("info");
44
45              String firstName = userData.get("firstName").asText();
46              String lastName = userData.get("lastName").asText();
47              String email = userData.get("email").asText();
48              String role = userData.get("role").asText();
49              LocalDate bDay = LocalDate.parse(userData.get("bDay").asText());
50
51      //      Tarkistetaan onko sähköpostilla tehty jo käyttäjää
52              Optional<User> userByEmail = userRepository.findUserByEmail(email);
53              if(userByEmail.isPresent()) {
54                  throw new IllegalStateException("Email taken");
55              }
56
57              User user = new User(firstName, lastName, email, role, bDay);
58
59      //      Tarkistetaan onko lisättävä käyttäjä urheilija vai valmentaja ja hoidetaan tietokantatallennus
60              if(role.equals("athlete")){
61                  String aEvents = infoData.get("events").asText();
62                  String club = infoData.get("club").asText();
63                  Athlete athlete = new Athlete(aEvents, club, user);
64                  athleteRepository.save(athlete);
65              }else if(role.equals("coach")){
66                  String cEvents = infoData.get("events").asText();
67                  String degree = infoData.get("degree").asText();
68                  Coach coach = new Coach(cEvents, degree, user);
69                  coachRepository.save(coach);
70              }else{
71                  throw new IllegalStateException("Ei valmentaja eikä urjeilija");
72              }
73
74      }
```

Kuva 14. Lisää käyttäjä -metodi

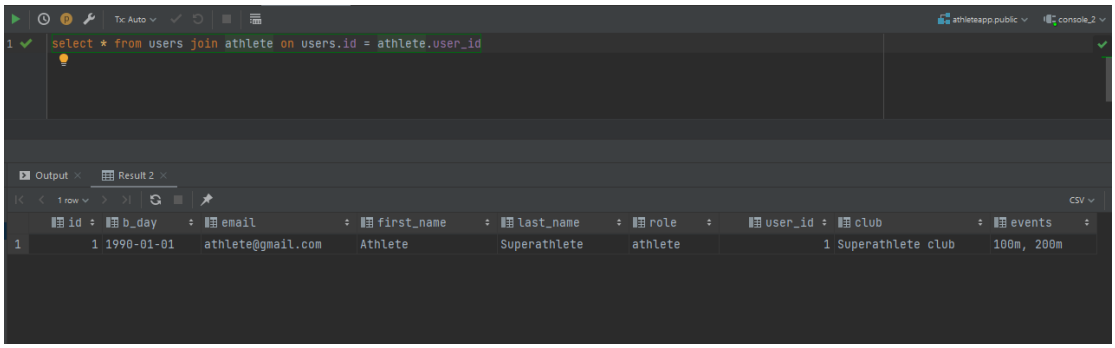
5.2.5 Testaus

Kun käyttäjälle on luotu tietokantaluokka, repository-, controller- ja service-kerros, on aika testata luotujen ominaisuuksien toimivuutta. Koska käyttöliittymää ei ole rakennettu, tulee sovellusta testata palveluilla, jotka suorittavat HTTP-pyyntöjä. Käytin tässä tapauksessa Postman-sovellusta. Kuvassa 15 olen lisännyt URL-osoitteen controller-luokassa määrittämäni polun mukaan. Tämän jälkeen olen määritellyt pyynnön mukana lähetettävän datan, joka on käyttäjää lisätessä JSON-muotoinen. Valmistelujen jälkeen pyyntö lähetetään ja vastauksesi saadaan jokin statuskoodi, tässä tapauksessa 200, mikä tarkoittaa, että pyyntö on mennyt läpi ilman ongelmia. Sovelluksen toiminta voidaan myös tarkistaa IDE:n puolelta, sillä properties-tiedostossa on määritelty spring.jpa.show-sql=true, mikä tarkoittaa, että IDE:n konsolissa näkyy kaikki SQL-lauseet, joita sovellus suorittaa. Tämä on näkyvillä liitteessä 4.

Viimeinen varmistus tulee kuitenkin tehdä itse tietokannasta. Kuvassa 16 on tehty SQL-kysely, missä yhdistetään käyttäjän tiedot sekä käyttäjätaulusta, sekä urheilijataulusta. Tulokseksi saadaan yhden rivin vastaus, missä on kaikki tiedot käyttäjältä. Vasta nyt voimme olla varmoja, että sovelluksen yritys tallentaa tiedot tietokantaan on onnistunut.



Kuva 15. Käyttäjän lisäys



Kuva 16. SQL-kysely

5.2.6 Kansiorakenne

Kun sovellus kasvaa suureksi, on hyvä pitää sovelluksen kansiot siistinä ja järjesteltyinä. Tämä helpottaa oikeiden tiedostojen löytämistä, kun ei tarvitse selata koko projektia läpi etsiessään yhtä tiedostoa. Vaikka IDE:n hakutoiminnot ovat nykyään hyviä, on myös jatkon kannalta hyvä, että tiedostot ovat niille kuuluvilla paikoilla. Vaikka kansiorakenteen tekemiseen onkin monia vaihtoehtoja, on liitteessä 5 tämän projektin rakenne, joka on mielestäni kaikista paras tapa tehdä se.

Toinen tapa voisi olla siten, että jokaiselle controllerille, repositorylle sekä servicelle tehtäisiin omat kansiot, ja samantyyppiset tiedostot sijaitsisivat kaikki samassa paikassa. Tämä tyyli ei mielestäni sopisi ainakaan tähän projektiin, sillä eteen saattaisi tulla tilanne, jolloin etsisin kansiosta urheilijan service-tiedostoa, vaikka sellaista tiedostoa ei olekaan. Tämän tyylinen kansion rakenne ei kerro mielestäni tarpeeksi sovelluksen rakenteesta, vaikka se pitääkin saman nimiset tiedostot samassa paikassa.

Liitteessä 5 jokainen aihealue on pidetty omissa kansioissaan ja esimerkiksi käyttäjään liittyvät tiedostot sisältävät myös valmentajan ja urheilijan tiedostot. *Valmentaja* ja *urheilija* ovat lisäksi omissa kansioissaan, jotta rakenne olisi yksinkertaisempi. Koska valmentaja ja urheilija käyttävät osittain samoja tiedostoja kuin käyttäjä (controller- ja service-luokat), on kansioiden sisällä vain kaksi tiedostoa. Tätä kansiorakennetta katsoessa tajuaa heti, että urheilija sekä valmentaja käyttävät käyttäjäluokan controller- ja service-tiedostoja. Olen toteuttanut samalla periaatteella myös harjoituksiin liittyvän rakenteen.

6 Omat havainnot

Vaikka en saanut projektityöstä muiden ihmisten kanssakäymisestä saatavaa hyötyä, sain kuitenkin itse johdettua itseäni läpi koko projektin. Isoissa yrityksissä on tärkeää muiden ihmisten välinen kanssakäyminen. Kanssakäyminen on kuitenkin enemmän yksilöistä ja persoonista kiinni, joten kokemus ei siinä ole niin isossa osassa. Mayukon (2019) mukaan yhä useammat yritykset arvioivat työhaastattelussa hakijan soveltuvuutta kommunikaatiotaidoissa. Kommunikaatiotaitoja voi harjoitella niin monessa eri tilanteessa, että ne eivät vaadi varsinaista projektia ympärilleen, jotta niitä voisi kehittää. Tietysti projektiympäristö luo oman ympäristönsä ja haasteensa kommunikaatiolle, kun stressi ja kiire painavat päälle luoden mahdollisesti kireän ilmapiirin. Itse työhaastattelu on jo kuitenkin sen verran stressaava tilanne, jossa kyseiset heikkoudet saattaisivat paljastua.

Koska olin yksin projektissa, aikataulun luominen, suunnittelu ja toteutus olivat kaikki omalla vastuullani. Aikataulun luominen onnistui hyvin, kun avasin kalenterin ja suunnittelin projektille lopetusajankohdan, jonka jälkeen kokonaisuuden pystyi pilkkomaan pienempiin osiin. Aikataulun luomisen jälkeen oli helppoa myös tarkastaa aikataulusta mitä aihetta tulisi minäkin ajankohtana tehdä. Tämä auttoi unohtamaan ison kokonaisuuden, jota olisi voinut muuten ajatella isona mörkönä. Jos kalenterissa luki, että tehdään vaatimusmäärittelyä, silloin tehtiin vaatimusmäärittelyä. Jos siellä luki ohjelmointia, silloin kirjoitettiin koodia. Aikataulun suunnitteleminen oli mielestäni suhteellisen helppoa ja vaivatonta. Se oli myös henkisesti helpottavaa, sillä itselleni tuli jokaista osa aluetta järjesteltäessä tunne, että kyseinen asia olisi jo lähes tehty. Sanonta ”hyvin suunniteltu on puoliksi tehty” sopii tähän erittäin hyvin.

Vaatimusmäärittelyn teko oli huomattavasti muuta suunnittelua haastavampaa. Vaikka omat ideat olivat selkeinä päässä, haasteet tulivat silloin, kun ne piti kirjoittaa ylös puhtaaksi. Selkeät ajatukset muuttuivatkin hieman vähemmän selkeiksi. Vaikka ohjelman määritykset vaikuttavat yksinkertaisilta, ne vaativat yllättävän paljon komponentteja koodin puolella. Pelkästään CRUD- ja REST-toimintojen – eli sovelluksen minimivaatimuksien – lisääminen vei ison osan työn ajankäytöstä. Kun siihen päälle lisättiin vielä bisneslogiikka ja muutama hankalampi metodi, niin kokonaisuus kasvoi yllättävän suureksi.

Vaatimusmäärittelyn tekeminen onnistui kuitenkin hyvin, vaikka se olikin päänvaivaa aiheuttavaa puuhaa. Kun sain yksitellen lisättyä ominaisuuksia paperille, sain pikkuhiljaa käsityksen siitä, millainen ohjelmasta oli tulossa. Vaatimusmäärittelyn valmiiksi saaminen oli helpottavaa ja ajatus siitä, että siihen ei enää tarvitse koskea kävi mielessä. Toisin kuitenkin kävi, kun ohjelmoitaessa tuli huomattua, että jotkin arvot olivat määritelty väärin

vaatimusmäärittelyssä. Jouduin muutamaan kertaan muokkaamaan vaatimusmäärittelyä ohjelmoinnin aikana hyppimällä edes takaisin koodin ja vaatimusmäärittelyn välillä. Tuli vääjäämättäkin mieleen ketterät menetelmät ja niiden tuomat edut ohjelmoinnissa, joita tässä tapauksessa tuli osin hyödynnettyä.

Projektitiimi olisi voinut olla vaatimusmäärittelyä tehdessä paljon hyödyksi, kun olisi yhdessä mietitty ohjelmiston ominaisuuksia ja täten vältytty mahdollisesti osasta virheistä, joita vaatimusmäärittely sisälsi. Yksin tehdessä saattaa juuttua omiin ajatuksiinsa eikä näe asiaa niin avoimesti kuin olisi tarve. Ihminen saattaa helposti täyttää omat tietämättömyytensä aukot omilla teorioillaan, eikä pysty täten erottamaan mikä on oikein ja mikä väärin ilman kriittistä ajattelua. Projekteissa, joissa liikutaan kovaa vauhtia eteenpäin, tulee kriittisestä ajattelusta yhä vaikeampaa, kun jokaiseen päätökseen käytettävä aika on rajallinen ja päätöksiä on paljon. Jos apuna olisi projektiryhmä, tulisi jokainen kyseenalaistamaan toisten ajatuksia ja tarjoamaan parempia vaihtoehtoja, mikä johtaisi parempaan lopputulokseen.

Ohjelmointiympäristön luominen onnistui helposti. Olen useaan otteeseen kyseisen asian tehnyt, joten kovin paljoa uutta ei tämän saralla ollut. Ainoa asia mihin paneuduin hieman enemmän, oli Java-version valinta. Olin kuullut paljon puhetta eri tahoilta, että Java 8 on se versio, jolla ohjelmia tulisi rakentaa. Aloin kuitenkin selvittämään asiaa, sillä ei kuulosta kovin loogiselta, että tehtäisiin uusia versioita, joita ei kannattaisi käyttää. Hyvin nopeasti selvisikin, että suurin osa siirtymää vastustavista, olikin isoja yrityksiä, joilla oli valmiiksi rakennetut ohjelmistot Java 8. versiota käyttäen. Siirtymä uudempiin versioihin olisi kallista ja hankalaa. Omaksi valinnaksi kohdistunut Java 11 oli siinä vaiheessa mielestäni seuraava luonnollinen valinta, sillä puhtaalta pöydältä aloittaessa ei tarvitsisi miettiä kyseisiä ongelmia.

Laadukkaan koodin määritelmiä on varmasti yhtä monta kuin on ohjelmoijia. Pääpiirteittäin laadukas koodi on alan ammattilaisten mielestä hyvin jäsenneltyä, toimivaa, tehokasta, hyvin nimettyä sekä minimaalista (Martin 2016.). Yhtä kaavaa ei siis ole, vaan kokonaisuuden täytyy olla hallinnassa niin projektin valmistautumisesta itse koodin kirjoittamiseen. Tämä kuvastaa omasta mielestäni todella hyvin laadukasta koodia. Ohjelmointia on jopa verrattu taiteeseen, joka kertoo mielestäni siitä, että liian tarkkoja linjoja ei pystytä vetämään siitä, mikä on oikein ja mikä väärin. Uskon, että tulee aina olemaan useampia koulukuntia, jotka tykkäävät tehdä asioita omalla tavallaan. Asiaa voi mielestäni verrata sisustustyyleihin. Vaikka perusteet ovat samat (pöytä, lipasto, matto) niin jokainen voi sisustaa omalla tyylillään (väri, materiaali, design). Jokaisella on oma makunsa, minkä takia hyvän lopputuloksen voi saada usealla eri tyylillä. Mutta jos ruokapöytä on aseteltu

makuuhuoneeseen, ei lopputulos ole kovin järkevä, vaikka se käytännössä olisikin toimiva. Sama perustuu ohjelmointiin, ei toki niin jyrkästi. Jos näytät työhaastattelussa koodia, joka on sekavaa, sisältää outoja kommentteja ja on epäloogista, niin työn saaminen on aika haastavaa. Huonolla koodilla ja omalla tyylillä on selvä ero ja työhaastattelija varmasti huomaa ne.

Olin oppinut koulussa omat tapani ohjelmoida, ja ne myötäilivät ohjelmoinnin hyviä tapoja lähes järjestäen. Tiedostojen koot eivät lähteneet rönsyilemään, koodi oli hyvin sisennettyä, järjesteltyä sekä loogisesti nimettyä.

Kommentointi oli ainoa asia, missä täytyi hieman miettiä parasta tapaa tehdä se. Kommenttien tulisi olla ytimekkäitä, eikä niitä tulisi laittaa kohtiin, missä metodit ovat helposti ymmärrettävät. Toisaalta, jos kommenteissa täytyy selittää todella monimutkaista metodia, herää kysymys, olisiko itse metodin voinut kirjoittaa paremmin siten, ettei kommentointia tarvitsisi. Tasapainoilin siis kommentoinnin ja kommentoimattomuuden välillä. Päätin kommentoida hankalaksi huomaaviani asioita häikäilemättä, jotta oma oppimiseni olisi mahdollisimman tehokasta.

Ohjelmointikielten opettelussa mielestäni tärkeintä on osata ajatella kuin ohjelmoija. Olen ajan saatossa oppinut, että lähes millä tahansa ohjelmointikielellä pystyy rakentamaan lähes minkä tahansa ohjelman. Ennen opintojeni alkua, kun suunnittelin lähteväni tälle alalle, ajattelin, että tietyillä ohjelmointikielillä voi tehdä vain tiettyjä ohjelmia. Vaikka ohjelmointikielillä suoritettavat tehtävät ovatkin jakautuneet karkeisiin kategorioihin, Javalla voi rakentaa esimerkiksi laskimen, verkkokaupan tai kännykkäpelin. Tärkeintä mielestäni on ollut huomata, kuinka eri ohjelmointikielillä on samantapainen logiikka. Syntaksi on erilainen joka kielellä, mutta ohjelmoinnin perusperiaate on lähes sama. On ollut iso hyöty ymmärtää tämä asia, sillä muiden ohjelmointikielten lukeminen on helpottunut huomattavasti, kun muilla kielillä kirjoitettu koodi ei ole vain kasa tekstiä.

Kokonaisuudessaan tekemäni projekti onnistui lähtökohtiin nähden hyvin. Onnistuin luomaan kokonaisuuden, joka oli sekä realistinen että yhtenäinen kokonaisuus. Ainut osa-alue, jolla koin epäonnistuvani, oli opinnäytetyön aikataulutus. Aikataulut venyivät kesän aikana turhan pitkiksi, minkä takia työn valmistuminen viivästyi. Suunnitellessani projektia määrittelin suurimmiksi haasteiksi aikataulun, ongelmiin jumiin jäämisen sekä kunnianhimoisuuden. Aikataulu venyi isoksi osaksi muun elämän luomien kiireiden vuoksi, ei niinkään ongelmanratkaisun tai kunnianhimon takia, toisin aluksi olin ennustanut.

Suurimmaksi ongelmaksi yksityiselämän ja työn yhdistämisessä tuli ohjelmointiympäristön sijainti. Olin luonut ohjelmointiympäristöni kotiin pöytäkoneelle, mikä vähensi ohjelmoimiseen käytettävää aikaa huomattavasti, sillä vietin kesän aikana enemmän aikaa tien päällä kuin kotona. Jatkon kannalta on tärkeää luoda ohjelmointiympäristö pilveen tai kannettavalle tietokoneelle. Kannettavan tietokoneen edut tulevat siinä, että kaikki asennetut ohjelmat sekä koneen sisällä muutetut asetukset ovat aina siellä missä itsekin on. Kannettavia tietokoneita voi liikutella helposti mukanaan kokouksiin, joissa voi tehdä muistiinpanoja tai heijastaa omia töitä helposti muiden nähtäville. Nykypäivän hyvät kannettavat tietokoneet ovat jo niin tehokkaita, että niitä voi hyvin käyttää päätyöasemana ilman, että tarvitsee murehtia suorituskyvystä. Kannettavan liittäminen lisänäyttöihin poistaa ongelman, joka liittyy koneessa olevaan yhteen pieneen näyttöön. Jos töitä tulee tehtyä liikkeessä, niin yksi pieni ruutu riittää, mutta jos työpaikalla tai kotona tulee vietettyä enemmän aikaa, on varsinkin ohjelmoitaessa useamman näytön asetelmasta iso hyöty. Tällöin voi pitää useamman ikkunan auki samaan aikaan, jotta pystyy toimimaan ilman jatkuvaa ikkunoiden sulkemista ja avaamista, mikä nopeuttaa työntekoa ja tekee työstä huomattavasti mukavampaa.

Jos tekisin saman projektin uudestaan, tekisin monta asiaa eri tavalla. En välttämättä loisi uusia rutiineja, vaan poistaisin välistä turhia ja aikaa vieviä yksityiskohtia. Työ kun työ on kuitenkin kokonaisuuden oppimista, ja lähes jokaisella alalla kokemusta saa vain työtä tekemällä. Vaikka olisi kuinka valmistautunut, ulkopuoliset muuttujat tuovat niin paljon haasteita, että vasta kokemuksen tuoman hyödyn kautta pystyy reagoimaan niihin nopeammin sekä tehokkaammin. Nyt olen itse yhtä kokemusta rikkaampi. Työn ohjelmakoodi löytyy GitHubistani (Koodikonna666).

Lähteet

Agile Education Research. 2019. Web-palvelinohjelmointi Java syksy 2019. Luettavissa: https://web-palvelinohjelmointi-s19.mooc.fi/osa-4/3-sovelluksen-rakenne. Luettu: 1.8.2021.

Altvater, A. 2017. What are CRUD Operations: How CRUD Operations Work, Examples, Tutorials & More. Luettavissa: https://stackify.com/what-are-crud-operations/. Luettu 13.9.2021.

Dagmar. 2015. Mitä markkinoijan tulee ymmärtää web-ohjelmoinnista. Luettavissa: https://www.dagmar.fi/verkkopalvelukehitys/mita-markkinoijan-tulee-ymmartaa-web-ohjelmoinnista/. Luettu: 13.9.2021.

Docker s.a. Luettavissa: https://www.docker.com. Luettu:15.10.2021.

Freitag, P. 2020. Java LTS Version Roadmap and Guide. Luettavissa: https://www.petefreitag.com/item/911.cfm. Luettu: 13.9.2021.

FutureLearn. 2021. What are different programming languages used for? Luettavissa: https://www.futurelearn.com/info/blog/what-are-different-programming-languages-used-for. Luettu: 13.9.2021.

GeeksforGeeks. 2020. Annotations in Java. Luettavissa: https://www.geeksforgeeks.org/annotations-in-java/. Luettu: 17.10.2021.

Git s.a. Luettavissa: https://git-scm.com/. Luettu: 13.9.2021.

Johansson, T. 2019. Ohjelmistoympäristö yleisesti. Luettavissa: https://tarjotin.cs.aalto.fi/guides/pycharm/python-asennus/ide_fi.html. Luettu: 13.9.2021.

JSON s.a. Luettavissa: https://www.json.org/json-en.html. Luettu: 24.9.2020.

Jämsen, P. 2016. Mitä ohjelmointi on? Luettavissa: https://peda.net/p/jamspe/omat-atk-t/ohjelmointi/moo2. Luettu: 13.9.2021.

Karjalainen, S. 2008. Prosessimallit ja ohjelmistoprojektin onnistuminen toimittajan näkökulmasta. Luettavissa: https://trepo.tuni.fi/bitstream/handle/10024/79507/gradu02960.pdf?sequence=1&isAllowed=y. Luettu: 8.4.2021.

Kasala. J. 2016. Ohjelmistotuotannon prosessit ja menetelmät. Luettavissa: https://www.theseus.fi/bitstream/handle/10024/106418/Kasala_Jussi.pdf?sequence=1&isAllowed=y. Luettu: 19.10.2021.

Kauppakamari s.a. Mikko Mäntyneva. Luettavissa: https://www.kauppakamarikauppa.fi/collections/mikko-mantyneva. Luettu: 14.10.2021.

Koodikonna666. ont.athleteapp. Luettavissa: https://github.com/Koodikonna666/ont.athleteapp.

Laaksonen, A. 2020. Mikä on tietokanta? Luettavissa: https://tikape-k20.mooc.fi/luku-1/1. Luettu: 13.9.2021.

Mallawaarachchi, V. 2017. Luettavissa: https://towardsdatascience.com/10-common-software-architectural-patterns-in-a-nutshell-a0b47a1e9013?gi=e0ec1ecab172. Luettu: 1.8.2021.

Mayuko. 2019. Communication Skills: A Core Part of Software Engineering. Luettavissa: https://betterprogramming.pub/communication-skills-a-core-part-of-software-engineering-c7d379cebd66. Luettu: 8.4.2021.

Martin, R. 2016. Clean Code. O´Reilly. E-kirja. Luettu: 8.4.2021.

McConnell, S. 1993. Code Complete 2. Kindle. E-kirja. Luettu: 8.4.2021.

Mäntyneva, M. 2016. Hallittu projekti - Jäntevästä suunnittelusta menestykselliseen toteutukseen. Helsingin Kamari Oy. E-kirja. Luettu: 8.4.2021.

Oracle s.a. Luettavissa: https://docs.oracle.com/javaee/6/tutorial/doc/bnbpz.html . Luettu: 24.9.2021.

Oracle. 2011. Annotation OneToOne. Luettavissa: https://docs.oracle.com/javaee/6/api/javax/persistence/OneToOne.html. Luettu: 24.9.2021.

Oracle. 1999. Naming Conventions. Luettavissa: https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html. Luettu. 16.10.2021.

Paakki, J. 2011. Ohjelmistojen vaatimusmäärittely. Luettavissa: https://www.cs.helsinki.fi/u/paakki/Vaatimus-11-Luentokalvot-4.pdf.%20. Luettu: 1.8.2021.

Räisänen, J. 2013. MVC-mallin mukainen web kehitys JavaScriptillä. Luettavissa: https://www.theseus.fi/bitstream/handle/10024/64770/inssityo_raisanen_final_v6.pdf?sequence=1. Luettu: 31.8.2021.

Ruuska, T. 2012. Vaatimusmäärittely ketterässä ohjelmistokehityksessä. Luettavissa: https://jyx.jyu.fi/bitstream/handle/123456789/38590/URN:NBN:fi:jyu-201209202463.pdf?sequence=1. Luettu: 1.8.2021.

Sourour, Bill. 2017. Putting comments in code: the good, the bad, and the ugly. Luettavissa: https://www.freecodecamp.org/news/code-comments-the-good-the-bad-and-the-ugly-be9cc65fbf83/. Luettu: 17.10.2021.

Tyson, M. 2020. What is the JDK? Introducing to the Java Development Kit. Luettavissa: https://www.infoworld.com/article/3296360/what-is-the-jdk-introduction-to-the-java-development-kit.html. Luettu: 13.9.2021.

Tverin, T. 2007. Arkkitehtuurituurityyli ohjelmarakenteen perustana. Luettavissa: https://www.cs.helsinki.fi/u/ttverin/tutkielma_ttverin.pdf. Luettu: 31.8.2021.

Vanhatapio, J. 2020. Mikä on HTTP? Luettavissa: https://www.zoner.fi/mika-on-http/. Luettu: 13.9.2021.

Vermeer, B. 2020. 64 % of developers report that Java 8 remains the most often used release. Luettavissa: https://snyk.io/blog/developers-dont-want-to-leave-java-8-as-64-hold-firm-on-their-preferred-release/. Luettu: 13.9.2021.

Liitteet

Liite 1. Vaatimusmäärittely

ICT-järjestelmän määritys

Harjoituspäiväkirja

Versio 1.0.0
Tekijät Oskari Lehtonen
1.8.2021

SISÄLTÖ

Oskari Lehtonen

1 Johdanto

Dokumentaatio on lisänä allekirjoittaneen opinnäytetyötä. Opinnäytetyön aiheena on tehdä Java web projektina harjoituspäiväkirja. Opinnäytetyössä keskitytään back-end koodiin, joten tämä vaatimusmäärittely poikkeaa hieman opinnäytetyöhön tehdystä sovelluksesta.

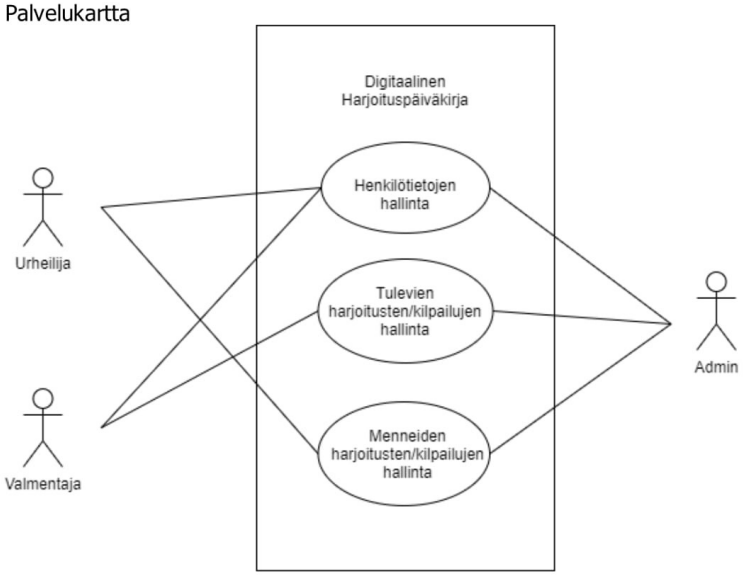
Dokumentaation tarkoituksena on kuvata harjoituspäiväkirjan vaatimukset, mitä tämän luomisessa, käytössä ja ylläpidossa tulee ottaa huomioon. Dokumentaatio on nopeasti silmäiltävissä, jotta lukija saa helposti riittävän kuvan järjestelmän toimivuuden vaatimuksista. Lisäksi lukija ymmärtää käyttäjien välisiä riippuvuuksia ja riippumattomuuksia.

2 ICT-systeemin yleiskuvaus

ICT-järjestelmän on tarkoitus helpottaa pikajuoksua harrastavien valmentajien ja urheilijoiden harjoitusten dokumentaatiota.

- Käyttäjä(urheilija) pystyy tarkastelemaan, lisäämään, poistamaan sekä muokkaamaan lisäämiään harjoituksia.
- Käyttäjä(valmentaja) pystyy tarkastelemaan, lisäämään, poistamaan sekä muokkaamaan tulevia harjoituksia.
- Admin hallinnoi kaikkia käyttäjätietoja.

Palvelukartta



2.1 Palvelukuvaukset

Urheilija sekä valmentaja voivat muokata omia henkilötietojaan. Ongelmatapauksissa myös admin pystyy korjaamaan käyttäjän henkilötietoja tämän pyytäessä.

Urheilija voi lisätä tehdyn harjoituksen, jonka jälkeen harjoitus on vielä vapaasti muokattavissa. Valmentaja voi lisätä urheilijalle harjoituksia, joita urheilija pystyy tarkastelemaan omalta tililtään. Urheilija pystyy kopioimaan määrätyn harjoittelun suoritukseksi päällekkäisen työn tekemisen estämiseksi. Urheilija pystyy vapaasti kirjoittamaan kommentteja omaan harjoitukseensa liittyen.

Admin hoitaa rekisteröinnissä sekä palvelun käy.

2.2 Toimijat

Urheilija on palveluun rekisteröitynyt käyttäjä, joka on halukas dokumentoimaan tekemiään harjoituksia.

Valmentaja on palveluun rekisteröitynyt käyttäjä, jolla on myös valmennettava urheilija rekisteröityneenä palveluun.

Admin ylläpitää kaikkia käyttäjätietoja, ja hoitaa ongelmatapauksia.

2.3 Yhteenveto käyttöoikeuksista

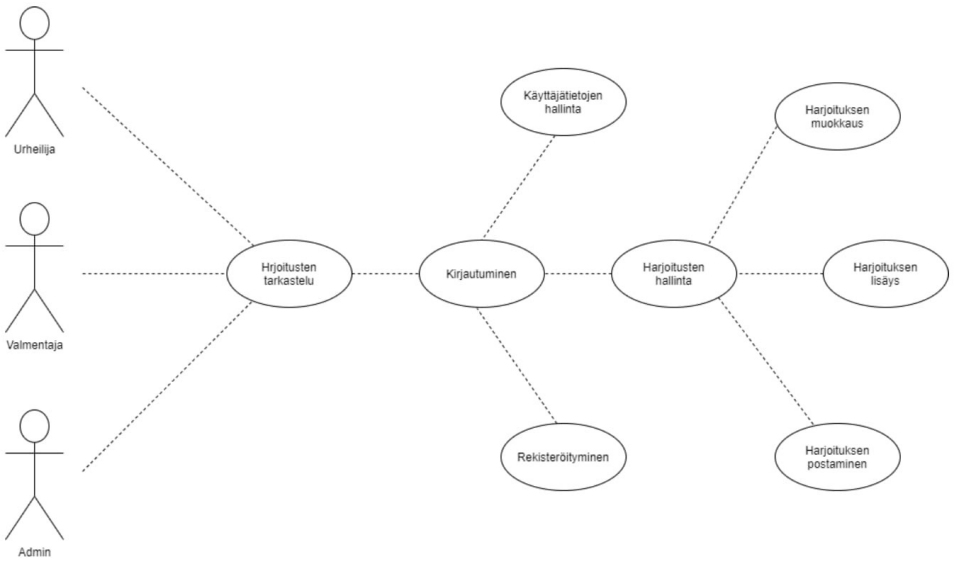
Taulukko 1. Käyttöoikeudet

toimija	Henkilötietojen hallinta	Tulevien harjoitusten /kilpailujen hallinta	Menneiden harjoitusten /kilpailujen hallinta
Urheilija	x		x
Valmentaja	x	x	
Admin	x	x	x

3 Käyttötapaukset

3.1 Käyttötapausten väliset riippuvuudet

Kuva 2. Käyttötapausten riippuvuuskaavio



3.1.n Käyttötapauksen kuvaus

Rekisteröityminen
Toimija(t) Urheilija, Valmentaja
Esiehto Voimassa oleva sähköpostiosoite
Lopputulos Käyttäjä saadaan tallennettua järjestelmän tietokantaan.
Käyttötiheys Päivittäin, viikoittain, kuukausittain.

1. Käyttäjällä on pääsy internettiin
2. Käyttäjä avaa applikaation selaimessa
3. Käyttäjä valitsee ”rekisteröidy”
4. Käyttäjälle tulee rekisteröitymis- kaavake näytölle, johon syöttää omat tietonsa
5. Käyttäjä tallentaa rekisteröitymisen ja saa käyttöoikeuden järjestelmään

Virhetilanteet: ”Tämä sähköposti on jo käytössä” – Asiakas on jo kirjautunut järjestelmään kyseisellä sähköpostilla.

Kirjautuminen
Toimija(t) Urheilija, Valmentaja, Admin
Esiehto Käyttäjän tulee olla rekisteröitynyt järjestelmään.
Lopputulos Käyttäjä pääsee palveluun sisälle.
Käyttötiheys Päivittäin, viikoittain, kuukausittain.

1. Käyttäjällä on pääsy internettiin
2. Käyttäjä avaa applikaation selaimessa
3. Käyttäjä syöttää omat tunnuksensa ”käyttäjänimi” ja ”salasana” kohtiin
4. Käyttäjä pääsee sisään järjestelmään

Virhetilanteet: ”Väärä käyttäjänimi tai salasana” - Käyttäjä syöttää väärät tunnustiedot ”Käyttäjänimellä ei löytynyt yhtään käyttäjää” - Käyttäjä ei ole rekisteröitynyt järjestelmään.

Harjoitusten tarkastelu
Toimija(t) Urheilija, Valmentaja, Admin
Esiehto Kirjautunut palveluun
Lopputulos Käyttäjä voi tarkastella urheilijan tekemiä harjoituksia.
Käyttötiheys Päivittäin, viikoittain, kuukausittain

1.Käyttäjällä on pääsy internettiin
2.Käyttäjä avaa applikaation selaimessa ja klikkaa kirjaudu sisään
3.Käyttäjä klikkaa kohdasta ”harjoitukset” ja siirtyy tarkastelemaan tekemiään harjoituksiaan, jossa voi sekä lisätä, muokata ja poistaa niitä

Käyttäjätietojen Hallinta

Toimija(t)	Urheilija, Valmentaja, Admin
Esiehto	Käyttäjän tulee olla kirjautunut järjestelmään.
Lopputulos	Käyttäjä voi muuttaa omia tietojaan.
Käyttötiheys	Päivittäin, viikoittain, kuukausittain.

1. Käyttäjä on kirjautuneena järjestelmään
2. Käyttäjä painaa ”oma sivu” nappia ja siirtyy sivulle
3. Käyttäjällä on mahdollisuus muokata omia tietojaan tai poistaa oma tunnus. Admin pystyy muokkaamaan kaikkien käyttäjien tietoja

Harjoituksen Lisäys

Toimija(t)	Urheilija, Valmentaja, Admin
Esiehto	Käyttäjän tulee olla rekisteröitynyt järjestelmään.
Lopputulos	Harjoitus saadaan lisättyä palveluun
Käyttötiheys	Päivittäin, viikoittain, kuukausittain

1. Käyttäjä on kirjautuneena järjestelmään
2. Käyttäjä painaa ”lisää harjoitus” kohdasta ja siirtyy kyseiselle sivulle
3. Käyttäjä lisää tarvittavat tiedot tehdystä harjoituksesta
4. Käyttäjä painaa nappia ”tallenna harjoitus”

Virhetilanteet: ”Puuttuvia tietoja” – Käyttäjä ei lisännyt tarvittavia tietoja. ”Kyseinen harjoitus on jo tallennettu” - Käyttäjä on jo tallentanut kyseisen harjoituksen tiedot järjestelmään.

Harjoitusten hallinta

Toimija(t)	Urheilija, Valmentaja, Admin
Esiehto	Käyttäjän tulee olla rekisteröitynä järjestelmään.
Lopputulos	Käyttäjä voi poistaa tai muokata harjoituksiaan.
Käyttötiheys	Päivittäin, viikoittain, kuukausittain.

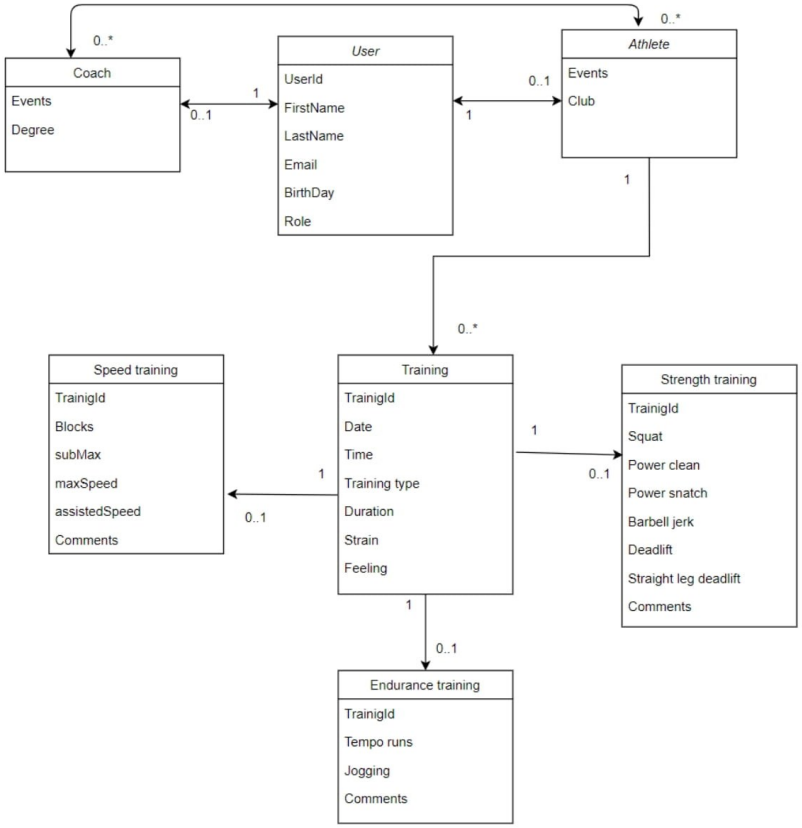
1. Käyttäjä on kirjautuneena järjestelmään
2. Käyttäjä painaa ”viimeisimmät harjoitukset”
3. Käyttäjä valitsee harjoituksen, jota haluaa muokata tai poistaa
4. Käyttäjä painaa nappia ”tallenna harjoitus”

Virhetilanteet: ”Puuttuvia tietoja” – Käyttäjä ei lisännyt tarvittavia tietoja. ”Kyseinen harjoitus on jo tallennettu”

4 Säilytettävät tiedot

4.1 Järjestelmän luokkamalli

Kuva 3. Luokkakaavio



4.1.1 **User**

Luokan määritelmä:

Palvelun käyttäjä, jolla on joko urheilijan tai valmentajan rooli.

Attribuutit

nimi	tietotyyppi	pituus	kuvaus	arvojoukko	oletusarvo
Id	Long	1000000	Käyttäjän henkilökohtainen tunniste	1–999	null
First name	Varchar	50	Käyttäjän etunimi	A-Ö	null
Last name	Varchar	50	Käyttäjän sukunimi	A-Ö	null
Email	Varchar	100	Käyttäjän sähköpostiosoite	A-9	Null
Birthday	LocalDate	100	Käyttäjän syntymäpäivä	1–9	null
Role	Varhcar	10	Käyttäjän rooli, joka määrittää oikeudet	Coach, Athlete	Athlete

Suhteet

Käyttäjällä täytyy olla joko yksi valmentaja tai yksi urheilija

Vastuut

Ylläpitää tietonsa

Operaatiot

get_user() hakee kaikki käyttäjän tiedot

Määrätiedot
Käyttäjiä on tuhansia – kymmeniä tuhansia

4.1.2 **Coach**

Luokan määritelmä:

Yhden käyttäjän määrittämä status omalle profiililleen

Yliluokka -

Attribuutit

nimi	tietotyyppi	pituus	kuvaus	arvojoukko	oletusarvo
Id	Long	1000000	Valmentajan henkilökohtainen tunniste	1–9	null
Events	Varchar	100	Valmentajan pääalueet valmennuksessa	A-Ö	null
Degree	Varchar	100	Valmentajan käymä valmentajan tutkinto	A-Ö	null

Suhteet

Valmentajalla on aina yksi käyttäjä

Valmentajalla voi olla useita urheilijoita

Vastuut

Ylläpitää tietonsa

Tietää asunnon numeron, josta varaus on tehty

Operaatiot -

Määrätiedot

Valmentajia voi olla yhtä paljon kuin käyttäjiä. Todellinen suhdeluku tulee olemaan noin 1/5 verrattuna käyttäjiin.

4.1.3 **Athlete**

Luokan määritelmä:

Yhden käyttäjän määrittämä status omalle profiililleen

Yliluokka -

Attribuutit

nimi	tietotyyppi	pituus	kuvaus	arvojoukko	oletusarvo
Id	Long	1000000	Urheilijan henkilökohtainen tunniste	0–9	null
Events	Varchar	200	Urheilijan lajit	A-Ö	null
Club	Varchar	100	Urheilijan urheiluseura	A-Ö	null

Suhteet

Urheilijalla on aina yksi käyttäjä

Urheilijalla voi olla useita valmentajia

Vastuut

Ylläpitää tietonsa

Operaatiot -

Määrätiedot

Urheilijoita voi olla yhtä paljon kuin käyttäjiä. Todellinen suhdeluku tulee olemaan noin 4/5 verrattuna käyttäjiin.

4.1.4 **Training**

Luokan määritelmä:

Urheilijan lisäämä harjoitus

Yliluokka -

Attribuutit

nimi	tietotyyppi	pituus	kuvaus	arvojoukko	oletusarvo
Id	Long	1000000	Harjoituksen yksilöivä tunnus	0–9	null
Date	LocalDate	8	Päivä, jolloin harjoitus on tehty	yyyy-mm-dd	null

Time	LocalTime	8	Aika, jolloin harjoitus on aloitettu	HH-mm-ss-ns	null
Training type	Varchar	100	Harjoituksen tyyppi	Speed training, Strength training, Endurance training	null
Duration	Long	4	Harjoituksen kesto minuutteina	0-9	null
Strain	Long	2	Harjoituksen kuormitus	1,2,3,4,5,6,7,8,9,10	null
Feeling	Long	2	Harjoituksesta jäänyt fiilis	1,2,3,4,5,6,7,8,9,10	null

Suhteet

Harjoituksella on yksi urheilija
Harjoituksella voi olla yksi kolmesta harjoitustyypistä

Vastuut

Ylläpitää tietonsa
Tietää milloin harjoitus on tehty

Operaatiot

getTraining() hakee harjoituksen tiedot
getTrainingByDate() hakee tietyn päivän harjoitukset

getTrainingsByType()
hakee tietyn harjoitustyypin harjoitukset

Määrätiedot

Yhdellä käyttäjällä on satoja, tuhansia tai kymmeniä tuhansia harjoituksia

4.1.5 **Speed training**

Luokan määritelmä:

Harjoituksen tyyppi

Yliluokka -

Attribuutit

nimi	tietotyyppi	pituus	kuvaus	arvojoukko	oletusarvo
Id	Long	1000000	Harjoituksen	0–9	null

			yksilöivä tunnus		
Blocks	Varchar	1 000	Telineharjoittelun tiedot	A-Ö	null
Submax speed	Varchar	200	Submax harjoittelun tiedot	0–9, m, x, +, (,)	null
Max speed	Varchar	200	Maksiminopeuden harjoittelun tiedot	0–9, m, x, +, (,)	null
Assisted speed	Varchar	200	Ylinopeusharjoittelun tiedot	0–9, m, x, +, (,)	null
Comments	Varchar	1 000	Harjoituksen kommentit	A-Ö	null

Suhteet

Nopeusharjoittelulla on yksi harjoitus

Vastuut

Ylläpitää tietonsa

Operaatiot -

Määrätiedot

Nopeusharjoitteluiden määrä on kymmeniä, satoja tai tuhansia

4.1.6 **Strength training**

Luokan määritelmä:

Harjoituksen tyyppi

Yliluokka -

Attribuutit

nimi	tietotyyppi	pituus	kuvaus	arvojoukko	oletusarvo
Id	Long	1000000	Harjoituksen yksilöivä tunnus	0–9	null
Squat	Varchar	200	Kyykkyharjoitteen tiedot	0–9, kg, (,)	null
Power clean	Varchar	200	Rinnallevetoharjoitteen tiedot	0–9, kg, (,)	null
Power snatch	Varchar	200	Tempausharjoitteen tiedot	0–9, kg, (,)	null
Barbell jerk	Varchar	200	Ylöstyöntöharjoitteen tiedot	0–9, kg, (,)	null
Deadlift	Varchar	200	Maastavetoharjoitteen tiedot	0–9, kg, (,)	null

Straight leg deadlift	Varchar	200	Suorin jaloin maastavedot	0–9, kg, (,)	null
Comments	Varchar	1 000	Harjoituksen kommentit	A-Ö	null

Suhteet

Voimaharjoittelulla on yksi harjoitus

Vastuut

Ylläpitää tietonsa

Operaatiot -

Määrätiedot

Voimaharjoitteluiden määrä on kymmeniä, satoja tai tuhansia

4.1.7 **Endurance training**

Luokan määritelmä:

Harjoituksen tyyppi

Yliluokka -

Attribuutit

nimi	tietotyyppi	pituus	kuvaus	arvojoukko	oletusarvo
Id	Long	1000000	Harjoituksen yksilöivä tunnus	0–9	null
Tempo runs	Varchar	200	Määräintervallien tiedot	0–9, m, x, +, (,)	null
Jogging	Varchar	200	Hölkkäjuoksun tiedot	0–9, m, x, +, (,)	null
Comments	Varchar	1 000	Harjoituksen kommentit	A-Ö	null

Suhteet

Kestävyysharjoittelulla on yksi harjoitus

Vastuut

Ylläpitää tietonsa

Operaatiot -

Määrätiedot

Kestävyysharjoituksien määrä on kymmeniä, satoja tai tuhansia

5 Tietojen käyttöyhteenvedot (CRUD)

Taulukko 2. Käyttöyhteenveto

	Create	Read	Update	Delete
User	x	x	x	x
Athlete		x		
Coach		x		
Training	x	x	x	x
Strength training		x		
Speed training		x		
Endurance training		x		

Liite 2 Käyttäjä

4.1.1 User

Luokan määritelmä:

Palvelun käyttäjä, jolla on joko urheilijan tai valmentajan rooli.

Attribuutit

nimi	tietotyyppi	pituus	kuvaus	arvojoukko	oletusarvo
Id	Long	1000000	Käyttäjän henkilökohtainen tunniste	1–999	null
First name	Varchar	50	Käyttäjän etunimi	A-Ö	null
Last name	Varchar	50	Käyttäjän sukunimi	A-Ö	null
Email	Varchar	100	Käyttäjän sähköpostiosoite	A-9	Null
Birthday	LocalDate	100	Käyttäjän syntymäpäivä	1–9	null
Role	Varhcar	10	Käyttäjän rooli, joka määrittää oikeudet	Coach, Athlete	Athlete

Suhteet

Käyttäjällä täytyy olla joko yksi valmentaja tai yksi urheilija

Vastuut

Ylläpitää tietonsa

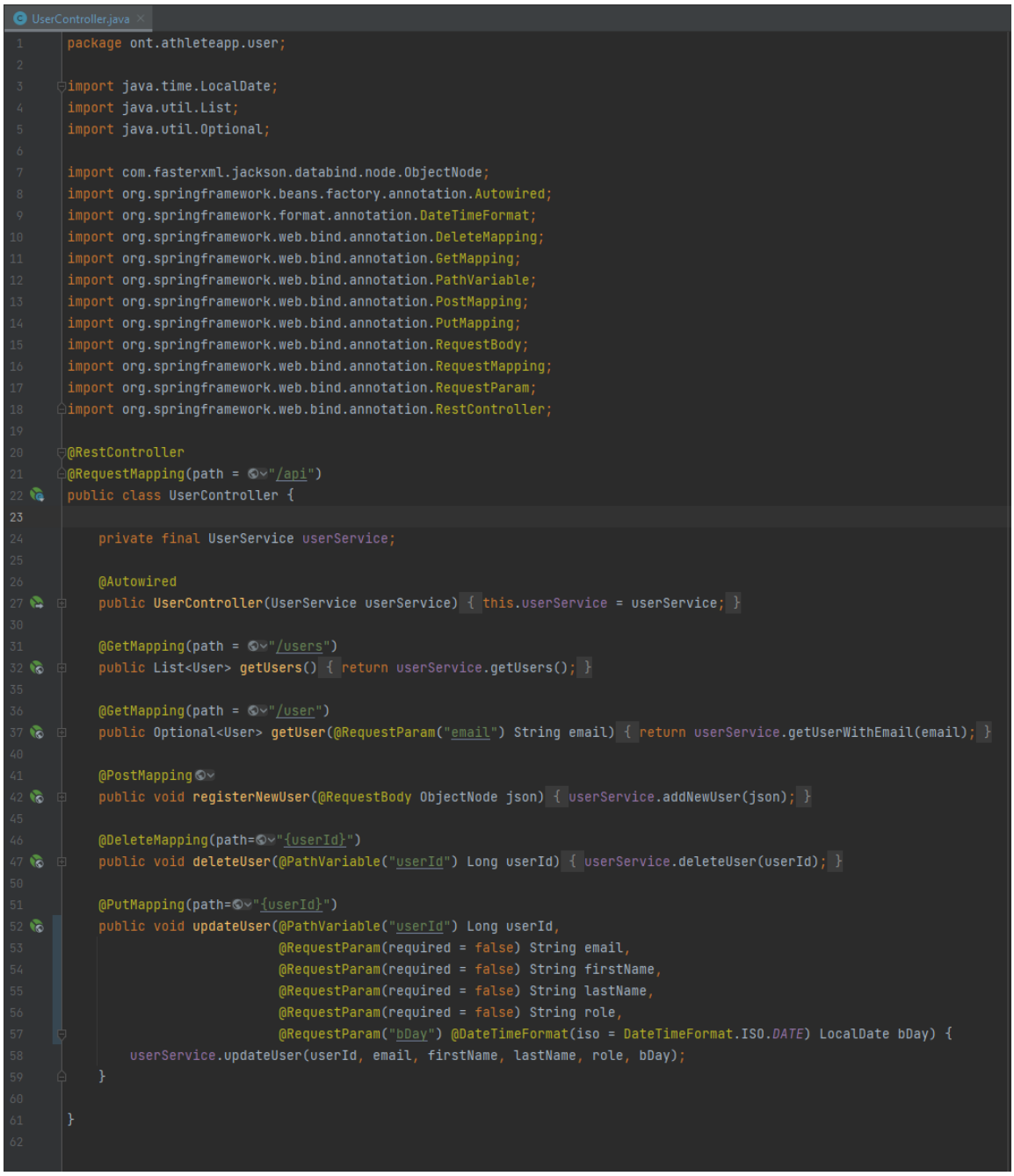
Operaatiot

get_user() hakee kaikki käyttäjän tiedot

Määrätiedot

Käyttäjiä on tuhansia – kymmeniä tuhansia

Liite 3 Controller-luokka



UserController.java

```
package ont.athleteapp.user;

import java.time.LocalDate;
import java.util.List;
import java.util.Optional;

import com.fasterxml.jackson.databind.node.ObjectNode;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.format.annotation.DateTimeFormat;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping(path = "/api")
public class UserController {

    private final UserService userService;

    @Autowired
    public UserController(UserService userService) { this.userService = userService; }

    @GetMapping(path = "/users")
    public List<User> getUsers() { return userService.getUsers(); }

    @GetMapping(path = "/user")
    public Optional<User> getUser(@RequestParam("email") String email) { return userService.getUserWithEmail(email); }

    @PostMapping
    public void registerNewUser(@RequestBody ObjectNode json) { userService.addNewUser(json); }

    @DeleteMapping(path="{userId}")
    public void deleteUser(@PathVariable("userId") Long userId) { userService.deleteUser(userId); }

    @PutMapping(path="{userId}")
    public void updateUser(@PathVariable("userId") Long userId,
                           @RequestParam(required = false) String email,
                           @RequestParam(required = false) String firstName,
                           @RequestParam(required = false) String lastName,
                           @RequestParam(required = false) String role,
                           @RequestParam("bDay") @DateTimeFormat(iso = DateTimeFormat.ISO.DATE) LocalDate bDay) {
        userService.updateUser(userId, email, firstName, lastName, role, bDay);
    }

}
```

Liite 4 Console

```
Hibernate:
    select
        user0_.id as id1_6_,
        user0_.b_day as b_day2_6_,
        user0_.email as email3_6_,
        user0_.first_name as first_na4_6_,
        user0_.last_name as last_nam5_6_,
        user0_.role as role6_6_
    from
        users user0_
    where
        user0_.email=?
Hibernate:
    select
        nextval ('user_sequence')
Hibernate:
    insert
    into
        users
        (b_day, email, first_name, last_name, role, id)
    values
        (?, ?, ?, ?, ?, ?)
Hibernate:
    insert
    into
        athlete
        (club, events, user_id)
    values
        (?, ?, ?)
```

Liite 5 Kansiorakenne