Centria
AMMATTIKORKEAKOULU

Johanna Lyytinen

ENDOKARDIITTIKORTTI

Endokardiittilaajennos Tampereen Sydänsairaalan sydäntietojärjestelmään

Opinnäytetyö
CENTRIA-AMMATTIKORKEAKOULU
Tieto- ja viestintätekniikan koulutusohjelma
Helmikuu 2022

TIIVISTELMÄ OPINNÄYTETYÖSTÄ

Centria AMMATTIKORKEAKOULU

Centria-ammattikorkeakoulu	**Aika** Helmikuu 2022	**Tekijä/tekijät** Johanna Lyytinen
Koulutus Tieto- ja viestintätekniikan koulutusohjelma		☒ AMK ☐ YAMK
Työn nimi ENDOKARDIITTIKORTTI. Endokardiittilaajennos Tampereen Sydänsairaalan sydäntietojärjestelmään		
Työn ohjaaja Sakari Männistö		**Sivumäärä** 37
Työelämäohjaaja Jani Mustonen, Tiina Rutanen		

Opinnäytetyön aiheena oli kehittää Tampereen Sydänsairaalan sydäntietojärjestelmään Kardioon endokardiittipotilaiden hoitoa ja seurantaa varten tarkoitettu endokardiittikortti. Toimeksiantaja oli Tampereen Sydänsairaala, joka on sydänsairauksien hoitoon keskittynyt osakeyhtiö.

Tietyn potilasryhmän hoitoon ja seurantaan liittyvää kokonaisuutta kutsutaan Tampereen Sydänsairaalan sydäntietojärjestelmässä Kardiossa kortiksi, joten opinnäytetyössäkin käytettiin samaa termiä.

Endokardiittikortti toteutettiin osana normaalia Kardion kehitystyötä. Kortin käyttöliittymäkerros toteutettiin Apache Wicketillä, joka on web-käyttöliittymien toteuttamiseen tarkoitettu Java-pohjainen sovelluskehys. Liiketoimintalogiikkakerros toteutettiin Javalla ja tietokantakerroksessa käytettiin PostgreSQL:ää. Tietokannan ja liiketoimintalogiikkakerroksen yhdistämisessä hyödynnettiin Java Persistence API:a (JPA).

Tampereen Sydänsairaala otti kehitetyn endokardiittilaajennoksen käyttöön syksyllä 2021.

Asiasanat

Apache Wicket, Endokardiitti, Java, JPA, PostgreSQL, Sydänsairaala

ABSTRACT

Centria UNIVERSITY OF APPLIED SCIENCES

<table>
<tr><td>Centria University
of Applied Sciences</td><td>Date
February 2022</td><td>Author
Johanna Lyytinen</td></tr>
<tr><td colspan="3">Degree programme
Information Technology</td></tr>
<tr><td colspan="3">Name of thesis
ENDOCARDITIS CARD. Endocarditis extension for Tampere Heart Hospital’s cardiology information system</td></tr>
<tr><td colspan="2">Centria supervisor
Sakari Männistö</td><td>Pages
37</td></tr>
<tr><td colspan="3">Instructor representing commissioning institution or company
Jani Mustonen, Tiina Rutanen</td></tr>
</table>

The subject of this thesis was to develop a patient card to register and follow the treatment of the patients with endocarditis. This card will become a part of the cardiology information system Kardio used in Tampere Heart Hospital. The commissioning company was Tampere Heart Hospital which is a hospital specialized in the treatment of heart problems.

In Kardio a collection of treatment and follow up information of a single patient group is called a card. This term is also used in the thesis.

Endocarditis card was developed as a part of the normal development work for Kardio. The user interface of the card was created with Apache Wicket, Java-based framework for developing websites. The Business logic layer was developed with Java and the database layer was created with PostgreSQL. Database layer and business logic layer was connected by using Java Persistence API (JPA).

Tampere Heart Hospital took the endocarditis card into use in fall 2021.

Key words

Apache Wicket, Endocarditis, Heart Hospital, Java, JPA, PostgreSQL

KÄSITTEIDEN MÄÄRITTELY

ANNOTAATIO

Merkintä. Java-ohjelmointikielen rakenne, joka mahdollistaa metadatan lisäämisen kielen perusrakenteisiin, kuten luokkaan tai metodiin. Annotaatiot merkitään @-notaatiolla

APACHE WICKET

Web-käyttöliittymien toteuttamiseen tarkoitettu Java-pohjainen sovelluskehys

ENTITEETTI

Tietokohde, olio, itsenäinen kokonaisuus

ENUMERAATIO

Lueteltu tyyppi

GITLAB

Pilvipalvelu, joka tukee Git-versionhallintaa hyödyntäviä ohjelmistokehitysprojekteja

JAVA

Olioperustainen ohjelmointikieli

JPA

Java Persistence Api

KARDIO

Sydänsairaalan sydäntietojärjestelmä

KORTTI

Sydäntietojärjestelmässä oleva yhden potilasryhmän hoitoon ja rekisteröintiin tarkoitettu kokonaisuus

POSTGRESQL

Avoimeen lähdekoodiin perustuva tietokanta

TIIVISTELMÄ
ABSTRACT
KÄSITTEIDEN MÄÄRITTELY
SISÄLLYS

1 JOHDANTO 1

2 SYDÄNSAIRAALA, KARDIO JA ENDOKARDIITTI 2
2.1 Tampereen Sydänsairaala 2
2.2 Sydäntietojärjestelmä Kardio 2
2.3 Endokardiitti 3

3 LÄHTÖKOHTA JA SOVELLUKSEN VAATIMUKSET 5

4 TOTEUTUKSESSA KÄYTETYT TEKNOLOGIAT 7
4.1 Tietokanta 7
4.1.1 Relaatiotietokanta 7
4.1.2 Tiedon normalisointi 8
4.1.3 PostgreSQL 9
4.2 Java 10
4.3 Spring-sovelluskehys 11
4.4 JPA 14
4.4.1 Entiteetti 15
4.4.2 Entiteettien väliset suhteet 16
4.4.3 Kokoelma 18
4.4.4 Enumeraatio 18
4.5 Apache Wicket 19
4.6 GitLab 19

5 SOVELLUKSEN TOTEUTUS 21
5.1 Kardion arkkitehtuurikerrokset 21
5.2 Tietokantakerros 22
5.3 Sovelluslogiikkakerros 24
5.4 Käyttöliittymäkerros 26

6 YHTEENVETO JA POHDINTA 33

LÄHTEET 35

KUVAT

KUVA 1. Javan tietokantarajapinnan transaktioiden hallinta 13
KUVA 2. Koodiesimerkki mappedBy-määrityksen käytöstä 16
KUVA 3. Koodiesimerkki @ManyToOne-suhteesta 17
KUVA 4. Koodiesimerkki @OneToMany-suhteesta 17
KUVA 5. Koodiesimerkki @ManyToMany-suhteesta 17
KUVA 6. Koodiesimerkki kahdensuuntaisesta @ManyToMany-suhteesta 18
KUVA 7. Kerroksittainen arkkitehtuuri 21
KUVA 8. Endokardiittikortin ER-kaavio 23
KUVA 9. Koodiesimerkki kokoelmasta 24
KUVA 10. Koodiesimerkki type-annotaation käytöstä 24

KUVA 11. Koodiesimerkki service-kerroksesta 25
KUVA 12. Koodiesimerkki enumeraation luokasta ja vakioarvosta 25
KUVA 13. Koodiesimerkki hakusivun lisäehdoista 26
KUVA 14. Käyttöliittymäesimerkki uuden tapahtuman lisäyksestä 26
KUVA 15. Käyttöliittymäesimerkki endokardiittikortin alkuosasta 27
KUVA 16. Endokardiittikortin Wicket-paneelit 27
KUVA 17. Koodiesimerkki paneeleiden kutsumisesta 28
KUVA 18. Paneelin funktiot 28
KUVA 19. Koodiesimerkki container-olion lisäämisestä 28
KUVA 20. Koodiesimerkki container-olion hyödyntämisestä 29
KUVA 21. Käyttöliittymäesimerkki pop up-ohjeen tarjoamisesta 30
KUVA 22. Koodiesimerkki aputekstin lisäämisestä 30
KUVA 23. Käyttöliittymäesimerkki automaattisesta täydennyksestä 31
KUVA 24. Käyttöliittymäesimerkki hakusivusta 31
KUVA 25. Koodiesimerkki hakupaneelista 32

1 JOHDANTO

Opinnäytetyön aiheena on kehittää Sydänsairaalan sydäntietojärjestelmään endokardiittipotilaiden hoitoon ja seurantaan suunniteltu rekisteröintiväline. Sydänsairaalan sydäntietojärjestelmässä tällaista tietyn potilasryhmän hoitoon ja seurantaan suunnattua kokonaisuutta kutsutaan kortiksi, joten kutsutaan sitä tässä opinnäytetyössäkin samalla nimellä.

Endokardiitti on mikrobien aiheuttama sydämen sisärakenteiden tulehdus, joka sijaitsee yleensä sydämen läppärakenteissa. Hoitamattomana endokardiitti on hengenvaarallinen. Hoitona käytetään suonensisäistä antibioottia. Endokardiitti voi aiheuttaa leikkaushoitoa vaativia vaurioita läppiin ja niitä ympäröiviin rakenteisiin. (Kettunen 2020.) Endokardiittin vuosittainen ilmaantuvuus on luokkaa 3–10 tapausta/100 000 asukasta (Saraste, Turpeinen & Hohenthal 2016, 895). Endokardiitin aiheuttama tautitaakka ei näytä vähenevän Suomessa eikä maailmalla (Suhonen, Halavaara, Lönnqvist, Teittinen, Rajala 2021 563–574). Voidaan siis todeta, että tämän potilasryhmän hoitoon ja seurantaan käytettävä kortti on hyvinkin tarpeellinen kliinisessä työssä.

Tähän mennessä Sydänsairaalalla ei ole ollut yhteneväistä rekisteröintityökalua näille potilaille, joten opinnäytetyön aihe on asiakaslähtöinen ja on tehty palvelemaan heidän tarpeitaan. Opinnäytetyön tärkein tavoite oli luoda käyttöön kortti, joka vastaa asiakkaan tarpeita mahdollisimman kattavasti. Lopputuloksena syntyvä kortti otetaan kliiniseen käyttöön Sydänsairaalan sydäntietojärjestelmään osana normaalia järjestelmän kehitystyötä. Toimeksiantajana on Tampereen Sydänsairaala, joka on sydänsairauksien hoitoon keskittynyt osakeyhtiö.

Lisäksi opinnäytetyön tavoitteena on oppia tuntemaan Sydänsairaalan sydäntietojärjestelmää paremmin ja oppia kehittämään isohko kokonaisuus tietyn potilasryhmän hoitoon jo olemassa olevien kokonaisuuksien rinnalle. Opinnäytetyön tavoitteena on myös tutustua itse endokardiittiin ja saada sitä kautta laajempaa ymmärrystä endokardiittikortin toteutukseen. Toteutusvaiheessa sain apua sekä kehitykseen että testaukseen työnantajaltani Cinia Oy:ltä. Opinnäytetyön jälkeen on tarkoitus kehittää endokardiittikortin toimintaa ja ominaisuuksia asiakkaan toivomalla tavalla ja lisäksi luoda erillinen raportointiosio, jota pystytään hyödyntämään niin kliinisen työn seurantaan ja analysointiin kuin myös mahdollisiin jatkotutkimustarpeisiin.

2 SYDÄNSAIRAALA, KARDIO JA ENDOKARDIITTI

2.1 Tampereen Sydänsairaala

Tampereen Sydänsairaala on sydänsairauksien hoitoon keskittynyt osakeyhtiö, ja se on perustettu vuonna 2004. Alun perin sydämen hoidon eri ammattiryhmät toimivat hajallaan Tampereen yliopistollisessa sairaalassa ja saattoi olla, että potilas joutui vaihtamaan hoitojaksonsa aikana useaan kertaan hoitavaa osastoa. Tämä aiheutti viiveitä hoidossa. Esimerkiksi potilas saattoi joutua odottamaan varjoainekuvaukseen pääsemistä viikonkin ja nykyään se tehdään noin vuorokauden sisällä. Vuonna 2004 Sydänsairaalasta tuli itsenäinen prosessiorganisaatio, joka tarkoittaa sitä, että kaikki sairaalan toiminnot sijoitetaan potilaan hoitopolun ympärille. Tällöin potilas pysyy koko hoitojaksonsa ajan samassa yksikössä ja tämä nopeuttaa hoidon saamista ja hoitoon pääsyä. Vuonna 2007 Sydänsairaalasta tuli liikelaitos, ja edelleen vuonna 2010 Sydänsairaala muuttui osakeyhtiöksi. Osakeyhtiömuotoinen hallintomalli mahdollistaa Sydänsairaalalle muun muassa joustavamman ja nopeamman toiminnan kehittämisen, mikä puolestaan lisää potilaan saaman hoidon laatua, kun voidaan keskittyä oikeisiin asioihin. (Sydänsairaalan tarina 2017.)

Sydänsairaalan yhtiöjärjestyksessä on kirjattuna, että tavoitteena on kustannustehokkaan ja vaikuttavan potilashoidon turvaaminen, ei niinkään voiton tekeminen. Luonnollisesti toiminnan tulee olla myös taloudellisesti kannattavaa, jotta hoitoa voidaan entisestään kehittää ja toimintaa parantaa. Sydänsairaalan omistavat Pirkanmaan sairaanhoitopiiri, Kanta-Hämeen sairaanhoitopiiri ja Keski-Suomen sairaanhoitopiirin kuntayhtymä. Tällainen julkisen ja yksityisen toimintamallin yhdistelmä on maailmanlaajuisestikin ainulaatuinen. (Sydänsairaalan tarina 2017.)

2.2 Sydäntietojärjestelmä Kardio

Sydänsairaalan sydäntietojärjestelmä on nimeltään Kardio, ja se on alun perin tehty kardiologian ja sydän- ja rintaelinkirurgian laadun seuraamista ja raportointia varten. Kardio on otettu käyttöön Tampereen Sydänsairaalassa vuonna 2012. Aluksi Kardion avulla oli tarkoitus kerätä muuta potilastietoa täydentävää tietoa, jota voidaan käyttää hoitopäätösten tueksi ja myös tieteellisen tutkimuksen tarpeisiin.

Lisäksi tavoitteena oli selkeyttää ja tehostaa lääkäreiden ja hoitohenkilökunnan työskentelyä. Sen jälkeen Kardio on kasvanut ja kehittynyt laaja-alaisemmaksi järjestelmäksi. (Toiminnanohjausjärjestelmä laajenee Sydänsairaalan tarpeiden tahtiin, 2021.)

Aikaisemmin kun käytössä olivat vielä fyysiset potilastietokansiot, oli tietyn potilasryhmän hoitoon ja seurantaan käytettävä kokonaisuus fyysisestikin kortti. Siksi nimi on jäänyt elämään myös Kardioon ja siellä olevia kokonaisuuksia kutsutaan edelleen korteiksi.

2.3 Endokardiitti

Endokardiitti on sydämen sisärakenteiden eli sydänläppien ja sydämen sisäkalvon tulehdus. Tulehdus aiheutuu, kun bakteerit tai sienieliöt kulkeutuvat verenkierron mukana sydämeen ja tarttuvat sydänläppien sisäpinnoille. Bakteerit voivat päästä verenkiertoon monista eri infektioporteista, esimerkiksi huonosti hoidetuista hampaista, suoliston, virtsateiden, hampaiden tai nielun alueen kirurgisesta tai kajoavasta toimenpiteestä. Suonensisäisten huumeiden käyttö lisää selkeästi endokardiitin mahdollisuutta. Myös tatuointi ja lävistykset voivat lisätä riskiä sellaisilla henkilöillä, joilla on synnynnäinen sydän- tai läppävika. (Kettunen 2020.) Muuhun hoitoon liittyvien endokardiittien osuus on lisääntynyt, koska yhä sairaammille ja vanhemmille potilaille tehdään leikkauksia tai kajoavia toimenpiteitä. Tällaisia hoitoon liittyviä endokardiitteja arvioidaan olevan 10–30 % kaikista endokardiiteista. (Saraste ym. 2016, 895.) Monilla endokardiittiin sairastuneilla ei kuitenkaan välttämättä ole tiedossa mitään altistavaa vammaa tai toimenpidettä tai läppä- tai muuta sydänvikaa (Kettunen 2020).

Endokardiitin oireet ovat vaihtelevat, ja niihin vaikuttaa se, mikä bakteeri tulehduksen aiheuttaa. Tyypillisesti endokardiitin aiheuttajabakteeri on stafylokokkibakteeri ja tällöin tulehdus on nopeasti etenevä kuumesairaus. Endokardiitti on toisinaan hitaasti etenevä, ja sen oireina ovat pitkään kestävä kuumeilu, yleinen sairauden tunne, laihtuminen ja yöhikoilu. Tällöin taudinaiheuttajabakteeri on tyypillisesti streptokokkibakteeri. Endokardiitti voi aiheuttaa sydämen läppään vuotovian, joka voi johtaa nopeasti sydämen vajaatoimintaan. Tällöin oireina kuumeilun ohella on myös suorituskyvyn lasku ja hengästyminen vähäisessäkin rasituksessa. Jos tulehdusta ei hoideta, sydämestä verenkiertoon pääsevät tulehdusjätteet voivat aiheuttaa muun muassa keskushermosto-oireita, ihomuutoksia tai nivelkipuja. Hoitamattomana endokardiitti on hengenvaarallinen sairaus, jonka tutkiminen ja hoito keskittyy aina sairaalaolosuhteisiin. Endokardiitti hoidetaan usein pitkäkestoisella suonensisäisellä antibiootilla, ja osalle potilaista joudutaan tekemään läppäleikkaus. (Kettunen 2020.)

Endokardiitti nostaa veren tulehdusarvoja, ja veren bakteeriviljelyssä löytyy aiheuttajabakteeri. Läppävika aiheuttaa stetoskoopilla kuultavissa olevan sivuäänen, jonka toteaminen on olennaista tutkimusten ja hoidontarpeen arvioinnissa etenkin kuumeisilla potilailla. Läppätulehdus voidaan todeta sydämen ultraäänitutkimuksella. (Kettunen 2020.) Endokardiitin diagnosointi perustuu huolelliseen kliiniseen arvioon. Kansainvälisessä diagnoosikriteeristössä, niin sanotuissa Duken kriteereissä, pääkriteereinä ovat muun muassa läppävegetaatiolöydös sydämen ultraäänitutkimuksessa ja veriviljelypositiivisuus. Sivukriteereihin luetaan kuume ja erilaiset endokardiitille altistavat tekijät. Yhdistelemällä näitä kriteereitä voidaan asettaa varma tai mahdollinen endokardiittidiagnoosi tai hylätä se. (Suhonen ym. 2021, 563–574.)

Sydämen ultraäänitutkimus tulee tehdä viipymättä epäiltäessä endokardiittia. Se voidaan tehdä transtorakaalisesti eli rintakehän päältä tai transesofageaalisesti eli ruokatorven kautta. Tutkimuksen löydöksiä voivat olla vegetaatiot, tekoläppien osittainen irtoaminen, valeaneurysmat ja paravalvulaaritilan paiseet. Jos endokardiittiepäily on vahva ja transtorakaalinen löydös ei ole yksiselitteisesti poissulkeva, tulee tehdä transesofageaalinen ultraääni. Jos molemmat löydökset ovat negatiivisia, mutta epäily on vahva, tulisi ultraäänitutkimus toistaa 5–7 vuorokauden kuluttua. (Suhonen ym. 2021, 563–574.) Kuvantamislöydösten lisäksi 80–90 % endokardiiteista on veriviljelypositiivisia (Saraste ym. 2016, 899).

Endokardiitin vuosittaisen esiintyvyyden Suomessa on arvioitu olevan 6,3/100 000 henkilövuotta. Endokardiitin aiheuttama tautitaakka ei näytä vähenevän Suomessa eikä maailmalla. Akuutin vaiheen kuolleisuus on noin 10–25 %, ja vuoden kuluttua endokardiitin toteamisesta kuolleisuus on noin 30 %. (Suhonen ym. 2021, 563–574.)

3 LÄHTÖKOHTA JA SOVELLUKSEN VAATIMUKSET

Alun perin endokardiittikortti oli tarkoitus toteuttaa osana Kardion normaalia kehitystyötä. Hyvin pian huomattiin endokardiittikortin toteuttamisessa olevan mahdollisuuksia myös opinnäytetyön aiheeksi. Endokardiittikortti oli sopivan laaja ja yhtenäinen kokonaisuus ja sen lisäksi täysin uusi kortti Kardioon. Asiakkaalta saatiin lupa hyödyntää toteutusta myös opinnäytetyön aiheeksi.

Endokardiittikortin sisältö määriteltiin yhdessä asiakkaan kanssa tarpeita vastaavaksi. Liikkeelle lähdettiin asiakkaan tekemästä pohjasta, johon vielä tarkennettiin määrittelyjä yhteisessä palaverissa. Tarkoitus oli tehdä kortti, joka toimii hyvin kliinisessä työssä endokardiittipotilaita tutkittaessa ja hoidettaessa. Tavoitteena oli myös muodostaa kortin sisältö loogisesti eteneväksi kokonaisuudeksi. Jokainen määritelty valikko ja kohta käytiin yhdessä läpi ennen toteutuksen aloitusta. Yhdessä pohdittiin myös muun muassa kortin valintojen vaikutusta myöhemmin toteutettavaan raporttiosuuteen. Korttiin tuli asiakkaan toiveesta paljon tekstikenttiä, joihin saa vapaasti täydentää haluamansa asian. Näiden näyttäminen raportoinnissa on erittäin haastavaa, mutta tämä hyväksyttiin yhdessä, koska näille vapaakentille oli kuitenkin selkeä tarve. Osa näistä vapaatekstikentistä sekä muista valinnoista haluttiin tehdä sellaisiksi, että käyttäjän valitessa tietyn valinnan aukeaa lisää täydennettävää tai vapaatekstikenttä. Näin vältytään siltä, että kaikki mahdolliset täydennyskohdat olisivat jo valmiiksi näkyvissä, vaikka niitä ei tarvitsisi täydentää kuin tiettyjen valintojen jälkeen. Tämä helpottaa huomattavasti kortin käyttämistä, koska käyttöliittymä itse opastaa käyttäjää täyttämään oikeat kohdat.

Korttiin olisi ollut mahdollista asettaa myös niin sanottuja pakollisia kenttiä, joiden täydentäminen olisi ollut edellytys kortin tallentamiselle ja käyttöliittymä olisi tuottanut käyttäjälle virheilmoituksen, jos korttia yritetään tallentaa ilman näitä vaadittuja tietoja. Tässä vaiheessa endokardiittikortin toteutusta mitään valinnoista ei kuitenkaan määritelty pakollisiksi. Nämä ovat lisättävissä myöhemmin kortille asiakkaan niin halutessa.

Endokardiittikorttia käytettäessä siihen täydennetään potilaan hoitoon ja seurantaan liittyviä asioita. Esimerkiksi jokainen tehty sydämen ultraäänitutkimus kirjataan kortille, samoin kuin endokardiittitiimin moniammatillinen käsittely. Kortille kirjataan potilaan mahdolliset altistavat tekijät, pääasialliset oireet ja infektoituneet kohteet. Myös sydänläpästä tehdyt mikrobitutkimukset löydöksineen kirjataan kortille. Kortin avulla voidaan seurata mikrobilääkehoidon kestoa ja mahdollisia hoidon aikaisia

komplikaatioita. Korttiin kirjataan myös sairaalajakson jälkeen suunnitellut kontrollit ja niiden toteuttamisyksiköt.

Endokardiittikortissa tuli ottaa myös huomioon se, että sitä tullaan hyödyntämään myös konsultoinnissa ja tämän vuoksi korttiin lisättiin välitallennuspainike. Välitallennuspainike sijoitettiin niin, että se on helposti saatavilla sen jälkeen, kun endokardiitin mahdollisuus on laskettu. Tässä vaiheessa konsulttilääkäri voi välitallentaa kortin ja sitten myöhemmin hoidon jatkuessa sitä voidaan edelleen täydentää, muun muassa mikrobitutkimusten tuloksilla, hoidon seurannalla ja muilla seurantatiedoilla. Tällainen lisätoiminnallisuus laajentaa kortin käyttömahdollisuuksia ja helpottaa kortin käyttäjän työtä.

Korttia kehittäessä oltiin aina tarvittaessa yhteydessä asiakkaaseen sähköpostitse ja tiedusteltiin tarvittavia lisätietoja. Ennen endokardiittikortin käyttöönottoa pidettiin asiakkaan kanssa demotilaisuus, jossa nousi esiin vielä muutamia muutos- ja lisäystoiveita. Nämä ehdittiin toteuttaa vielä ennen käyttöönottoa ja asiakas sai tarpeitaan vastaavan kortin käyttöön. Käyttöönoton yhteydessä endokardiittikortin toimintaa esiteltiin vielä uudelleen, jotta kortin käyttöönottaminen olisi mahdollisimman helppoa asiakkaalle. Muutaman kuukauden käyttökokemusten jälkeen asiakkaalta tuli vielä lisätoiveita korttiin, ja nämä muutokset toteutetaan osana normaalia Kardion kehitystyötä. Tähän opinnäytetyöhön niitä ei kuitenkaan enää sisällytetty, koska se ei olisi ollut aikataulullisesti mahdollista.

4 TOTEUTUKSESSA KÄYTETYT TEKNOLOGIAT

Sydäntietojärjestelmä Kardion rakenne voidaan karkeasti jakaa kolmeen eri osaan: tietokantaan, sovelluslogiikkaan ja graafiseen käyttöliittymään. Tietokantaa käytetään tiedon tallentamiseen. Sovelluslogiikka hoitaa tiedon käsittelyn ja sen siirtämisen tietojärjestelmän osien välillä. Käyttöliittymällä tarkoitetaan käyttäjälle näkyvää osuutta. (Chang 2016.)

Sydänsairaalan sydäntietojärjestelmä Kardio on toteutettu käyttäen tietokantana PostgreSQL-kantaa. Sovelluksen sovelluslogiikka on tehty Javalla ja käyttöliittymä Apache Wicketillä. Koska ne ovat käytössä jo valmiiksi Kardiossa, käytetään niitä myös endokardiittikortin toteutuksessa. Kehitysympäristönä oli käytössä IntelliJ IDEA, joka on muutenkin käytössä Kardion kehittämisessä. Versionhallintaan käytettiin GitLabia. Tässä luvussa esitellään sovelluksessa käytetyt tekniikat ja käsitellään tietokantoja myös yleisemmällä tasolla.

4.1 Tietokanta

Tietokanta on yksinkertaistettuna kokoelma toisiinsa sidoksissa olevaa tietoa. Esimerkiksi aiemmin käytössä olleet paperiset potilaskansiot ovat myös eräänlainen tietokanta. Tällaisen tietokannan haaste on siinä, että tieto ei ole niin helposti haettavissa ja sen hakeminen saattaa viedä kohtuuttomasti aikaa. Myös arkistotilojen tulisi suurentua kasvavien potilaskansioiden määrän mukaan. Sähköisestä tietokantajärjestelmästä tiedon hakeminen ja sen yhdistely on huomattavasti helpompaa ja nopeampaa. (Beaulieu 2020.) Tietokannalle voidaan asettaa tiettyjä vaatimuksia. Tietokannassa ei saisi esiintyä turhaa toistoa eli tiedon tulisi olla tallennettuna tietokannassa vain yhteen paikkaan. Tietoja tulisi pystyä hakemaan tietokannasta joustavasti ja erilaisin perustein. Tietokannan rakenteen muuttamisen tulisi olla joustavaa. Lisäksi tietokannasta löytyvän tiedon käytön ja eri sovellusohjelmien tulisi olla riippumattomia tietojen fyysisestä tallennusrakenteesta. (Lahtonen 2001, 2–3.)

4.1.1 Relaatiotietokanta

Relaatiotietokannassa tieto tallennetaan ja esitetään tauluina. Taulujen yhteneviä tietoja käytetään linkittämään taulut yhteen. Jokaisessa relaatiotietokannan taulussa on sarake, joka identifioi jokaisen erillisen rivin taulussa. Tätä kutsutaan yleensä pääavaimeksi *(engl. primary key)*. Pääavain voi koostua myös sarakkeiden yhdistelmästä, kuitenkin niin, että se on väistämättä jokaisella rivillä uniikki. Tällaista monen sarakkeen yhdistelmää kutsutaan yhdistelmäavaimeksi *(engl. compound key, composite*

key). Tietokantapalvelut tarjoavat automaattisen mekanismin, jonka avulla uniikkeja pääavaimia voidaan generoida ilman, että tarvitsee itse pitää kirjaa käytetyistä avaimista. (Beaulieu 2020.) Tämä automaattinen pääavaimen luominen on eri tietokannoissa hieman erityyppinen keskenään. Oracle- ja PostgreSQL-tietokannoissa tämä on toteutettu sekvenssimallilla, SQL Server ja MySQL käyttävät tunnisteen luomismallia. Ne eivät toimi ristiin, vaan pääavaimen luominen pitää tehdä sillä mallilla, jota kyseinen käytössä oleva tietokantapalvelu tukee. (Coelho & Kiourtzoglou, 11.) Osa tietokannan tauluista sisältää tietoa, joiden avulla voidaan yhdistää muita tauluja yhteen tietyn sarakkeen avulla. Tällainen sarake määritellään yleensä myös avaimeksi ja sitä kutsutaan vierasavaimeksi *(engl. foreign key)*. Jokaista vierasavainta pitää vastata perusavain viittauksen kohteena olevassa taulussa. (Lahtonen 2001, 12.)

Relaatiotietokantaa havainnollistaessa voidaan käyttää apuna käsitteellistä mallintamista eli entity-relationship-mallia. Mallista tehty kaavio on nimeltään ER-kaavio. Sen avulla voidaan graafisesti esittää tietokantataulut ja niiden väliset suhteet. ER-kaaviossa tietokantataulu viittaa aina yhteen sovelluksessa olevaan entiteettiin, ja kaaviosta näkyy taulussa käytetty pää- tai vierasavain. Eri tietokantatauluja yhdistävistä viivoista voidaan nähdä taulujen väliset suhteet. Suhteet voivat olla yksi yhteen, yksi moneen tai monta moneen -suhteita. Suhteet kertovat siitä, moneenko suhteeseen entiteetti voi osallistua tai tulee osallistumaan. (Biscobing 2019.)

4.1.2 Tiedon normalisointi

Tiedon normalisoinnilla pyritään siihen, että tietokannassa ei esiinny tietojen toistamista ja siitä mahdollisesti aiheutuvia ongelmia tietojen lisäämisessä, poistamisessa ja päivityksessä. Normalisoinnilla pystytään lisäämään tietokannan rakenteen selkeyttä, yhtenäisyyttä ja laajennettavuutta. (Lahtonen 2001, 30.) Normalisointi tapahtuu melko automaattisesti, jos noudatetaan seuraavia ohjenuoria: kohteen yhteyteen tallennetaan vain siihen suoraan liittyviä tietoja ja jokaisen tiedon päivitys tapahtuu vain yhteen paikkaan. (Lahtonen 2001, 31.)

Usein puhutaan normaalimuodoista, joita on ensimmäisestä viidenteen. Yleensä käytetään kolmea ensimmäistä, joskus myös neljättä, mutta viides on käytössä huomattavasti harvinaisempi. Normaalimuotoja pidetään ohjenuorina, joten normalisoinnin voi toteuttaa myös ilman näitä. Normalisoinnin periaatteet ovat kuitenkin samat, käytetään sitten normaalimuotoja tai ei. Koska vain kolme ensimmäistä

ovat laajasti käytössä, käsitellään ne lyhyesti tässä yhteydessä. Ensimmäisessä normaalimuodossa tarkastetaan, että taulussa ei ole samoja sarakkeita kahteen kertaan. Tässä vaiheessa luodaan myös omat taulut jokaiselle ryhmälle, joka sisältää tähän tauluun liittyvää tietoa. Toisessa normaalimuodossa huolehditaan, että kaikki tietokantataulun sarakkeet ovat täysin riippuvaisia koko pääavaimesta. Jos näin ei ole, luodaan omat taulut sellaisille arvoille, jotka eivät ole kokonaan riippuvaisia pääavaimesta. Lisäksi luodaan suhteet näiden uusien taulujen ja alkuperäisen taulun välille. Kolmannessa normaalimuodossa poistetaan sarakkeet, jotka eivät ole riippuvaisia pääavaimesta. (Chapple 2020.)

Normalisoidussa mallissa tietoa tallennetaan useisiin tietokantatauluihin niin, että tietty tieto olisi vain yhdessä paikassa. Tällä varmistetaan, että tietoa tietokantaan päivittäessä tarvitsee se päivittää vain yhteen paikkaan. Tietokantakyselyä tehtäessä tarvitaan kuitenkin paljon tietokannan suorituskykyä lukuisiin liitoksiin eri tietokantataulujen välillä. Tietokantakyselyjen suorituskykyä voidaan lisätä denormalisoinnilla. Siinä pidetään hyväksyttävänä sitä, että tietokannassa saattaa olla toisteista tietoa, koska tietokantakysely on paljon tehokkaampi vähemmillä tietokantataulujen välisillä tauluiitoksilla *(engl. join)*. Denormalisoinnin yhteydessä lisätään aina tietokantaan toisteista dataa yhteen tai useampaan tietokantatauluun. Tällä voidaan välttyä monilta tietokantakyselyn taululiitoksilta, jotka hidastavat tietokannan suorituskykyä. Käytännössä denormalisointi siis parantaa tietokannasta lukemisen suorituskykyä mutta heikentää samalla tietokantaan tallentamisen suorituskykyä. (Gaur 2020.)

Koska denormalisoidussa tietokannassa data voi muuttua useammassa eri paikassa, on oltava sovelluksen kehitysvaiheessa erityisen tarkkana ja kehitettävä käsittelyt, jotka estävät epäkonsistentin datan esiintymisen tietokannassa. Se voidaan tehdä muun muassa käyttämällä tietokannassa triggereitä, transaktioita tai proseduureja. Näitä ei kuitenkaan tämän opinnäytetyön yhteydessä käsitellä tämän tarkemmin. (Gaur 2020.)

4.1.3 PostgreSQL

PostgreSQL:n on kehittänyt Michael Stonebraker kollegoineen vuonna 1986. Tämän jälkeen PostgreSQL on kehittynyt paljon, ja uusia versioita uusine ominaisuuksineen on julkaistu. Nykyään se on yksi suosituimmista relaatiotietokannoista. (Peterson 2021.)

PostgreSQL on avoimeen lähdekoodiin perustuva tietokanta. Sitä pidetään vakaana, turvallisena ja skaalautuvana tietokantana (Ferrari & Pirozzi 2020, 9). PostgreSQL toimii lähes kaikkien käyttöjärjestelmien kanssa yhteen. PostgreSQL-tietokanta tukee ACID-transaktioita. Tämä tarkoittaa tietokannan ominaisuuksia, joilla varmistetaan tallennettavan datan eheys huolimatta virheistä, virtakatkoksista tai muista vastaavista ongelmista. Näillä tietokannan ominaisuuksilla tarkoitetaan muun muassa tukea vierasavaimiin, taululiitoksiin *(engl. join)*, näkymiin, proseduureihin *(engl. stored procedures)* ja triggereihin. (Singh 2018.)

PostgreSQL on ilmainen, eikä siihen tarvitse erillistä lisenssiä. Koska tietokanta perustuu avoimeen lähdekoodiin, on käyttäjillä mahdollisuus muokata, jakaa ja kehittää sitä omien tarpeidensa mukaan. PostgreSQL:ää pidetään helposti laajennettavana, koska käyttäjä voi luoda sinne muun muassa omia tietotyyppejä tai funktioita. PostgreSQL käyttää nimensä mukaisesti SQL-kieltä tietokantakyselyiden tekoon. (Sardjoski 2021.)

4.2 Java

Javan on kehittänyt Sun Microsystems. Alkuperäinen idea oli kehittää ohjelmointikieli sulautetuille järjestelmille. Nämä suunnitelmat kuitenkin muuttuivat, kun graafiset käyttöliittymät yleistyivät webselaimissa ja Javallekin avautui uusi markkina-alue. Tämä lisäsi Javan suosiota, ja se on levinnyt laajaan käyttöön. Vuonna 2009 Oracle osti Sun Microsystemsin. Oracle hyödyntää tuotteissansa laajasti Javaa, ja kaupan myötä Oracle pystyi varmistamaan Javan jatkokehittämisen. (Vesterholm & Kyppö 2018, 15–16.) Java on ollut olemassa 26 vuotta, ja silti se on edelleen yksi suosituimmista ohjelmointikielistä. Javan ehkäpä suurin etu on se, että sitä on kehitetty pitkään ja sen takia lähes kaikissa teknologioissa tai kirjastoissa on valmis integraatio Javaan. (Izydorczyk 2021.)

Java on ohjelmointikielenä puhdas oliokieli, jossa käytetään olioita ja luokkia. Javassa on käytössä vahva tyypitys, ja käytettäville muuttujille voidaan sijoittaa vain niiden tyypin mukaisia arvoja. Javan hyvinä puolina pidetään automaattista roskien keruuta, unicode-merkistön käyttöä ja virhetilanteiden käsittelyä. Automaattisella roskien keruulla tarkoitetaan sitä, että olion elinkaaren päättyessä vapautetaan niiden varaama muisti automaattisesti. Unicode-merkistön käyttö riittää useiden kielten merkkien näyttämiseen, koska merkkien skaala on niin laaja. Virhetilanteista voidaan ilmoittaa käyttäjälle aiheuttamalla poikkeus. Kääntäjä tarkistaa, että koodin mahdollisille poikkeuksen aiheuttamille kohdille on kirjoitettu myös käsittely. Tällöin jo useimpiin mahdollisiin virhetilanteisiin on otettava kantaa jo

koodausvaiheessa ja luotava näille virhetilainteille sopiva käsittely. (Vesterholm & Kyppö 2018, 18.) Tässä kohtaa on huomattava, että Javassa on olemassa kahdenlaisia poikkeuksia: tarkistamattomia *(engl. unchecked exception)* ja tarkistettuja *(engl. checked exception)* poikkeuksia. Suurin ero näiden kahden eri poikkeuksen välillä on se, että tarkistetut poikkeukset tarkistetaan jo käännösvaiheessa, mutta tarkistamattomat poikkeukset vasta ohjelman ajon aikana. Tarkistetuille poikkeuksille kehittäjän on luotava koodausvaiheessa käsittely. Ilman tätä käsittelyä ohjelma ei käänny, ja siitä ilmoitetaan virheviestillä. Tarkistamattomat poikkeukset syntyvät useimmiten ohjelman kanssa yhteensopimattoman datan vuoksi. Tällä tarkoitetaan sitä, että käyttäjä syöttää ohjelmaan dataa, joka ei sovi ohjelman kanssa yhteen, eikä kehittäjä ole tätä osannut ottaa huomioon koodausvaiheessa. Tärkeää olisikin kehitysvaiheessa ottaa huomioon kaikki mahdolliset erikoistilanteet datan syöttämisessä ja mahdollistaa vain tarkoituksenmukaisen datan syöttämisen ohjelmaan. (Singh 2012.)

Javan ja muiden olio-ohjelmointikielien perusyksikkö on olio, joka vastaa tai kuvaa jotain reaalimaailman asiaa sovelluksessa. Olion rakenne ja käyttäytyminen määritellään luokassa, ja ne yhdessä muodostavat kyseisen olion piirteet. Rakennetta määrittävät jäsenmuuttujat ja käyttäytymistä metodit. Jäsenmuuttujien saamat arvot kuvaavat olion tilan, ja tilaa voidaan muuttaa ohjelman suorituksen aikana. Jokaisella oliolla on uniikki identiteetti, joten vaikka kahden olion tila olisi sama, eivät ne kuitenkaan ole identiteetiltään sama olio. Metodit määräävät sen, mitä toimia olio voi suorittaa. Kaikilla luokan olioilla on samat metodit. Metodeilla voidaan muuttaa olion tilaa, tai ne voivat palauttaa jonkin arvon. Tyypillisesti olio piilottaa kaikki jäsenmuuttujansa ja tarjoaa käyttöön sellaisia julkisia metodeja, joilla olion tilaa voidaan kysyä ja muuttaa. (Vesterholm & Kyppö 2018, 76–77.)

4.3 Spring-sovelluskehys

Spring-sovelluskehys on avoimeen lähdekoodin perustuva alusta, jonka on kehittänyt Rod Johnson, ja se on julkaistu vuonna 2003 (Tutorials point 2016, 1).

Spring tunnetaan ehkä parhaiten riippuvuusinjektiosta *(engl. dependency injection, DI)*. Hallinnan muutossuunnan vaihtaminen *(engl. inversion of control, IoC)* on yleinen ohjelmistokehityksessä esiintyvä periaate, jossa vastuuta sovelluksen eri osien luomisesta ja niiden välisestä kommunikaatiosta voidaan siirtää sovelluskehykselle. Tämä helpottaa kehittäjän työtä, koska osa asioista voidaan antaa sovelluskehyksen tehtäväksi. Esimerkiksi kehittäjä luo luokat, mutta sovelluskehys on vastuussa olioiden

luomisesta luokkien pohjalta. (Spring-sovelluskehys.) Riippuvuusinjektio on yksi konkreettinen esimerkki hallinnan muutossuunnan vaihtamisesta. Kun tehdään sovellusta, Javan luokkien tulisi olla itsenäisiä toisistaan. Silloin luokkia voidaan uudelleenkäyttää ja tehdä testejä pelkästään kyseiselle luokalle, eivätkä muut luokat vaikuta siihen. Riippuvuusinjektio auttaa yhdistämään luokkia keskenään niin, että ne samalla säilyttävät itsenäisyytensä. Esimerkiksi jos on olemassa luokat A ja B ja A on riippuvainen luokasta B, hallinnan muutossuunnan vaihtamisen avulla B voidaan injektoida A luokkaan. Riippuvuusinjektio tarkoittaa sitä, että luokkien oliomuuttujat voidaan antaa luokan rakentajalle parametreinä tai käyttäen luokan `setter`-metodia. (Tutorials point 2016, 1.) Riippuvuusinjektio ja hallinnan muutossuunnan vaihtaminen saavat yhdessä aikaan tilanteen, jossa sovelluskehyksen vastuulla on luoda luokista olioita ja injektoida ne sovelluksen käyttöön. Näin voidaan vähentää olioiden välisiä riippuvuuksia. (Spring-sovelluskehys.)

Riippuvuusinjektion yhteydessä on huomioitava mahdollinen olioiden välinen syklinen riippuvuus. Syklinen riippuvuus tarkoittaa tilannetta, jossa Spring ei osaa päättää, mikä olio pitäisi luoda ensin ja injektoida toiseen. Esimerkiksi on olemassa luokat A, B ja C ja riippuvuudet ovat A → B → C. Spring luo luokan C, sen jälkeen B:n ja injektoi C:n B:hen. Sen jälkeen luodaan A ja injektoidaan B siihen. Tällaisessa tilanteessa Spring ei tiedä, mikä luokka sen pitäisi luoda ensin, ja aiheuttaa poikkeuksen *BeanCurrentlyInCreationException.* Tämä on mahdollista silloin, kun riippuvuudet annetaan luokan rakentajalle parametreinä. Muilla keinoin injektoitujen riippuvuuksien ei pitäisi muodostua ongelmaksi, koska ne injektoidaan vasta silloin, kun niitä tarvitaan, eikä sovelluskontekstia alustettaessa. (Baeldung 2022.)

Spring-sovelluskehys helpottaa transaktioiden hallintaa. Sisäisesti Spring käyttää tähän aspektiorientoitunutta ohjelmointia. Javan tietokantarajapinta JDBC *(engl. Java Database Connectivity)* tarjoaa transaktioiden hallintaan yhden mahdollisuuden `setAutoCommit`-funktion avulla. Tutkitaan asiaa kuvan 1 avulla tarkemmin. Ensin tarvitaan yhteys tietokantaan. `Try`-lohkossa aloitetaan transaktio. Kun asetetaan `setAutoCommit` arvoon `true`, toteutetaan jokainen SQL-lause omana transaktionaan. Jos taas arvo on epätosi, jokainen SQL-lause tulee erikseen vahvistaa ennen muutoksen voimaantulemista. Tämän vuoksi seuraavalla rivillä vahvistetaan muutokset. Tätä on hyödyllistä käyttää esimerkiksi silloin, jos tietokantaan päivitettävät tiedot ovat riippuvaisia toisistaan, ja jos vain toinen SQL-lause vahvistetaan onnistuneesti, päätyy tietokantaan epäjohdonmukaista dataa. Jos transaktiossa tulee poikkeus, päädytään `catch`-lohkoon ja tällöin peruutetaan kyseinen transaktio. (Behler 2019.)

```
import java.sql.Connection;

Connection connection = dataSource.getConnection();

        try(connection) {
        connection.setAutoCommit(false);
        // Execute SQL statements
        connection.commit();
        } catch (SLQException e) {
        connection.rollback();
        }
```

KUVA 1. Javan tietokantarajapinnan transaktioiden hallinta

Transaktioiden hallinta tarkoittaa siis käytännössä sitä, miten transaktio aloitetaan, vahvistetaan tai perutaan. Spring-sovelluskehyksessä on käytössä `@Transactional`-annotaatio, jonka avulla nämä vaiheet voidaan toteuttaa, ja lisäksi se tuo lisämahdollisuuksia transaktioiden käsittelyyn. Tämän annotaation käyttö mahdollistetaan sillä, että Spring konfiguraatio on annotoitu `@EnableTransaction-Management`-annotaatiolla ja siellä määritellään myös transaktioiden hallinta. Tämän jälkeen voidaan käyttää `@Transactional`-annotaatiota julkisissa metodeissa ja niiden sisältämät tietokantatransaktiot suoritetaan yhden transaktion sisällä. Kaikki tapahtuu aivan kuten JDBC:n esimerkissä, mutta paljon vähemmällä vaivalla, koska Spring suorittaa tämän ikään kuin automaattisesti. Tälle `@Transacti-onal`-annotaatiolle voidaan antaa lisämääreitä, jotka määrittävät lisää sen toimintaa. Oletuksena käytetään *Required*-määrittelyä, joka tarkoittaa sitä, että kyseinen metodi tarvitsee uuden transaktioyhteyden. *Supports*-määrittelyllä voidaan määrittää se, että ei ole väliä, onko transaktio auki vai ei, kumpikin toimii kyseisen metodin kanssa. *Mandatory*-määrittely tarkoittaa, että metodi ei avaa transaktiota itse, mutta tarvitsee jonkun muun avaamaan sen. *Required new*-määrittely kertoo, että metodi tarvitsee täysin oman transaktioyhteyden. *Not_Supported*-määrittely määrittää, että kyseinen metodi ei halua transaktioita, ja yrittää jopa keskeyttää tai estää jo käynnissä olevan transaktion. *Never*-määrittely kertoo, että metodi ei missään tapauksessa halua transaktioyhteyttä. *Nested*-määrittely tarkoittaa sitä, että metodilla on tallennuspisteitä *(engl. savepoint)* eli käytännössä JDBC:n transaktiossa käytettäisiin tässä kohdassa merkintää `connection.setSavepoint()`. Tämän avulla voidaan määritellä tallennuspisteitä, joissa dataa tallennetaan huolimatta siitä, meneekö koko transaktio onnistuneesti loppuun asti. (Behler 2019.)

Kaikkia Spring-sovelluskehyksiä käyttäessä voidaan käyttää myös `@Transactional`-annotaatiota. Ainut ero on, että Spring Boot tekee annotaation `@EnableTransactionManagement` automaattisesti konfiguraatioon, eikä sitä tarvitse kehittäjän erikseen määritellä. (Behler 2019.)

4.4 JPA

Java tarjoaa jo aiemminkin mainitun tietokantarajapinnan JDBC:n eli Java Database Connectivity-rajapinnan. JDBC saa tietokantojen käsittelyn näyttämään samalta, huolimatta siitä, mikä tietokanta on kyseessä. Käytettäessä JDBC:tä ohjelmassa muuttuu yhteyden kantaan muodostava osa sen mukaan, mikä kanta on käytössä. Käytettäessä pelkkää JDBC:tä tietokantaohjelmointiin, muodostuu aikaa vieväksi kirjoittaa koodi, joka muuttaa tietokannasta tulevat tiedot olioiksi ja päivittää niiden tilaan tehdyt muutokset tietokantaan. (Vesterholm & Kyppö 2018, 517–518.)

Olio-relaatiomallinnus tarkoittaa relaatiotietokannan tietorakenteen ja oliomallin yhteensovittamista. Tässä haasteena ovat oliomallin ja relaatiotietokannan erot *(engl. impendance mismatch)*. (Haack 2004.) Rakenteiden ero on nähtävissä siinä vaiheessa, kun dataa halutaan tallentaa tai hakea tietokannasta ja olioiden jäsenmuuttujien arvot tulee siirtää tietokannassa niitä vastaavaan tauluun. Luokat eivät välttämättä sovi sellaisenaan relaatiotietokannan tauluihin, koska luokan rakenne voi olla sellainen, että se vaatii datan tallentamista useampaan tietokantatauluun. Relaatiotietokannassa kaikki taulut ovat samanarvoisia. Tämän seurauksena voidaan helposti saada tietokannasta dataa, joka on oliohierarkiassa syvälläkin. Lisäksi tämä vähentää syklisen riippuvuuden riskiä. Toinen huomionarvoinen seikka on se, että olioiden kanssa käytetään usein periytyvyyttä. On tavallista, että luokka periytyy toisesta, ja samalla lisätään periytyvälle luokalle jäsenmuuttujia. Relaatiotietokanta ei pysty tekemään näin, toinen tietokantataulu ei voi periytyä toisesta. Tietokannassa tarvitaan tällaisen tilanteen ratkaisemiseksi yleensä kaksi tietokantataulua. Toinen sisältää datan alkuperäisestä luokasta ja toinen lisäominaisuudet periytyvästä luokasta. Tämä täytyy toimia tietenkin molempiin suuntiin eli tietoa tietokannasta haettaessa voidaan käyttää tauluiitoksia, jotta saadaan haettua tietokantatauluista kaikki jäsenmuuttujat ja yhdistettyä ne oikeisiin luokkiin. (Walker 2021.)

Tämän edellä kuvatun epäsopivuuden ratkaisemiseksi on tehty olio- ja relaatiomallin välisen kuilun täyttäviä välineitä, joista yksi on JPA eli Java Persistence API. (Vesterholm & Kyppö 2018, 536.) Lyhyesti sanottuna JPA hoitaa tietokantataulujen yhdistämisen koodissa oleviin luokkarakenteisiin. JPA:n implementaatioita on olemassa useita, sekä ilmaisia että maksullisia. Näistä esimerkkeinä ovat Hibernate, OpenJPA, EclipseLink ja Batoo. (Coelho & Kiourtzoglou, 4.) Kardiossa käytetään Hibernatea, joten se on käytössä myös endokardiittikortin toteutuksessa.

JPA:n tärkein ominaisuus on sen kyky yhdistää tietokantataulun sarake ja luokan muuttuja yhteen huolimatta siitä, miten ne on nimetty (Coelho & Kiourtzoglou, 4). Tämän mahdollistamiseksi on luotu entiteetti ja JPA käsittelee tietokantataulun muutoksia sen kautta (Coelho & Kiourtzoglou, 8).

4.4.1 Entiteetti

Java-luokkaa voidaan pitää entiteettinä, kun se noudattaa tiettyjä sääntöjä. Sääntöjä on kolme: luokan tulee olla merkitty annotaatiolla `@Entity`, luokalla tulee olla julkinen rakentaja ilman argumentteja, ja luokalla tulee olla kenttä, jossa on `@Id`-annotaatio, joka kertoo entiteetin yksilöllisestä tunnuksesta. Tunniste-kentällä ei tarvitse olla `get`- ja `set`-metodeja, koska JPA olettaa entiteetin tunnisteen olevan muuttumaton. (Coelho & Kiourtzoglou, 8–9.) Tunnisteen generoimiseen JPA tarjoaa automaattisia keinoja, mutta on muistettava, että sen valinta riippuu käytettävästä tietokannasta (Coelho & Kiourtzoglou, 11–12).

PostgreSQL käyttää sekvenssimenetelmää yksilöllisen tunnisteen luomiseen, joten sitä käytetään tässä opinnäytetyössäkin. Tällöin entiteetti tulee merkitä annotaatiolla `@SequenceGenerator` ja tälle määritellään käytettävän sekvenssin nimi. Tämä kertoo sen, että tietokannassa on olemassa kyseisen niminen sekvenssi tunnisteen luomiseksi. Tämä sekvenssi on luotava tietokantaan. Eri entiteeteille voidaan käyttää myös samoja sekvenssejä, mutta se ei ole suositeltavaa, koska silloin tunnistenumerot eivät mene tietokantatauluissa selkeässä numerojärjestyksessä. (Coelho & Kiourtzoglou, 11–12.) Sekvenssille voidaan määritellä myös aloitusarvo, joka määrittää ensimmäisen arvon, jonka sekvenssi antaa. Toinen määriteltävä arvo on allokaation koko, joka tarkoittaa sitä määrää generoituja tunnisteita, jotka ladataan tietokannan välimuistiin. Tällä voidaan jonkin verran optimoida muistin käyttöä, koska tietokannan ei tarvitse joka kerralla luoda uutta tunnistenumeroa. (Coelho & Kiourtzoglou, 13.)

Annotaatiot voidaan jakaa loogisiin ja fyysisiin annotaatioihin. Loogiset annotaatiot määrittelevät sen, miten Java-luokka käyttäytyy entiteettinä. Esimerkkinä loogisesta annotaatiosta ovat muun muassa `@Entity`, `@Id` ja `@OneToMany`. Fyysisiä annotaatioita ovat puolestaan `@Table`, `@Column` ja `@Basic`. Nämä fyysiset annotaatiot luovat tietokannan ja entiteettin välisen suhteen. Niiden avulla on mahdollista määritellä tietokantaan liittyviä tietoja, kuten taulun tai sarakkeen nimen. (Coelho & Kiourtzoglou, 10.)

4.4.2 Entiteettien väliset suhteet

Entiteeteillä on suhteita toisiinsa ja nämä suhteet määritellään annotaatioiden avulla. Kuten aiemmin jo tietokantojen yhteydessä kerrottiin, suhteita voi olla yhden suhde yhteen, yhden suhde moneen, monen suhde yhteen ja monen suhde moneen. Tämä kertoo sen, montako entiteettiä voi liittyä toiseen entiteettiin. Suhteet voivat olla yhdensuuntaisia tai kahdensuuntaisia. (Coelho & Kiourtzoglou, 36.)

Yhdensuuntainen yhden suhde yhteen tarkoittaa sitä, että vain toinen, omistava entiteetti on tietoinen toisesta entiteetistä. Toisessa entiteetissä ei ole siis tietoa omistavasta entiteetistä. Yksi yhteen -suhteissa käytetään JPA:n annotaatiota `@OneToOne`. Tälle kerrotaan lisämääritteenä `@JoinColumn`-annotaation avulla mikä sarake on vierasavain toisesta entiteetistä. Annotaatio kertoo myös sen, kumman entiteetin tietokantataulussa vierasavain sijaitsee, ja kumpi on omistava entiteetti. Kahdensuuntainen yhden suhde yhteen tarkoittaa sitä, että molemmilla entiteeteillä on tieto toisistaan. Annotaation `@OneToOne` lisäksi käytetään lisämääritystä `mappedBy`, jolla kerrotaan omistava osapuoli ja se, missä tietokantataulussa vierasavain tulee sijaitsemaan (KUVA 2). Kuvan esimerkistä voidaan päätellä, että omistava entiteetti on `EndocarditisTeamMeeting` ja vierasavain sijaitsee `endocarditisTeamMeetings`-tietokantataulussa. (Coelho & Kiourtzoglou, 37–38.)

```
@OneToOne(mappedBy = "endocarditisOperation")
private List<EndocarditisTeamMeeting> endocarditisTeamMeetings = new ArrayList<>();
```

KUVA 2. Koodiesimerkki mappedBy-määrityksen käytöstä

Yhden suhde moneen tai monen suhde yhteen määritellään käyttäen annotaatioita `@OneToMany` ja `@ManyToOne`. Tällaista suhdetta käytetään silloin, kun entiteetillä on suhde listaan toisia entiteettejä. Käytettäessä `@ManyToOne`-annotaatiota käytetään myös annotaatiota `@JoinColumn`. Se entiteetti, joka sisältää `@ManyToOne`-annotaation on aina omistava entiteetti (KUVA 3). Kuvan esimerkissä omistava entiteetti on `EndocarditisTeamMeeting`. (Coelho & Kiourtzoglou, 39.)

```
@Entity
public class EndocarditisTeamMeeting implements Serializable {

    @Id
    private long id;

    @ManyToOne
    @JoinColumn(name = "endocarditisoperation_id")
    private EndocarditisOperation endocarditisoperation;
}
```

KUVA 3. Koodiesimerkki @ManyToOne-suhteesta

Suhteesta saadaan kahdensuuntainen lisäämällä toiseen entiteettiin annotaatio `@OneToMany` ja `mappedBy`-määritys. Tällöin annotaatio tulee olla määriteltynä listalle toisia entiteettejä. Kuten jo aiemmin todettiin, `mappedBy` määrittää omistavan entiteetin (KUVA 4.). Kuvan esimerkistä nähdään, että omistava entiteetti on `EndocarditisTeamMeeting`. (Coelho & Kiourtzoglou, 40.)

```
@OneToMany (mappedBy = "endocarditisOperation")
private List<EndocarditisTeamMeeting> endocarditisTeamMeetings;
```

KUVA 4. Koodiesimerkki @OneToMany-suhteesta

Monen suhde moneen vaatii tietokantaan uutta taulua, jotta voidaan yhdistää tunnisteet jokaisen suhteen välillä. Tämä uusi tietokantataulu sisältää ainoastaan tunnisteet kumpaankin entiteettiin. Tällöin entiteetissä käytetään `@ManyToMany`-annotaatiota, ja sen lisäksi `@JoinTable`-annotaatiota, jonka avulla voidaan määrittää tietokantatauluissa olevat sarakkeet, joissa entiteettien tunnisteet sijaitsevat. (KUVA 5.) Opinnäytetyön toteutuksessa ei ole käytössä monen suhdetta moneen, joten kuvassa esitetään kuvitteellinen tilanne, jossa endokardiittiin voi liittyä useita sydämen ultraäänitutkimuksia ja tutkimus voi liittyä useampaan endokardiittiin. Omistava entiteetti määritetään `@JoinColumn`-annotaation avulla. Kuvan esimerkissä omistava entiteetti on `EndocarditisOperation`-entiteetti. Kuvassa on määriteltynä myös tunnisteet yhdistävä tietokantataulu `@JoinTable`-annotaation avulla, jolle annetaan lisämäärittelynä kyseisen tietokantataulun nimi. Esimerkin suhde on yhdensuuntainen. (Coelho & Kiourtzoglou, 40–42.)

```
@ManyToMany
@JoinTable(name="endocarditisoperation_transesophagealechocardiographies",
        joinColumns=@JoinColumn(name="endocarditisoperation_id"),
        inverseJoinColumns=@JoinColumn(name="echocardiography_id"))
private List<EndocarditisOperationEchocardiography> transesophagealEchocardiographies; // TEEs
```

KUVA 5. Koodiesimerkki @ManyToMany-suhteesta

Jos edellisen esimerkin suhteesta halutaan kahdensuuntainen, lisätään `EndocarditisOperationEchocardiography`-entiteettiin `@ManyToMany`-annotaatio ja `mappedBy`-määrittely. (KUVA 6.) Nämä märittelyt kertovat sen, että omistava entiteetti on edelleen `EndocarditisOperation`-entiteetti. Jokaisessa suhteessa jompikumpi entiteeteistä on oltava omistava entiteetti. Jos `mappedBy`-määrittely jätetään monen suhde monesta pois, olettaa JPA molempien entiteettien olevan omistavia entiteettejä. (Coelho & Kiourtzoglou, 42–43.)

```
@ManyToMany(mappedBy = "transesophagealEchocardiographies")
private List<EndocarditisOperation> endocarditisOperations;
```

KUVA 6. Koodiesimerkki kahdensuuntaisesta @ManyToMany-suhteesta

4.4.3 Kokoelma

Joskus on tarpeen saada yhdistettyä lista arvoja entiteettiin niin, että nuo arvot eivät ole itsessään entiteettejä. Tämä onnistuu JPA:n `@ElementCollection`-annotaation avulla. Tämä annotaatio kertoo, että jäsenmuuttuja ei ole yksittäinen arvo, vaan kokoelma olioita. Kokoelmaan voi olla tarpeellista käyttää myös annotaatiota `@CollectionTable`, jonka avulla voidaan määritellä mihin tietokantatauluun kokoelma lisätään. Jos tätä ei määritellä annotaation avulla, lisää JPA automaattisesti tietokantaan uuden taulun ja nimeää sen luokan ja jäsenmuuttujan mukaan. (Coelho & Kiourtzoglou, 35–36.)

4.4.4 Enumeraatio

Enumeraatio on yleinen ohjelmointikielien rakenne, jossa muuttuja voi saada ainoastaan tiettyjä etukäteen määriteltyjä arvoja (Bauer, King & Gregory 2016, 89). Enumeraation muuttujat tulee merkitä `@Enumerated`-annotaatiolla, koska muuten Hibernate tallentaa muuttujan järjestysluvun eikä sen arvoa. Ongelmaksi tämä nousee silloin, jos muuttujia lisätään tai niiden järjestystä muutetaan myöhemmin. Tämä aiheuttaa haasteen enumeraatioiden käsittelyssä, koska niitä tulee ylläpitää tällöin sekä tietokannassa että koodipohjassa. Annotaation `@Enumerated` avulla voidaan käyttää `EnumType.String`-määrittelyä, jonka avulla muuttujille luodaan vakioarvot ja ne tallennetaan sellaisenaan. (Bauer, King & Gregory 2016, 90.)

4.5 Apache Wicket

Wicket on Java-pohjainen sovelluskehys web-käyttöliittymien toteuttamiseen, jonka on kehittänyt alun perin Jonathan Locke vuonna 2004. Wicket perustuu komponentteihin, joilla jokaisella on oma tilansa *(engl. state)*. Yksinkertaistettuna kehittäjä luo sivun, tarvittavat komponentit ja määrittää miten jokainen komponentti reagoi käyttäjän toimiin. (Atlassian Confluence.) Wicket-komponentit ja HTML yhdistetään yhteen wicket-tunnisteella (Sery, 2020). Komponentti voi olla esimerkiksi `Panel` tai `Label` (Alves 2016). Tämän lisäksi komponentti tarvitsee mallin *(engl. model)*, joka toimii ikään kuin julkisivuna, jonka kautta komponentti saa yhteyden dataan (Baeldung 2019). Wicket-komponentit ovat hierarkisia, joten Wicketin Java-komponenttien rakenteen tulee täsmätä HTML-komponenttien rakenteeseen.

4.6 GitLab

Git on versionhallintatyökalu, jonka avulla kehittäjät pysyvät ajan tasalla jokaisesta tehdystä muutoksesta. Tämä helpottaa kehittäjien samassa projektissa tapahtuvaa yhtäaikaista työskentelyä. Sen lisäksi on helpompaa kehittää uusia ominaisuuksia hajottamatta toimivaa versiota sovelluksesta. Kun ominaisuus todetaan toimivaksi, voidaan se lisätä toimivaan versioon ja näin ollen saadaan koko ajan turvallisesti kehitettyä sovellusta eteenpäin. Git:n on kehittänyt Linus Torvalds vuonna 2005. (Williams 2017.)

Jokaisella kehittäjällä on kopio koko kehitettävästä projektista. Tätä sanotaan paikalliseksi projektiksi *(engl. repository)*, koska se on vain kehittäjällä itsellään. Yleensä tästä otetaan uusi haara, johon kehitetään uutta ominaisuutta. Kun halutaan tallentaa sen hetkinen tilanne, vahvistetaan muutokset paikallisesti. Lopulta kun muutokset toteuttavat sen, mitä uudelta ominaisuudelta halutaan, voidaan tämä uusi haara siirtää viralliseen versioon projektista, jolloin se yhdistyy osaksi sitä. Tämä virallinen versio on palvelimella ja sitä kutsutaan etäversioksi *(engl. remote repository)*. (Williams 2017.) Git:n avulla on mahdollista säilyttää versioiden historiatietoa ja tarvittaessa palata myös vanhempaan versioon. Historiatietojen avulla voidaan myös verrata mitä muutoksia eri versioihin on tehty. Git säilyttää tiedon myös siitä, kuka on viimeksi muokannut tiettyä koodin osaa. (Quickscrum.)

GitLab on pilvipalvelu, jossa voidaan säilyttää Git:n projekteja. GitLab on siis hallintatyökalu Git:lle. GitLab tarjoaa sekä ilmaisia että maksullisia projekteja. Sen avulla pystytään seuraamaan tehtäviä

(engl. issue). (Gaba 2022.) GitLab:ssa pystytään muokkaamaan käyttäjien oikeuksia tehdä tiettyjä toimintoja perustuen heidän rooleihinsa. Oikeudet voidaan antaa myös esimerkiksi pelkkään tehtävienhallintaan, ilman että annetaan oikeutta lähdekoodiin. GitLab tarjoaa oman jatkuvan kehittämisen työkalun *(engl. Continuous Integration/Delivery, CI)*. Jos projektissa on käytössä jo jokin muu työkalu CI:n hallintaan, voidaan GitLab integroida käyttämään sitä. (Peham.)

Alun perin GitLab oli täysin ilmainen ja avoimeen lähdekoodiin perustuva palvelu. Myöhemmin vuonna 2013 siitä eriytettiin GitLab CE (Community Edition) ja GitLab EE (Enterprise Edition). (Gaba 2022.) GitLabin ovat kehittäneet Dimitriy Zaporozhets ja Valery Sizov vuonna 2011 (Peham). Gitlabin kanssa samanlaisia palveluita tarjoavat ainakin Github ja BitBucket (Gaba 2022).

5 SOVELLUKSEN TOTEUTUS

Tässä luvussa käydään läpi endokardiittikortin käytännön toteutusta. Käytettävät teknologiat valikoituivat jo Kardiossa valmiiksi käytössä olevien teknologioiden pohjalta, ja ne esiteltiin aiemmassa kappaleessa. Kovin yksityiskohtaisia ja pitkiä koodiesimerkkejä tai selityksiä ei voida tehdä paljastamatta liikesalaisuuksia. Siitä syystä toteutuksen käsittely on pyritty pitämään tarpeeksi pintapuolisena, mutta kuitenkin niin, että lukija saa tarvittavan laajan käsityksen opinnäytetyötä varten tehdystä toteutuksesta.

5.1 Kardion arkkitehtuurikerrokset

Arkkitehtuuri kuvaa sovelluksen keskeiset osat ja niiden väliset dynaamiset ja staattiset suhteet korkealla abstraktiotasolla (Vesterholm & Kyppö 2018, 60). Hyvin yleistä on käyttää kerroksittaista arkkitehtuuria, jossa sovelluksen rakenteessa on kerroksia, jotka hoitavat kukin tiettyä kokonaisuutta (KUVA 7). Ylempi kerros käyttää hyväkseen alemman kerroksen palveluja. (Vesterholm & Kyppö 2018, 540.) Kardiossa ja toteutetussa endokardiittikortissa on myös käytössä samanlainen kerroksittainen arkkitehtuuri. Koska ylempi kerros hyödyntää aina alemman kerroksen palveluja, käsitellään tässäkin yhteydessä eri kerrokset samassa järjestyksessä, tietokantakerroksesta alkaen.

KUVA 7. Kerroksittainen arkkitehtuuri

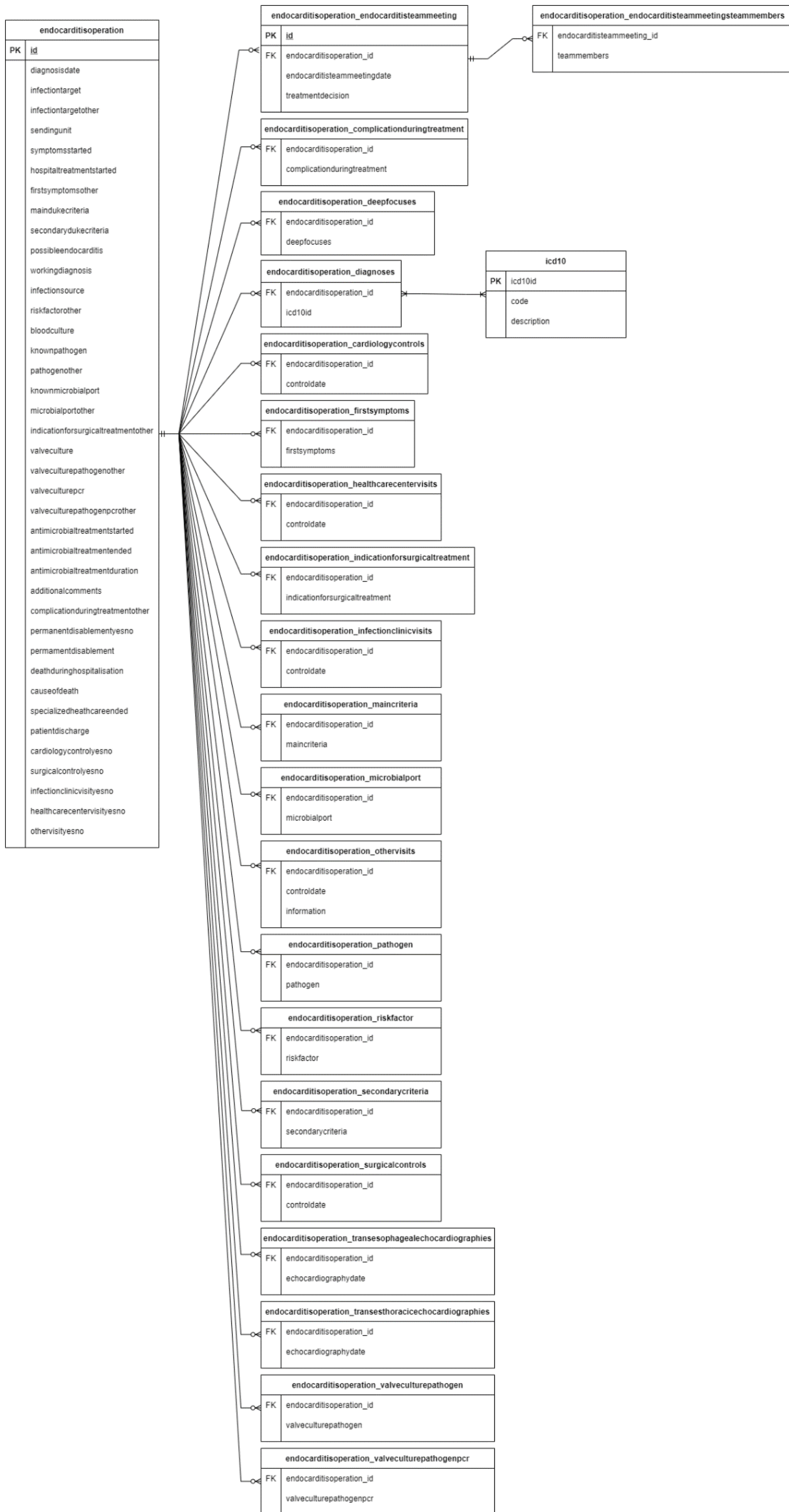
5.2 Tietokantakerros

Tietokanta on selkeä havainnollistaa ER-kaavion avulla. Kaaviossa tietokantataulu viittaa aina yhteen entiteettiin ja kaaviosta näkyy taulussa käytetty pääavain tai vierasavain. Eri tietokantatauluja yhdistävistä viivoista voidaan nähdä taulujen väliset suhteet: ovatko ne yksi yhteen, yksi moneen vai monta moneen -suhteita.

ER-kaaviota voidaan hyödyntää jo tietokannan suunnitteluvaiheessa. Tämän työn yhteydessä ER-kaaviota käytettiin esittämään endokardiittikorttiin liittyvät tietokantataulut visuaalisessa muodossa ja selkiyttämään tietokannan rakennetta kehitysvaiheessa (KUVA 8). Tietokantaa ja koko toteutusta kehitettiin pienissä osissa, ja kaavio helpotti hahmottamaan sopivan pienet kokonaisuudet kehitettäväksi.

Endokardiittikortin toteuttaminen lisäsi Kardion tietokantaan useita uusia tauluja. Taulujen ja jäsenmuuttujien nimeäminen noudatteli Kardion nimeämiskäytäntöä. Vapaatekstikentät nimettiin esimerkiksi pääteellä other. Tämä ei ulkopuoliselle suoraan kerro mitä jäsenmuuttuja sisältää, mutta Kardiota tunteville kyseinen nimeäminen on selkeä. Taulujen ja jäsenmuuttujien nimeämisessä sai käytettyä aikaa myös miettimiseen. Nimeämisen tuli noudatella Kardion käytäntöjä ja olla englanninkielistä. Nimeämisessä on hyvä käytäntö myös se, että jäsenmuuttujan nimi kertoo lähes suoraan, mitä kyseinen jäsenmuuttuja sisältää.

Endokardiittikorttia toteuttaessa ei toteutettu koko tietokantaa kerrallaan, vaan jaettiin toteutus pienempiin osiin sen varmistamiseksi, että lähes aina oli toimiva versio käytettävissä. Näin oli myös helpompi löytää mahdolliset ongelmakohdat, koska oli helpompi hahmottaa, mitä uutta oli tehty edellisen toimivan version jälkeen.



KUVA 8. Endokardiittikortin ER-kaavio.

Toteutuksessa hyödynnettiin paljon enumeraatioita. Kardiossa enumeraatiot ovat tallennettuna arvosarjoiksi, joten niitä on helppo lisätä ja muokata tietokantaan. Näiden avulla monivalintojen valinnat on helppo esittää käyttäjälle.

5.3 Sovelluslogiikkakerros

Endokardiittista tehtiin oma entiteettinsä, johon tulivat jäsenmuuttujiksi kaikki endokardiittikorttiin liittyvät muuttujat. Kokoelmat merkittiin annotaatiolla `@ElementCollection`, ja näin saatiin yhdistettyä lista kyseisiä arvoja entiteettiin niin, että nuo yhdistetyt arvot eivät ole itsessään entiteettejä (KUVA 9). Samassa yhteydessä määritetään se, miten kyseinen kokoelma halutaan ladata tietokannasta. Tämä tehdään määrittelemällä kokoelmalle `fetchtype`. Sen ollessa `eager` haetaan tiedot aina ja vastaavasti sen ollessa `lazy`, haetaan tiedot vain tarvittaessa. Endokardiitissa on `fetchtype` asetettu arvoon `eager`, koska halutaan varmistua, että tiedot ovat varmasti käytettävissä ilman ylimääräistä tietokantahakua. Samaa periaatetta käytettiin muihinkin endokardiitissa oleviin kokoelmiin.

```
@ElementCollection(fetch = FetchType.EAGER)
@Fetch(value = FetchMode.SUBSELECT)
private List<FirstSymptoms> firstSymptoms;
```

KUVA 9. Koodiesimerkki kokoelmasta

Tietokannassa käytettiin monessa kohtaa date-tietotyyppiä. Se ei suoraan toimi Javan `LocalDate`-tietotyypin kanssa yhteen. Tämän vuoksi koodissa määriteltiin annotaatiolla `@Type` käytettävä `LocalDate`-tyyppi (KUVA 10). Päivämääräkentissä haluttiin alustaa arvo suoraan täyttöhetken päivämäärään, jota voi sitten käyttöliittymältä vaihtaa halutessaan. Tämä helpottaa käyttäjää kortin täyttämisessä, koska oletuspäivämäärä on käyttöliittymällä jo valmiiksi näkyvissä.

```
@Type(type = "org.jadira.usertype.dateandtime.joda.PersistentLocalDate")
private LocalDate diagnosisDate = LocalDate.now();
```

KUVA 10. Koodiesimerkki type-annotaation käytöstä

Sovelluslogiikkakerroksen vastuulla on endokardiittikortin tietojen hakeminen. Siinä hyödynnettiin aiemmin toteutettuja yleisiä toteutuksia. Toimenpiteiden haku on merkitty annotaatiolla `@Transactional`, joka tarkoittaa käytännössä sitä, että toteutetaan joko kaikki transaktiossa tehtävät toiminnot tai ei mitään. Sovelluslogiikkakerroksen `getEndocarditisOperation` kutsuu `operationDao`:ta,

jonka avulla haetaan toimenpide tunnisteen perusteella, esimerkiksi silloin kun ladataan kortin paneeli näkyviin (KUVA 11).

```
@Override
@Transactional(readOnly = true)
public EndocarditisOperation getEndocarditisOperation(int operationId) {
    return operationDao.getOperationByIdDefaultSettings(EndocarditisOperation.class, operationId,
            subOperationSettingService.supportsSubOperations(EndocarditisOperation.TYPE));
}
```

KUVA 11. Koodiesimerkki service-kerroksesta

Kardiossa hyödynnetään paljon enumeraatioita ja ne ovat sijoitettuna arvosarjoihin tietokannassa. Jokaisella arvosarjalla tulee olla Kardiossa oma luokka toimiakseen. Endokardiittikortin toteutuksen yhteydessä luotiin enumeraatioita varten useita uusia luokkia. Luokkaan voidaan lisätä myös vakioarvoja sellaisille enumeraation arvoille, joita käytetään suoraan koodissa esimerkiksi näkyvyyden määrittämiseen tietyn valinnan perusteella tai muun toiminnallisuuden määrittämiseksi. Tällaiset enumeraatiot merkitään tietokantataulussa erillisellä `fixed`-sarakkeen `true`-arvolla, joka kertoo kehittäjille kyseisen arvon sisältävän logiikkaa koodissa. Samalla se suojaa arvoa ylläpitokäyttäjien toimilta. Luokassa määritelty GROUP ja SERIES vastaavat tietokannassa olevaa tiettyä tietokantataulun saraketta, jonka avulla enumeraatiot voidaan yhdistää oikeaan paikkaan (KUVA 12).

```
public class FirstSymptoms extends ValueSeriesValue {
    public static final String GROUP = "endocarditis";
    public static final String SERIES = "FirstSymptoms";
    public static final int OTHER = 4;
```

KUVA 12. Koodiesimerkki enumeraation luokasta ja vakioarvosta

Endokardiittikortin lisäksi toteutettiin myös hakusivu endokardiitille. Sen avulla käyttäjä pystyy hakemaan kaikki potilaat, joille on täytetty endokardiittikortti Kardiossa tai rajaamaan hakua tiettyjen perusvalintojen kautta. Asiakkaan toivomuksen mukaan tässä vaiheessa ei lisätty vielä mitään endokardiittikohtaisia valittavia hakuehtoja, vaan se on mahdollista toteuttaa myöhemmässä vaiheessa, jos se katsotaan tarpeelliseksi. Koodiin jätettiin siksi poiskommentoituna hakutermeille valmis paikka, johon ne tarvittaessa on helppo lisätä. Haku tehdään `CriteriaBuilder`:in avulla, jolloin saadaan haettavat toimenpiteet kokonaisuudessaan (KUVA 13).

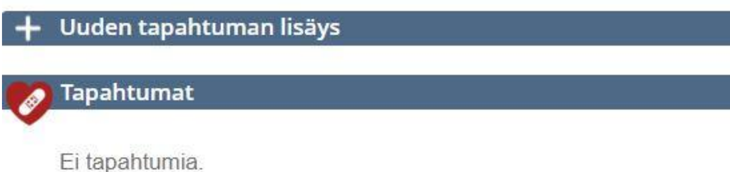
```
public class EndocarditisOperationCriteriaBuilder extends OperationSearchCriteriaBuilder<EndocarditisOperation> {

    public EndocarditisOperationCriteriaBuilder() { super(EndocarditisOperation.class); }

    @Override
    public void addWhereClauses(List<Predicate> wherePredicates, OperationSearchTerms operationSearchTerms,
                                Root<EndocarditisOperation> root, CriteriaBuilder cb) {
        // EndocarditisOperationSearchTerms terms = (EndocarditisOperationSearchTerms) operationSearchTerms;
    }
}
```

KUVA 13. Koodiesimerkki hakusivun lisäehdoista

5.4 Käyttöliittymäkerros

Käyttöliittymänäkymän suunnittelu tehtiin yhdessä asiakkaan kanssa heidän tarpeidensa pohjalta. Ulkoasumuotoilu perustui jo Kardiossa käytössä olevaan ulkoasuun ja endokardiittikortissa mukailtiin tätä olemassa olevaa toteutusta, jotta koko sovellus olisi yhtenäinen.

Kun potilas on valittuna, voidaan lisätä uusi tapahtuma, esimerkiksi endokardiittikortti, valitsemalla käyttöliittymästä uuden tapahtuman lisäys (KUVA 14).



KUVA 14. Käyttöliittymäesimerkki uuden tapahtuman lisäyksestä

Tämän jälkeen avautuu valikko, josta käyttäjä valitsee haluamansa tapahtuman. Kun endokardiittikortti on avautunut, pääsee käyttäjä täyttämään kysyttyjä tietoja (KUVA 15).

Endokardiitti - uusi tapahtuma

Diagnoositiedot

Dg pvm: 12.01.2022

Endokardiittidiagnoosi: Valitse kaikista koodeista...

Lisää uusi diagnoosi

1. TTE : Lisää uusi päivämäärä

1. TEE : Lisää uusi päivämäärä

Infektoitunut läppä/kohde: Valitse yksi

Kliininen kuva

Lähettävä/konsultoiva taho: Valitse yksi

Oireiden alkupäivä:

Sairaalahoidon alkupäivä:

Ensimmäiset oireet : ☐ Kuume ☐ Stroke ☐ Sydämen vajaatoiminta ☐ Muu tromboosi ☐ Muu

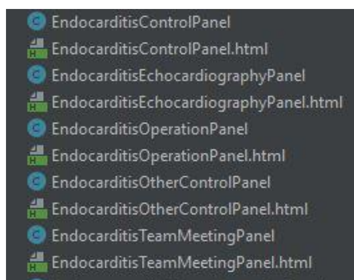
Täytä yksi tai useampi

Syvät fokukset (muu kuin läppä) : ☐ Keuhkot ☐ Perna ☐ Maksa ☐ Nivel ☐ Selkäranka ☐ Iho ☐ Aivot

Täytä yksi tai useampi

KUVA 15. Käyttöliittymäesimerkki endokardiittikortin alkuosasta

Käyttöliittymäkerros toteutetiin Apache Wicketillä, joka on muutenkin Kardiossa käytössä. Endokardiittikorttia varten tehtiin viisi eri paneelia, joiden avulla saatiin käyttöliittymä rakennettua ja samalla saatiin koodia jaoteltua osiin (KUVA 16).



KUVA 16. Endokardiittikortin Wicket-paneelit

Pohjana endokardiittikortissa on `EndocarditisOperationPanel`-luokka, joka sitten kutsuu muita paneeleita tarvitsemiinsa kohtiin `ListEditor`:in avulla (KUVA 17).

```
ListEditor<EndocarditisControl> cardiologyControlListEditor = new ListEditor<~>( id: "cardiologyControls",
        getOperationModel().bind( property: "cardiologyControls")) {

    @Override
    protected void populateItem(ListItem<EndocarditisControl> item) {
        item.add(new EndocarditisControlPanel( id: "cardiologyControl", item.getModel(), getMode()));

    }
};
```

KUVA 17. Koodiesimerkki paneeleiden kutsumisesta

`EndocarditisOperationPanel`-luokka jaettiin eri osioihin käyttämällä funktioita loogisesti kortin ulkoasun mukaan. Nämä funktiot nimettiin myös kuvaavasti niin, että funktion nimi kertoisi jo suoraan mitä kyseisessä funktiossa tehdään. Tämä helpottaa jatkossa myös kortin muokkaamista, kun koodi on selkeästi jaoteltu, eikä oikean kohdan etsimisessä mene liikaa aikaa (KUVA 18.)

```
@Override
protected void populateForm() {
    addDiagnosisInformation();
    addClinicalPicture();
    addBackroundInformation();
    addEnsureDiagnosis();
    addMicrobialDiagnostics();
    addTreatmentFollowup();
    addFollowupInformation();
}
```

KUVA 18. Paneelin funktiot

Jokaiselle osiolle tehtiin myös oma container-olio, johon muut kyseiseen kokonaisuuteen kuuluvat komponentit lisätään (KUVA 19). Myös tämä selkeyttää koodia ja auttaa jatkossa löytämään halutun muokattavan kohdan helpommin.

```
WebMarkupContainer diagnosisContainer = new WebMarkupContainer( id: "diagnosisContainer");
diagnosisContainer.setOutputMarkupId(true);
addFormComponent(diagnosisContainer);
```

KUVA 19. Koodiesimerkki container-olion lisäämisestä

Container-olioiden avulla pystytään myös tarvittaessa piilottamaan tai tuomaan esiin käyttöliittymälle eri näkymiä. Endokardiittikortissa näitä container-olioita hyödynnettiin paljon, koska kortilla oli useita valintoja, joiden toteutuessa haluttiin käyttäjälle tuoda joko uusia valintoja tai vapaatekstikenttä näkyville. Tällöin oliolle tulee asettaa ehto, millä käyttäjän valinnalla se tulee käyttöliittymälle näkyviin. Container-olioon voi lisätä tarvittavat muut komponentit, muun muassa vapaatekstikentän. Kuvan 20 esimerkissä on tehty juuri näin (KUVA 20). Käyttäjän valitessa infektoituneeksi kohteeksi valinnan

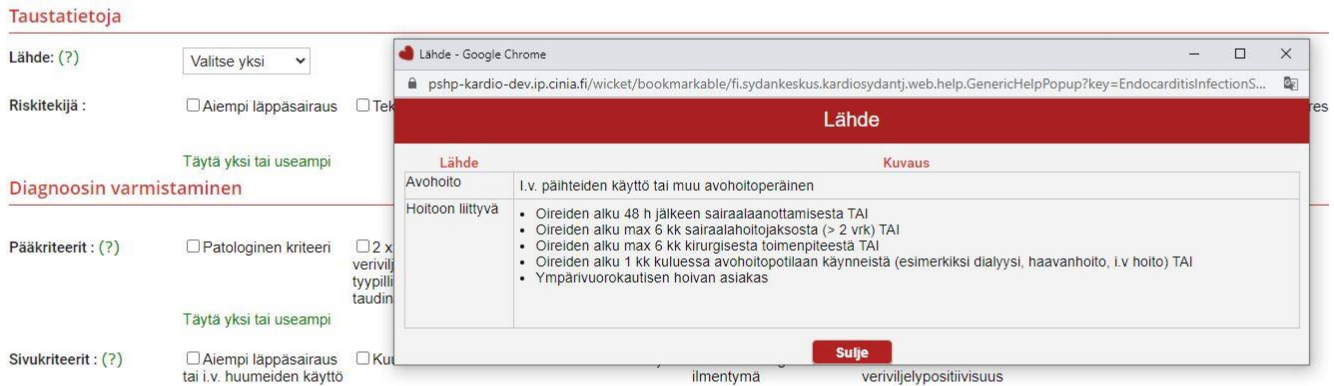
”muu”, halutaan käyttöliittymälle tuoda näkyviin vapaatekstikenttä, johon käyttäjä voi täydentää ja selittää valintaansa. Jos käyttäjä vaihtaa valintaansa ja poistaa valinnan ”muu”, piilotetaan vapaakenttä uudelleen näkyvistä. Tallennusvaiheessa tämä on otettu huomioon niin, että jos kyseisen esimerkin valinta ”muu” ei ole tallennusvaiheessa valittuna, ei myöskään vapaatekstikenttään mahdollisesti aiemmin syötettyä tekstiä tallenneta tietokantaan. Näin vältytään siltä, että tietokannassa olisi sinne kuulumatonta tietoa.

```
final WebMarkupContainer infectionTargetOtherContainer = new WebMarkupContainer("infectionTargetOtherContainer"
infectionTargetOtherContainer.add(new TogglableDropdownVisibleBehavior(infectionTarget, InfectionTarget.OTHER));
infectionTargetOtherContainer.setOutputMarkupPlaceholderTag(true);
diagnosisContainer.add(infectionTargetOtherContainer);

final MaxLengthTextArea infectionTargetOther = new MaxLengthTextArea<String>("infectionTargetOther");
infectionTargetOther.setOutputMarkupPlaceholderTag(true);
infectionTargetOtherContainer.add(infectionTargetOther);

infectionTarget.add(new AjaxFormComponentUpdatingBehavior("change") {

    @Override
    protected void onError(AjaxRequestTarget target, RuntimeException e) {
        if (e != null) {
            throw e;
        } else {
            infectionTarget.setModelObject(null);
            onUpdate(target);
        }
    }

    @Override
    protected void onUpdate(AjaxRequestTarget target) { target.add(infectionTargetOtherContainer); }
});
```

KUVA 20. Koodiesimerkki container-olion hyödyntämisestä

Endokardiittikorttin tuli paljon monivalintavaihtoehtoja ja alasvetovalikoita. Alasvetovalikot toteutettiin `ValueSeriesDropDownChoice`:n avulla, joka on Kardiota varten tehty ja se on Wicketin yleisen `DropDownChoice`:n aliluokka. Monivalintavaihtoehtokentät toteutettiin vastaavasti `ValueSeriesCheckBoxMultipleChoice`:n avulla, joka on samalla tavoin Kardiota varten tehty, ja se on Wicketin yleisen `CheckBoxMultipleChoice`:n aliluokka.

Endokardiittikortille lisättiin aputekstejä tarvittaviin kohtiin, jotka toimivat tukena kortin käyttäjälle kyseisissä kohdissa. Käyttöliittymässä kysymysmerkki indikoi aputekstiä, ja siitä painamalla käyttäjä saa aputekstin näkyviin. Aputeksti avautuu uuteen ponnahdusikkunaan (KUVA 21). Ponnahdusikkunan ja aputekstin ulkoasu määräytyi Kardion vastaavien samanlaisten kohtien mukaan niin, että kokonaisuus säilyy harmonisena läpi Kardion.



KUVA 21. Käyttöliittymäesimerkki pop up-ohjeen tarjoamisesta

Aputekstit lisättiin Kardiota varten rakennetulla `createGenericHelpPopupLink`-funktiolla, jonka avulla saadaan määriteltyä haluttu aputeksti ja avautuvan pop up:in koko (KUVA 22). Kyseinen funktio palauttaa Wicketin yleisen `BookmarkablePageLink`:n.

```
backgroundInformationContainer.add(infectionSource);
backgroundInformationContainer.add(kardioComponentFactory.createGenericHelpPopupLink( componentId: "infectionSourceHelp",
        titleKey: "endocarditis.infectionSourceHelp.title",  contentKey: "EndocarditisInfectionSourceHelp.html",  width: 910,  height: 250));
```

KUVA 22. Koodiesimerkki aputekstin lisäämisestä

Endokardiittikorttiin tehtiin myös automaattista laskemista ja täydennystä, jotta käyttöliittymä ja sen toiminta olisivat mahdollisimman käyttäjäystävällisiä. Duke-kriteerien ja endokardiitin mahdollisuuden laskeminen niiden perusteella ovat kenttiä, joita käyttäjän ei tarvitse itse täydentää, vaan ne lasketaan pää- ja sivukriteerien valintojen perusteella valmiiksi. Käyttöliittymässä kyseiset automaattisesti täydentyvät kentät indikoituvat käyttäjälle vaaleanharmaalla taustavärillä (KUVA 23).

KUVA 23. Käyttöliittymäesimerkki automaattisesta täydennyksestä

Endokardiittikortin lisäksi toteutettiin myös hakusivu, jossa käyttäjä voi hakea kaikki potilaat, joille on Kardiossa olemassa endokardiittikortti. Haku voidaan toteuttaa myös tietyillä hakuehdoilla (KUVA 24).

KUVA 24. Käyttöliittymäesimerkki hakusivusta

Hakusivu toteutettiin myös Wicketin paneelirakenteella ja endokardiittia varten tehtiin oma hakupaneeli, joka periytyy `OperationSearchPanel`-luokasta, ja joka lopulta toteuttaa Wicketin oman

yleisen paneeliluokan. `EndocarditisOperationSearchPanel`-luokassa määritellään hakuehdot haulle (KUVA 25).

```
public class EndocarditisOperationSearchPanel extends OperationSearchPanel {

    EndocarditisOperationSearchTerms searchTerms;

    public EndocarditisOperationSearchPanel(String id, EndocarditisOperation operation, EndocarditisOperationSearchTerms searchTerms) {
        super(id, operation, searchTerms);
        this.searchTerms = searchTerms;
    }

    @Override
    protected void populateForm() {
    }

    @Override
    public void resetPanel() { searchTerms.reset(); }
}
```

KUVA 25. Koodiesimerkki hakupaneelista

6 YHTEENVETO JA POHDINTA

Työn tekeminen oli erittäin antoisaa ja opettavaista. Olin toteuttanut aiemmin Kardioon pienempiä muutoksia, mutta kokonaisen uuden kortin toteuttaminen avasi uusia mahdollisuuksia oppia lisää. Itse kortin tekemisen ohessa pääsi kunnolla tekemään yhteistyötä myös asiakkaan kanssa. Koska työn lopputulos tuli oikeaan käyttöön, eikä ollut pelkkä demona tehtävä työ, tuntui sen tekeminen myös erittäin mielekkäältä. Endokardiitin teoriaan tutustuminen opinnäytetyötä kirjoittaessa avasi uudella tavalla kortin toteutusta ja asiakkaan toiveita. Tämä onkin hyvä pitää mielessä myös muita toteutuksia tehdessä ja käyttää jatkossa myös aikaa myös sovellusalaan tutustumiseen. Koen tämän mielekkäänä myös siksi, että asiat nivoutuvat hyvin kokonaisuudeksi, kun ymmärtää niistä molemmat puolet.

Asiakkaan kanssa yhteydenpito ja yhteistyö oli jouhevaa ja sujui kaikin puolin hyvin. Tarkentaviin kysymyksiin sain aina nopeasti vastauksen, eivätkä ne missään vaiheessa hidastaneet näin ollen työn toteutusta. Asiakas oli kortin lopputulokseen tyytyväinen. Myöhemmin korttiin päätettiin tehdä vielä joitain muutoksia, ja nämä tullaan toteuttamaan osana normaalia Kardion kehitystyötä. Endokardiittikortin jatkokehittäminen yhdessä asiakkaan kanssa jatkuu siis tämän opinnäytetyön jälkeenkin.

Koska teknologiavalinnat olivat samat kuin Kardiossa muutenkin, oli alkuun pääseminen helppoa ja pääsin keskittymään suoraan itse kortin toteutukseen. Olin jo aiemmin tehnyt suppeampia kehitystöitä Kardioon ja siksi työn aloittaminen oli kohtuullisen helppoa. Myös apua ja neuvoja kehitystyötä tehdessä oli tarvittaessa helposti saatavilla työnantajan puolesta. Tein korttia pienissä osissa, joten lähes aina oli toimiva versio saatavilla versionhallinnasta. Kortti on melko laaja ja siksi alun haasteena tuntuikin olevan sen laajuus, mutta tällä tavoin osissa kehittämällä sai laajuuden pilkottua pienimpiin osiin. Testasin omassa paikallisessa kehitysympäristössä aina muutosten jälkeen kortin toimivuutta. Kortin valmistumisen jälkeen se meni läpi normaalista koodikatselmoinnista ja testauksesta, joista palattiin vielä tekemään muutamia muutoksia kortin toteutukseen, jotta se toimisi halutulla tavalla. Kortin toteuttaminen toi sekä paljon oppia kehittämistyöstä ja Kardiosta että luottoa siihen, että pystyn hallitsemaan myös isompia kokonaisuuksia. Lisäksi ison kokonaisuuden toteuttaminen toi paljon uutta itseluottamusta osaamiseen ja helpotti uuden ammatillisen identiteetin kasvua.

Jatkokehityksenä kokonaisuuteen tullaan lisäämään jossain vaiheessa raportointiosuus, jonka sisältö määritellään yhdessä asiakkaan kanssa heidän tarpeisiinsa vastaavaksi. Raportointiosuuden avulla voidaan kootusti seurata endokardiittipotilaiden hoitoa ja käytäntöjä ja tarvittaessa korostaa raportoinnissa

jotain tiettyä osa-aluetta. Itse korttia tullaan varmasti myös kehittämään, kun asiakkaan tarpeet sen osalta muuttuvat ja täydentyvät.

LÄHTEET

Alves, E. 2016. Understanding the Apache Wicket basics. Saatavissa: https://medium.com/dev-trail/understanding-the-apache-wicket-basics-8bc4e353e370. Viitattu 26.1.2022.

Atlassian Confluence. About Wicket. Saatavissa: https://cwiki.apache.org/confluence/display/WICKET#Index-AboutWicket. Viitattu 25.1.2022.

Baeldung. 2022. Circular Dependencies in Spring. Saatavissa: https://www.baeldung.com/circular-dependencies-in-spring. Viitattu 18.2.2022.

Baeldung. 2019. Introduction to the Wicket framework. Saatavissa: https://www.baeldung.com/intro-to-the-wicket-framework. Viitattu 25.1.2022. LÄHDE B

Bauer, C., King, G. & Gregory, G. 2016. Java persistence with hibernate. 2. painos. Shelter Island: Manning Publications Co.

Beaulieu, A. 2020. Learning SQL. Generate, manipulate, and retrieve data. 3. painos. Sebastopol: O'Reilly Media.

Behler, M. 2019. Spring Transaction Management: @Transactional In-Depth. Saatavissa: https://www.marcobehler.com/guides/spring-transaction-management-transactional-in-depth. Viitattu 14.2.2022.

Biscobing, J. 2019. Entity Relationship Diagram (ERD). Saatavissa: https://searchdatamanagement.techtarget.com/definition/entity-relationship-diagram-ERD. Viitattu 31.1.2022.

Chang, R. 2016. Learning How to Build a Web Application. Saatavissa: https://medium.com/@rchang/learning-how-to-build-a-web-application-c5499bd15c8f. Viitattu: 13.10.2021.

Chapple, M. 2020. Database Normalization Basics. Saatavissa: https://www.lifewire.com/database-normalization-basics-1019735. Viitattu 3.2.2022.

Coelho, H. & Kiourtzoglou, B. Java Persistence API Mini Book. Java Code Geeks. Saatavilla: https://enos.itcollege.ee/~jpoial/java/naited/JPA_Mini_Book.pdf. Viitattu 02.12.2021.

Ferrari, L. & Pirozzi, E. 2020. Learn PostgreSQL. Birmingham: Packt Publishing Ltd.

Gaba, I. 2022. What is GitLab and How to Use It? Saatavissa: https://www.simplilearn.com/tutorials/git-tutorial/what-is-gitlab. Viitattu 16.2.2022.

Gaur, C. 2020. Data Denormalization – A New Way to Optimize Databases. Saatavissa: https://www.xenonstack.com/insights/data-denormalization. Viitattu 10.2.2022.

Haack, P. 2004. The meaning of Impendace Mismatch. Saatavissa: http://haacked.com/archive/2004/06/15/impedance-mismatch.aspx/. Viitattu 9.2.2022.

Izydorczyk, A. 2021. Java is dominating the programming landscape. Saatavissa: https://espeo.eu/blog/java-programming-landscape/. Viitattu 14.2.2022.

Kettunen, R. 2020. Endokardiitti (sydänläppien tulehdus). Lääkärikirja Duodecim 3.12.2020. Saatavissa: https://www.terveyskirjasto.fi/dlk00679. Viitattu: 17.12.2021.

Lahtonen, T. 2001. SQL. Jyväskylä: Docendo.

Peham, T. GitLab vs GitHub: Key differences & similarities. Saatavissa: https://usersnap.com/blog/gitlab-github/. Viitattu 16.2.2022.

Peterson, R. 2021. What is PostgreSQL? Introduction, Advantages & Disadvantages. Saatavissa: https://www.guru99.com/introduction-postgresql.html. Viitattu 11.2.2022.

Saraste, A., Turpeinen, A. & Hohenthal U. 2016. Endokardiitti. Teoksessa Airaksinen, J., Aalto-Setälä, K., Hartikainen, J., Huikuri, H., Laine, M., Lommi, J., Raatikainen, P. & Saraste, A. (toim.). Kardiologia. 3. uudistettu painos. Helsinki: Kustannus Oy Duodecim, 895–910.

Sardjoski, K. 2021. Advantages of using PostgreSQL. Saatavissa: https://www.keitaro.com/2021/09/14/advantages-of-using-postgresql/. Viitattu 14.2.2022.

Sery, R. 2020. Five reasons you should use Apache Wicket. Saatavissa: https://dev.to/romansery/five-reasons-you-should-use-apache-wicket-1lij. Viitattu 25.1.2022.

Singh, C. 2012. Checked and unchecked exceptions in Java with examples. Saatavissa: https://beginnersbook.com/2013/04/java-checked-unchecked-exceptions-with-examples/. Viitattu 10.2.2022.

Singh, G. 2018. PostgreSQL Deployment in Kubernetes with Deployment Strategy. Saatavissa: https://www.xenonstack.com/blog/postgresql-deployment. Viitattu 11.2.2022.

Spring-sovelluskehys. Saatavissa: https://web-palvelinohjelmointi-19.mooc.fi/osa-1/3-spring-sovelluskehys. Viitattu 4.2.2022.

Suhonen, J., Halavaara, M., Lönnqvist, L., Teittinen, K. & Rajala, H. 2021. Infektiivinen endokardiitti. Lääketieteellinen aikakausikirja Duodecim 137(6), 563–574. Saatavissa: https://www.duodecimlehti.fi/duo16127. Viitattu: 17.12.2021.

Sydänsairaalan tarina. 2017. Saatavissa: https://www.sydansairaala.fi/hoitoon-sydansairaalaan/sydansairaalan-tarina/. Viitattu 24.1.2022.

Toiminnanohjausjärjestelmä laajenee Sydänsairaalan tarpeiden tahtiin. 2021. Saatavissa: https://www.cinia.fi/referenssit/toiminnanohjausjarjestelma-laajenee-sydansairaalan-tarpeiden-tahtiin. Viitattu 3.2.2022.

Tutorials Point 2016. Spring Framework. Saatavissa: https://www.tutorialspoint.com/spring/spring_tutorial.pdf. Viitattu 4.2.2022.

Vesterholm, M. & Kyppö, J. 2018. Java-ohjelmointi. 10. uudistettu painos. Helsinki: Alma Talent.

Walker, J. 2021. What’s an ”Impendance Mismatch” in Programming? Saatavissa: https://www.cloudsavvyit.com/10526/whats-an-impedance-mismatch-in-programming/. Viitattu 15.2.2022.

Williams, A. 2017. History of Git. Saatavissa: https://hackaday.com/2017/05/11/history-of-git/. Viitattu 10.2.2022.

Quickscrum. What is Git? What benefits does Git offer? Saatavissa: https://guide.quickscrum.com/git-guide/. Viitattu 15.2.2022