Full stack-projekti asiasanojen keräämiseen ja analysointiin

Riitta Väisänen

Haaga-Helia ammattikorkeakoulu

Amk-opinnäytetyö

2022

Tradenomin tutkinto

Tiivistelmä

Tekijä(t)
Riitta Väisänen

Tutkinto
Tradenomi

Raportin/Opinnäytetyön nimi
Full stack-projekti asiasanojen keräämiseen ja analysointiin

Sivu- ja liitesivumäärä
28 + 2

Opinnäytetyö esittelee vapaaehtoisvoimin järjestettävän Kajo 2022-partioleirin leirin aikaisten työtehtävien allokointiin tuotetun web-sovelluksen ja analysointiprosessin. Leiriorganisaatio tilasi työkalun, jolla osallistujat voivat ilmaista mieltymyksensä, joiden perusteella työtehtävät jaettiin. Web-sovellus julkaistiin joulukuussa 2021 ja otettiin pois käytöstä helmikuussa 2022.

Kokonaisuus suunniteltiin hyödyntäen ketteriä kehitysperiaatteita prosessiksi, joka kehitetään, julkaistaan, ajetaan ja poistetaan käytöstä lyhyen ajan sisällä. Se koostuu käyttäjäkerroksesta (front-end), tarkistavasta kerroksesta (back-end), tietokannasta, allokointiperusteet luovasta algoritmista sekä leiriorganisaatiolle annetusta allokoinnit sisältävästä Excel-tiedostosta.

Työtehtävien (’pestien’) allokointiperiaate perustui pestille annettuihin avainsanoihin. Osallistuja saattoi valita samoista avainsanoista mieleisensä React-kirjastolla toteutetusta front-endistä, jonka jälkeen tiedot lähetettiin tarkistettavaksi back-endiin. Tämän jälkeen tiedot tallennettiin tietokantaan.

Pestien tiedot avainsanoineen kerättiin leiriorganisaatiolta ja käsiteltiin ennen tallennusta tietokantaan. Tämän jälkeen ajettiin allokointiperusteet antava algoritmi, joka tuotti jokaista osallistujaa ja pestiä kohden yhteensopivuusluvun, joka tallennettiin tietokantaan.

Hyödyntämällä tietokantanäkymää, jokaista osallistujaa kohden luodut pestikohtaiset yhteensopivuusluvut vietiin Excel-tiedoston taulukkoon. Taulukosta muodostettiin pestin ja osallistujan näkökulmasta 2 Pivot-taulukkoa, joiden perusteella jokaiselle osallistujalle allokoitiin tietty pesti.

Allokoinnin jälkeen Excel-tiedostoon yhdistettiin osallistujien henkilötiedot ja tiedosto luovutettiin leiriorganisaation käyttöön.

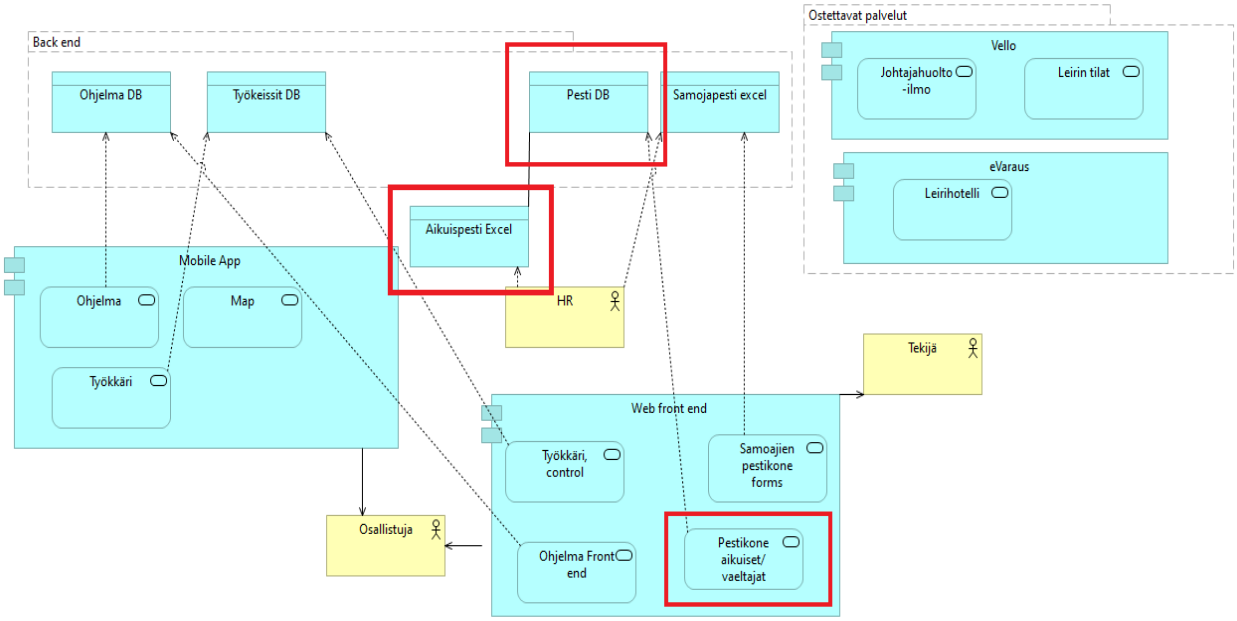
Asiasanat
Ohjelmointi, resursointi, tietokannat, suunnittelu, verkkopalvelu

Sisällys

1 Johdanto

Suuressa vapaaehtoisorganisaatiossa työvoiman resursointi ja kohdennus on usein hankala ja mittavaa käsityötä vaativa prosessi. Ongelma korostuu entisestään tilanteissa, missä myös prosessia suorittavat ovat yhtä lailla vapaaehtoisia. Tässä opinnäytetyössä esitetään yksi ratkaisutapa työtehtävien allokointiin vapaaehtoisorganisaation suurtapahtuman kontekstissa hyödyntäen työtehtäville määritettyjä tunnisteita. Opinnäytetyö toteutetaan toiminnallisen opinnäytetyön muodossa.

Työn kohteena on Suomen Partiolaiset – Finlands Scouter ry:en kesällä 2022 järjestettävän 8. Finnjamboree Kajo, tästä eteenpäin Kajo, ja erityisesti sinne tuotettavan digipalvelukokonaisuuden osa. Toimin osana organisaation 6-henkistä digipalvelutiimiä sovelluskehityskympin pestissä vapaaehtoisena. Käytännössä olen vastuussa leirin digipalveluiden suunnittelusta, toteutuksesta sekä jälkikäsittelystä yhteistyössä 2 muun ohjelmoijan, designerin ja mobiili- ja pilvipalveluvastaavan kanssa. Koska kokonaisuus on laaja (Kuva 1), tässä opinnäytetyössä keskitytään ’pestikoneen’ osakokonaisuuden, johon liittyvät osat kuvattu alla punaisella, suunnitteluun ja toteutukseen. Pestikone on loogisesti erillinen kokonaisuus niin ajallisesti kuin arkkitehtuuriltaan. Sen tulee olla tuotantovalmis osallistujia varten ilmoittautumisen alkaessa joulukuussa 2021, toisin kun muut järjestelmät, joiden tulee olla tuotantokäytössä vasta itse leirin aikana heinäkuussa 2022. Toimintatarkoitus on myös merkittävästi rajatumpi. Lyhyesti ilmaistuna, se suunnitellaan, tuotetaan, viedään tuotantoon, ’ajetaan’ ja otetaan pois tuotannosta analyysin valmistuessa.

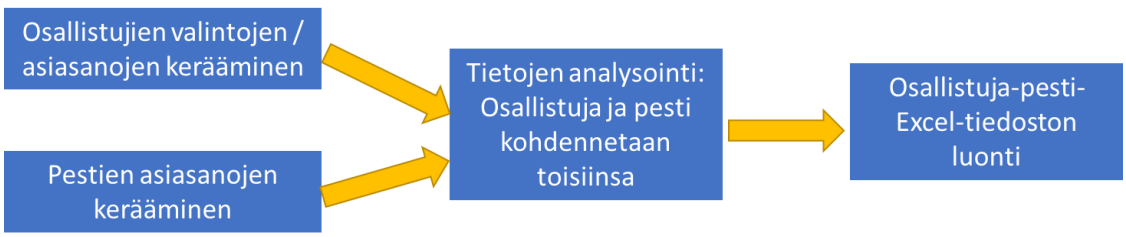


Kuva 1. Kajon kartoitetut digipalvelutarpeet. Opinnäytetyön käsittelemät osat punaisella

Kuten yllä on jo mainittu, pestikoneen päätarkoitus on tehtävien allokointi. Kajon kaltainen tapahtuma, jonka varsinainen kohderyhmä on nuoret, vaatii ennen leiriä ja leirin aikana merkittävän määrän myös ns. aikuisia mahdollistajia huolehtimaan mm. ohjelmasuunnittelusta ja -toteutuksesta, muonituksesta, logistiikasta, rakentamisesta ja tietoliikenneyhteyksistä. Lukuun ottamatta muutamaan projektihallinnollista tehtävää kaikki toteutetaan vapaaehtoisvoimin. Leirin aikana jokaisella täysi-ikäisellä osallistujalla on jokin tehtävä, 'pesti'. Pestien allokointi on aiemmin tapahtunut pääsääntöisesti käsityönä ja suhteellisen sattumanvaraisin perustein. Tähän tilanteeseen leiriorganisaatio halusi muutoksen. Ratkaisuna leirin digipalveluilta tilattiin asiasanapohjainen valintametodi. Yleisellä tasolla asiasana on johonkin tietosisältöön tai muuhun kokonaisuuteen liitetty sisältöä kuvaava avainsana, jota hyödynnetään esim. sisällön luokitteluun tai määrittelyyn (Techopedia 2017). Pestikoneen tapauksessa jaettaville pesteille haluttiin antaa asiasanat, joiden perusteella niitä voitaisiin jakaa.

Leirin organisaatio jätti jakoprosessin tarkemmat toimintaperiaatteet leirin HR-osaston ja digipalveluiden päätettäväksi. Käytännössä pestikonekokonaisuuden tilaajana toimi HR-osasto, joka asetti reunaehdot ja määritteli tietosisällön, jättäen toteutuksen digipalveluiden ratkaistavaksi. Tilattuun kokonaisuuteen kuului koodattu ohjelmistokokonaisuus, tästä eteenpäin produkti, jolla tiedot kerättiin, sekä Excel-tiedoston, jossa olisi valmiina osallistuneiden kohdennetut pestit. Reunaehdoksi asetettiin tasa-arvon ilmoittautumisajankohdan näkökulmasta, eli ilmoittautumisajan alussa ilmoittautunut henkilö ei pääsisi 'valitsemaan' itselleen tiettyä mielenkiintoisinta pestiä, vaan kaikki ilmoittautuneet ovat samalla lähtöviivalla pestejä allokoitaessa.

Tässä opinnäytetyössä kuvataan ongelmaan suunniteltua kokonaisuutta, johon kuluu full-stack web-sovellus, sekä määritetyt prosessit tiedon analysointiin ja vientiin lopputuotteeksi, eli Excel-tiedostoksi (Kuva 2). Produktin perustarkoituksena on kerätä osallistujien mieltymyksiä asiasanojen avulla ja jakaa pestit näihin mieltymyksiin perustuen.



Kuva 2 Pestikonekokonaisuuden perusprosessi

Produktiin tilausluonteen ja siihen liittyvän organisaatiotaustan vuoksi tästä opinnäytetyöstä on rajattu pois vaatimusmäärittely, testaus ja käyttöönotto. Myös käyttöliittymän

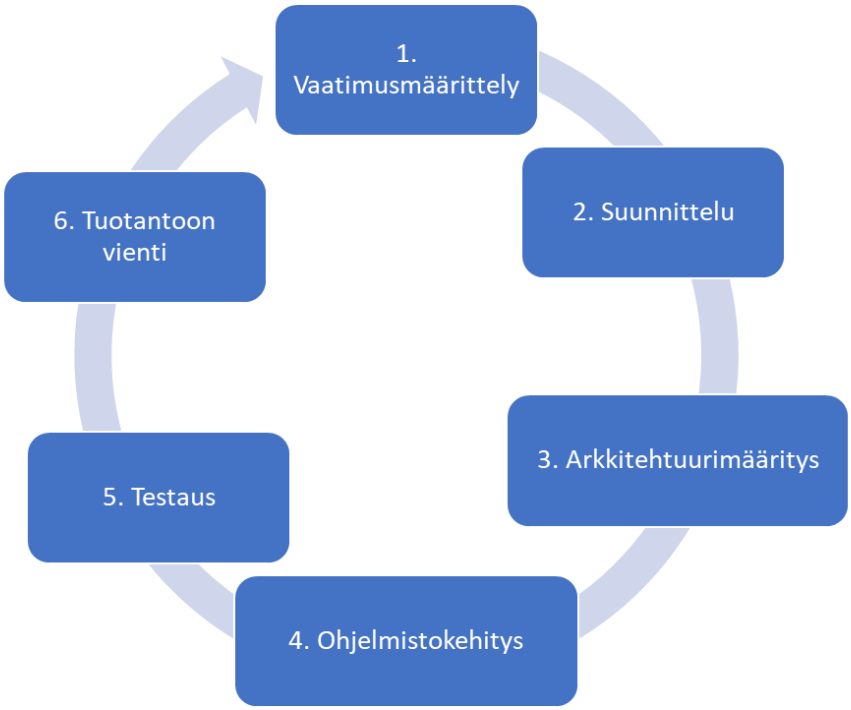
ulkoasun design-periaatteet oli luonnosteltu erikseen digipalveluiden designerin toimesta, joten niitä käsitellään vain toteutuksen näkökulmasta.

Työn rakenne etenee seuraavasti: Osassa 2 käsitellään produktin suunnitteluvaiheita ja kehitysperiaatteita yleisellä tasolla. Osa 3 käsittelee produktiin kuuluvaa tietokantaa, sen suunnittelua ja toteutusta, sekä myös allokointiin olennaisena osana kuuluvaa pestialgoritmia ja siihen liittyviä pohdintoja. Osassa 4 käsitellään produktin käyttäjille näkyvä kerros, eli front-end ja sen toteutusprosessi keskittyen tapoihin, jolla suunnitellusta konseptista toteutetaan asianmukaiset toiminnallisuudet ja ulkoasu. Osassa 5 käydään lyhyesti läpi produktiin kuuluva, serverless-logiikalla rakennettu back-end, jonka jälkeen osassa 6 avataan käyttäjiltä ja organisaatiolta kerätyn datan käsittelyprosesseja. Työ mukailee ns. vetoketjuperiaatetta, jossa luvun aiheeseen liittyvä teoria ja pohdinta löytyvät jokaisesta luvusta erikseen.

2 Produktin suunnitteluvaiheet

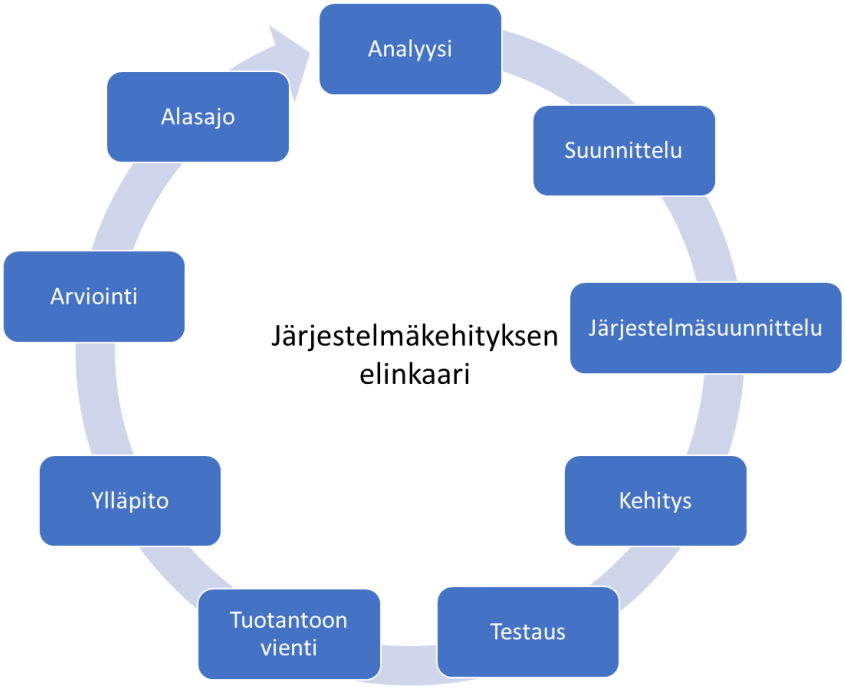
2.1 Ohjelmistokehityksen teoriaa

Ohjelmistokehitys kuvaa prosessia, jonka aikana kartoitetaan tarpeita, suunnitellaan ja toteutetaan ja saatetaan jokin ohjelmisto käyttöön (IBM 2022). Metodit jaetaan usein perinteiseen vesiputousmalliin ja ketterään kehitykseen. Vesiputousmallissa kaikki vaatimukset kartoitetaan ennen kehitystyön aloittamista (Adobe Workfront 2022). Ketterä kehitys (eng. Agile) kuvaa prosessia, jossa vaatimukset tarkentuvat kehitysprosessin aikana ja tuotekehitystä iteroidaan sykleissä ominaisuus kerrallaan (Inflectra, 2022). Agile-metodologiaa hyödynnetään pääsääntöisesti varsinaisen ohjelmistokehityksen aikana, eli ohjelmistokehityksen elinkaaren kohdassa 4 (Kuva 3).



Kuva 3. Ohjelmistokehityksen elinkaari (SDLC) (mukaillen Wanli 2020)

Ohjelmistokehitys voidaan myös ymmärtää osana laajempaa järjestelmäelinkaariajattelua, jota myös opinnäytetyön ohjelmistoprodukti edustaa. Mikään yksittäinen järjestelmä, tässä tapauksessa ohjelmisto, ei ole käytössä ikuisesti. Kuten järjestelmäelinkaaren kuvaus (Kuva 4) kertoo, jokainen järjestelmä suunnitellaan, tuotetaan ja viime kädessä poistetaan käytöstä.



Kuva 4. Järjestelmäkehityksen elinkaari (mukaillen Eby 2012)

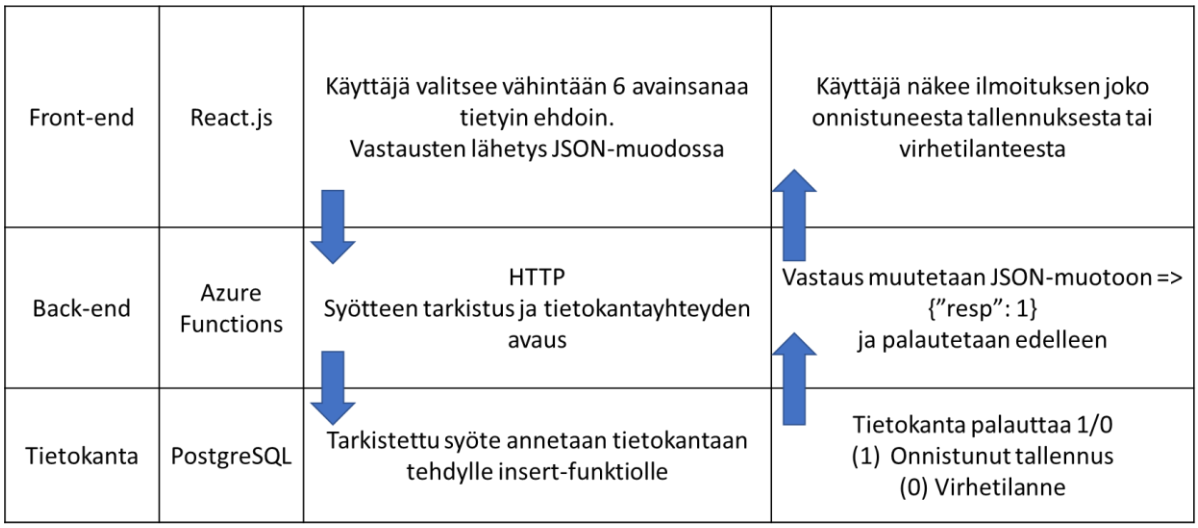
Ohjelmistoproduktin näkökulmasta kyseessä oli rajattu kokonaisuus, joka suunniteltiin, kehitettiin, vietiin tuotantoon ja ajettiin alas lyhyen aikaikkunan sisällä.

2.2 Pestikone-produktin suunnittelu

Pestikone-kokonaisuuden suunnittelu aloitettiin syksyllä 2021, tavoitteena tuotantoon vienti ja tallennuksen toimintavalmius joulukuun 2021 alkuun mennessä. Koska varsinainen tilattu lopputuote on Excel-tiedosto täytettyinä pesteillä ja henkilötiedoilla, tätä edellyttävän produktin suunnitteluprosessi päätettiin alusta alkaen pitää mahdollisimman yksinkertaisena ja ominaisuuksiltaan rajattuna. Varsinainen toteutus tehtiin vapaaehtoisperiaatteella muun elämän ohessa, joten monimutkainen toteutus olisi sitonut liikaa resursseja. Ylimääräisen suunnittelun tarvetta rajoitti myös pitkälle tehty organisaation päätöksenteko, joka toimi käytännön tasolla tarveanalyysina. Lyhyesti ilmaistuna, jos työtä tarkastellaan tarkemmin ohjelmistokehityksen elinkaarimallin näkökulmasta (Kuva 3), se kattaa pääosin osat 3, 4 ja 6. Jos taas produktia tarkastellaan laajemmin järjestelmän elinkaaren (Kuva 4) näkökulmasta, se kattaa myös alasajon. Produktia ei suunniteltu pitkäaikaiseen käyttöön, vaan kyseessä on kokonaisuus, jolla ratkaistaan tunnettu ja toistuva ongelma organisaation tällä kertaa määrittämin parametrein rajatussa aikaikkunassa. Vaikka produktikokonaisuus pyrittiin rakentamaan mahdollisimman uudelleenkäytettäväksi, sitä ei välttämättä käytetä enää uudelleen.

Varsinainen suunnittelu oli suoraviivainen. Produktin perustarkoitus oli kerätä tietoja myöhempää analyysia varten loppukäyttäjälle mahdollisimman helpossa muodossa. Produkti päätettiin toteuttaa julkisena internet-pohjaisena sovelluksena. Produktin luonne ennen kaikkea tiedonkeräämisen välineenä tarkoitti, että ratkaisun tulisi olla täysimittainen full-stack ratkaisu. Web-kontekstissa full-stack viittaa usein kolmikerrosrakenteeseen, jossa käyttäjälle näkyvä kerros (front-ent), tietoa käsittelevä kerros (back-end) ja tietoa tallentava kerros ovat kaikki loogisesti ja fyysisesti eriytetty toisistaan ja viestivät keskenään API-kutsuilla, eli dokumentoiduilla rajapinnoilla (IBM 2020). Tämä mahdollistaa myös kehitystyön eriyttämisen, sillä rajapinnat mahdollistavat jokaisen kerroksen hyödyntämisen ilman tietoa sen sisäisestä toteutuksesta.

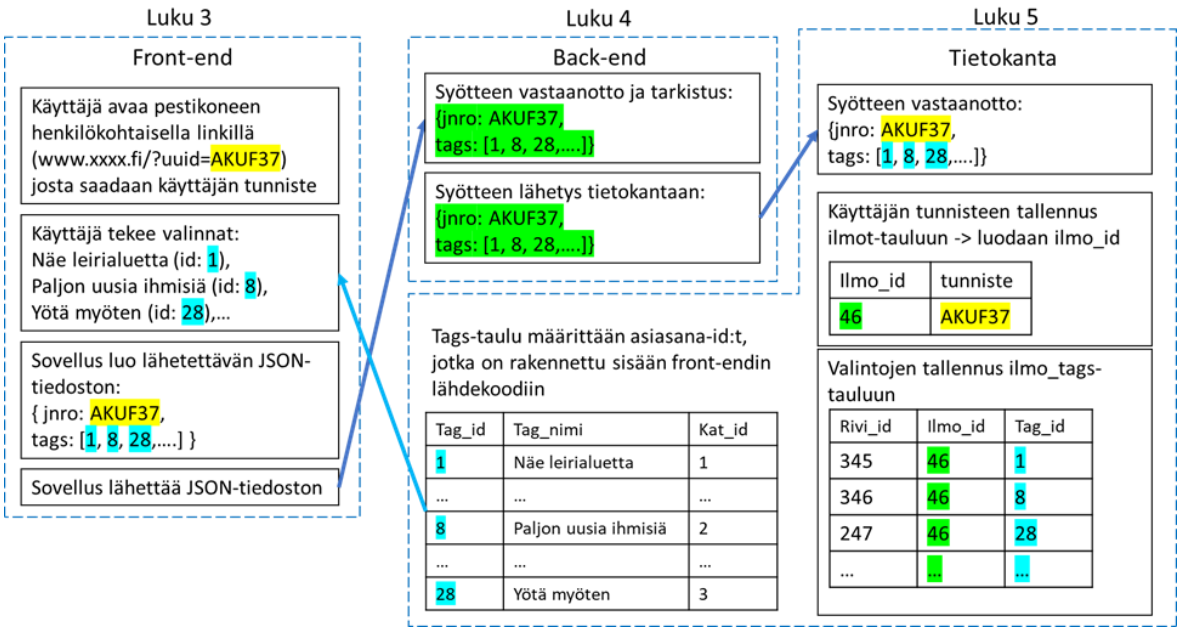
Pestikoneeseen vaadittiin siis käyttäjään päin näkyvä kerros (front-end), jonka käyttäjäkokemus olisi helppo ja suoraviivainen. Tiedot tulisi tallentaa helposti käsiteltävään muotoon analyysia ja mahdollisesti myös tilastointia varten. Välissä tulisi olla taustajärjestelmä (back-end), joka suorittaisi syötteen tarkistukseen ja varsinaisen tallennuksen. Tieto kaikkien eri osien välissä kulkisi JSON-muodossa, mahdollistaen mahdollisimman modulaarisen kehitysprosessin, eli jokaista osaa voitiin kehittää itsenäisesti toisista. Osallistujan kannalta merkittävä tiedonkulun koko prosessi löytyy kuvasta 5.



Kuva 5. Pestikoneen full-stack kokonaisuuden sisäinen tiedonkulku. Siniset nuolet kuvaavat JSON-datan siirtymistä eri kerroksissa

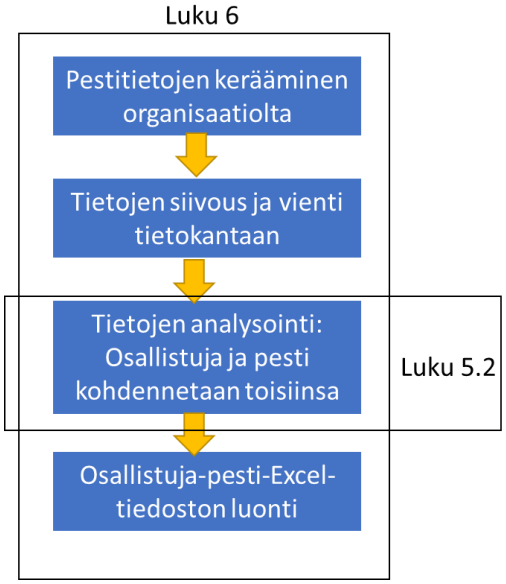
Ohjelmistokehitysvaiheessa pestikonekokonaisuuden tuotantometodina hyödynnettiin aikataulusyistä sekä ennakoiden muuttuvia vaatimuksia sovellettua agile-metodia. Käyttäjäkerros (front-end), josta koko prosessi aloitettiin, vaati myös kaikkein eniten aikaa ja sen toimintalogiikka edellytti usean iteraation läpikäymistä ennen halutun toiminnallisuuden varmistumista. Kehitysprosessin edetessä jouduttiin myös uudelleenneuvottelemaan sovelluskokoanaisuuden henkilötietojen käsittely, mikä edellytti kokonaisprosessiin kulkuun

liittyviä neuvotteluja ja salausominaisuuksien testausta. Lopputulemana salausprosessi hylättiin ja henkilötietojen käsittely siirrettiin prosessin loppuvaiheeseen. Lopullinen ohjelmistoproduktin sisäinen tiedonkulun prosessi muotoutui kuvan 6 mukaiseksi.



Kuva 6. Ohjelmistoproduktin sisäinen tiedonkulkuprosessi tarkentavine lukuineen

Tämä jälkeen kokonaisuuteen kuului vielä tiedon keräämistä ja analysointia, jonka yleisprosessi on kuvattuna kuvassa 7.



Kuva 7. Pestikoneen tiedonkäsittelyn kokonaiskuva tarkentavine

Lopputuloksena on prosessi, jonka loppuosa sisältää huomattavan määrän Excelissä tapahtuvaa käsityötä varsinaisessa allokointivaiheessa mutta vähentää merkittävästi allokoinnin sattumanvaraisuutta kertomalla selkeästi yksittäisen loppukäyttäjän ja tietyn pestin yhteensopivuuden.

3 Front-end

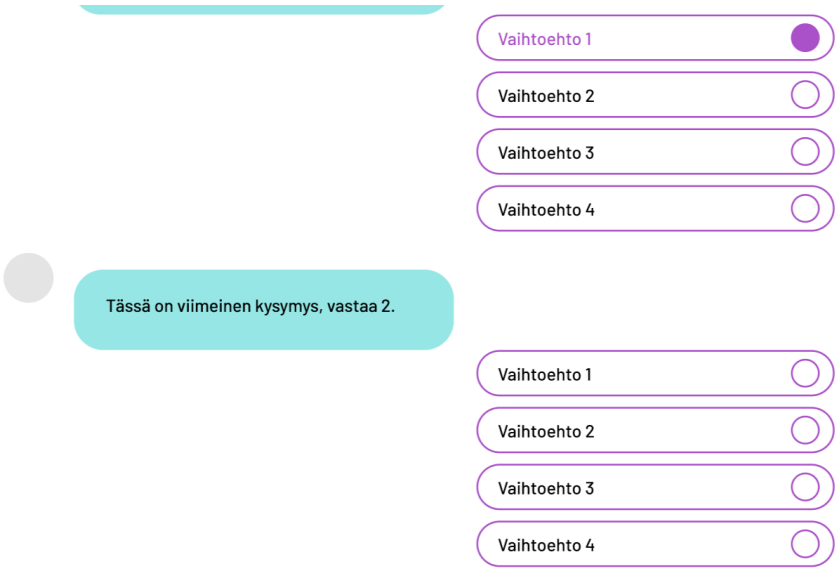
3.1 Front-end, teoriaa ja tekniikoita

Front-endin, eli käyttäjälle näkyvän sovelluskerroksen tehtävänä on ennen kaikkea luoda toimiva ja suoraviivainen käyttäjäkokemus, sekä mahdollistaa tallennetun tiedon katselu ja manipulointi. Web-pohjainen, eli selaimessa toimiva front-end on aina viime kädessä toteutettu selaimen ymmärtämillä teknologioilla, eli HTML, CSS ja Javascript-koodilla. Moderni front-end koodi toteutetaan usein suoraviivaistamisen nimissä erilaisilla framework-kirjastoilla, kuten Angular, Vue tai React. Frameworkin avulla usein automatisoidaan sovelluksen ulkoasun ja toiminnallisuuksien luontia nopeamman kehitystyön nimissä.

Yksi suosittu front-end-kirjasto on Facebookin luoma mutta vuonna 2013 avoimen lähdekoodin projektiksi muutettu React.js kirjasto (RisingStack 2021). Se perustuu modulaarisiin viitekehyksen tarjoamiin valmiisiin ja kehittäjän itse rakentamiin uudelleenkäytettäviin komponentteihin (React 2022a). Komponentit ovat jokaisen React-sovelluksen uudelleenkäytettäviä rakennuspalikoita, jotka palauttavat aina kuvauksen siitä, miltä käyttöliittymän komponenttiin liittyvän osan tulisi näyttää (Kagga 2018). Reactin avulla rakennetaan niin kutsuttuja SPA-sovelluksia (Single Page Application), joissa varsinainen sivu haetaan vain kerran, ja sen sisältö päivitetään hyödyntäen javascriptiä (Mozilla 2022). React on laajalti käytössä ja erittäin suosittu teknologia front-end kehitykseen (StackOverflow 2021).

3.2 Pestikoneen front-end

React valikoitui myös pestikoneen front-endin rakennuksen teknologiaksi ennen kaikkea hyvän tuen, lisättävien ominaisuuksien ja dokumentaation perusteella. Myös nopea kehityssykli, jonka React mahdollistaa, oli tärkeä kriteeri. Organisaatio ei määrittänyt parametreja ulkoasulle, joten digipalvelutiimi sai vapaat kädet toteutukseen. Perustasolla pestikoneen ulkoasu olisi voinut olla normaali lomake, toteutettuna jollain valmiilla kyselypalvelulla. Tässä tapauksessa projektin designer loi konseptin, jota voidaan tässä kontekstissa kutsua termillä ’simuloitu chat-sovellus’ (Kuva 8). Ajatuksena oli siis luoda visuaalisesti kiinnostava tiedonkeräysväline, jolla on tarkoitus kannustaa osallistujia pesteihin ’hakemiseen’. Mielekkään ulkoasun toivottiin kannustavan useampia kohderyhmäläisiä ilmaisemaan mieltymyksensä, ja sitä kautta vähentämään tarvetta sattumanvaraiselle pestien allokoinnille. Esimerkiksi Kajon tapauksessa ne kohderyhmäläiset, jotka eivät vastaisi pestikoneeseen, jaettiin edelleen sattumanvaraisesti pestaajien tarpeiden mukaan.



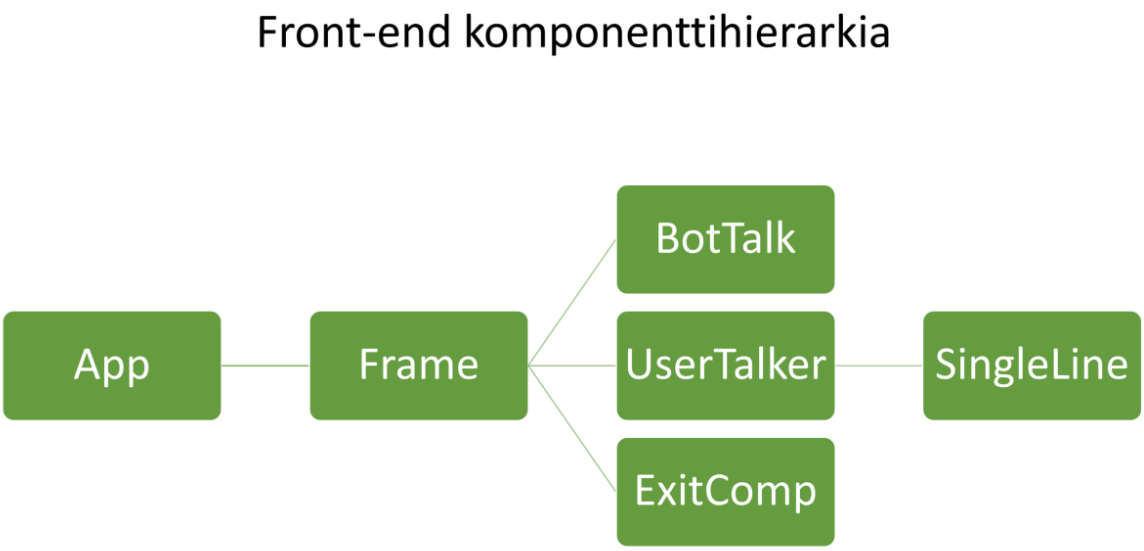
Kuva 8. Pestikoneen design-konsepti

Chatin konseptina toimi kysymys-vastaus-ratkaisu, joka heijasti avainsanojen ’mitä-missä-milloin’ (ks. Liite 1) luokittelua, jossa avainsanat on jaoteltu loogisiin kategorioihin. Digipalveluiden piirissä päätettiin avainsanojen tarkastelun jälkeen rajoittaa eri kategorioiden valintamahdollisuuksia ns. poissulkulogiikan perusteella, perusteena kerättävän tiedon kokonaislaatu. Tietyt avainsanat kategorioiden ’missä’ ja ’milloin’ (Liite 1) sisällä arvioitiin toisensa poissulkeviksi perustuen datan laatuvaatimuksiin. Esimerkiksi mahdollisuus valita ’milloin’-kategorian sisällä molemmat ’Päivätyö’ ja ’Iltahommia’ on mieltymysten ja pestien vertailun kannalta vääristävä mahdollisuus, sillä yksittäinen pesti voi käytännössä olla vain yhtä näiden väliltä, ja saman kategorian sisältä löytyvä ”Aina valmiina-ilman työvuoroja”-asiasanan kattaa sananmukaisesti ne pestit, joiden ”työaika” on vaikeammin määriteltävissä. Nämä poissulkulogiikat rakennettiin sisään front-endin toteutukseen.

Päätös tehtiin myös asiasanojen ’kovakoodauksesta’ suoraan sovellukseen tietokanta-avaimineen. Normaalitilanteessa myös avainsanat saatettaisiin hakea tietokannasta GET-kutsulla sovelluksen käynnistyessä, mutta kehitysprosessin aikaisessa vaiheessa tehtiin päätös viedä asiasanat ja niiden tietokanta-avaimet JSON-muodossa suoraan sovellukseen erilliseen tiedostoon sovelluskehityksen nopeuttamiseksi. Taustalla vaikutti myös kehityksen aloittaminen front-endistä, sekä englanti-ruotsi-käännöksen rakenteen ja viennin suoraviivaistaminen. Mahdollisuus sovelluksen muokkaamiseen tietokannasta haettavilla asiasanoilla on olemassa, mutta sen tarve riippuu front-endin uudelleenkäytön todennäköisyyksistä. Esimerkiksi lomakkeenomaisempi front-end, käytännössä vähemmillä dynaamisilla ominaisuuksilla, mahdollistaisi uudelleenkäytön lähes sellaisenaan, jolloin myös avainsanojen haku tietokannasta olisi ollut perusteltu ratkaisu.

Sovelluksen rakennusvaiheessa hyödynnettiin useita erilaisia valmiita kirjastoja. Perusrakenne generoitiin suoraan CRA-pohjalla (Create React App 2021) erilaisten konfigurointitarpeiden vähentämiseksi ja luotettavuuden takaamiseksi. Käännökset automatisoitiin hyödyntämällä i18next ja react-i18next kirjastoja, joiden avulla sovelluksen tekstisisällön kääntäminen voitiin automatisoida muuttamalla kaikki tekstisisältö viittaukseksi sovelluksen sisältä löytyviin tiedostoihin, joista kirjasto haki kullekin kielelle oman käännöstekstin. Chat-komponenttien siirtymien animointiin tapahtui react-animations paketilla, joka mahdollistaa erilaisten siirtymisanimaatioiden, kuten fade-in ja fade-out, luomisen helposti. Koska animaatiot vaativat toimiakseen myös keyframe-toimintoja, prosessi automatisoitiin hyödyntämällä aphrodite-kirjastoa, jonka avulla myös keskitettiin koko sovelluksen css-tyylit yhteen tiedostoon.

Varsinaista tarkoitukseen soveltuvaa chat-ratkaisua olemassa olevan kirjaston muodossa ei etsinnöistä huolimatta onnistuttu löytämään, joten chat-rakenne, jossa kysymys on vasemmalla ja vastaus oikealla toteutettiin itse hyödyntäen Reactin tarjoamaa mahdollisuutta komponenttien tekemiseen. Kysymyksiä edusti BotTalk-komponentti, kun taas yksittäistä vaihtoehtojen listaa edustaa UserTalker-komponentti (Kuva 7).

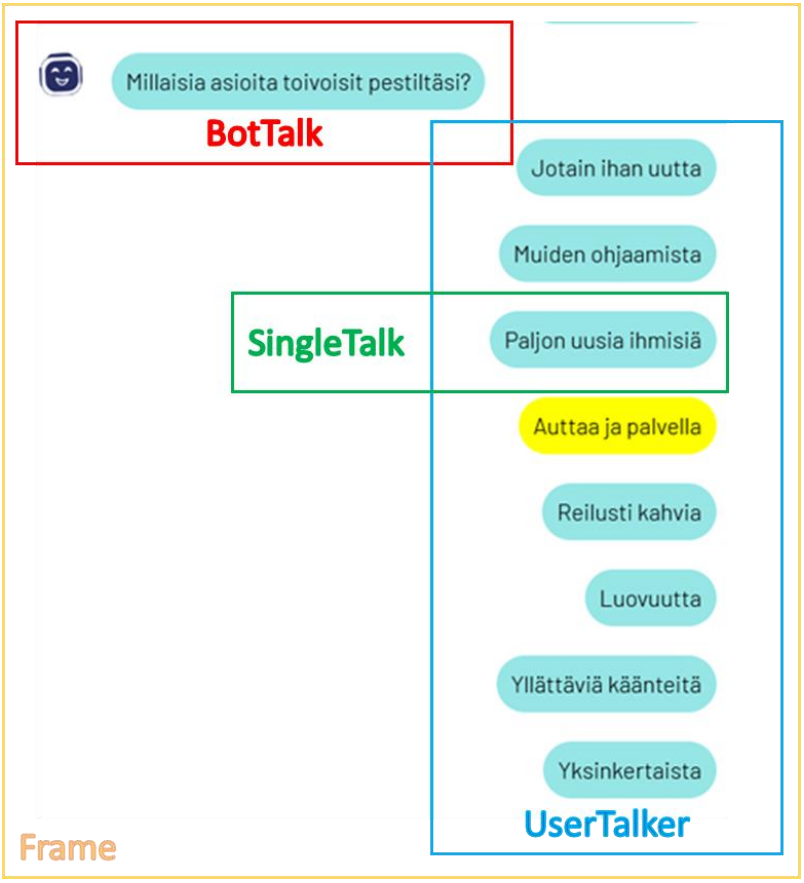


Kuva 7. Kuvaus front-endin komponenttihierarkiasta. Tarkempi selitys taulukossa 1

Taulukko 1. Komponenttien tehtävät

Komponentti	Tarkoitus	Kutsuu komponenttia
Frame	Hallinnoi sovelluksen tilaa ja näyttää alikomponentteja tilan mukaan ehdollistetusti	BotTalk, UserTalker
UserTalker	Ottaa vastaan listan näytettäviä avainsanoja ja kutsuu jokaista listan jäsentä kohtaan SingleTalk komponenttia.	SingleTalk
BotTalk	Ottaa vastaan tiedon siitä, mitä sisäisen listaan jäsentä näytetään	-
SingleTalk	Ottaa vastaan tiedon näytettävästä asiasanasta.	-

Komponenttirakenne pyrittiin pitämään mahdollisimman yksinkertaisena. Sovelluksen tilaa hallinnoitiin Frame-komponentin sisällä, jonka sisällä varsinainen chat juoksee. Visuaalisessa mielessä kokonaisuus on kuvattu kuvassa 8.



Kuva 8. BotTalk-, UserTalker- ja SingleTalk-komponenttien lopulliset ulkonäöt Frame-komponentin sisällä. SingleTalk-komponentti ’Auttaa ja palvellla’ on valittu

Koska Frame-komponentissa hallinnoidaan sovelluksen tilaa, eli sen sisäistä tietoa, se määrittää mitä sen sisällä näytetään kutsumalla erilaisin ehdoin BotTalk- ja UserTalker-komponentteja tietyillä sisällöillä.

Käyttäjälle esitetään kysymys BotTalk-komponentilla, johon vastataan valitsemalla yksi tai useampi UserTalker-komponentin tarjoama SingleTalk-vaihtoehto, joka valittaessa muuttuu keltaiseksi. Valinnat tallennetaan Frame-komponentin Valinta-listaan, jonka perusteella Frame-komponentti kutsuu taas seuraavaa BotTalk- ja UserTalker-komponentteja. Tätä toistetaan, kunnes kaikki valinnat on tehty, jonka jälkeen käyttäjä tarkistaa vastauksensa ja tallentaa vastauksensa painamalla nappia. Tallennuksen onnistuessa käyttäjä saa ilmoituksen, jonka jälkeen käyttäjä sulkee sovelluksen.

Frame-komponentti toteutettiin toimivaksi eri kokoisilla ruuduilla hyödyntäen responsiivistä suunnittelutapaa. Käytännössä sovelluksen css-tyylit on toteutettu siten, että sovellus on käytettävissä ja myös näyttää hyvältä niin kännykän kuin tietokoneen ruudulla (Marcotte 2010). Tietokoneen ruudulta tarkastellessa Frame ei täytä koko ruutua, kun taas kännykän ruudulta tarkastellessa sovellus täyttää koko ruudun, ja fontit ja BotTalk-komponenttiin kuuluva robotin kuva pienenee.

4 Back-end

4.1 Back-end tekniikoita

Web-projektissa back-end, yleisnimitykseltään serveri, palvelee sovelluskokonaisuuden dynaamista logiikkaa ja tietotarpeita. Jos jokainen sovellus käsitetään ennen kaikkea tiedon manipulointivälineenä, back-end tekee suurimman osan työstä front-endin pyyntöjen käsittelijänä. Taustalla ovat ennen kaikkea turvallisuustekijät, sillä kaikki front-endissä oleva koodi on periaatteessa julkisesti tarkasteltavissa, jolloin esimerkiksi suora yhteys front-endin ja tietokannan välillä muodostaisi valtavan tietoturvariskin. Back-end ympäristössä toteutetaan myös monimutkaisempia toimintoja, kuten pääsynhallintaan ja sovelluksen tiedonkäsittelyyn (businesslogiikkakerros) liittyviä ominaisuuksia, jotka saattavat vaatia prosessointitehoja, joita ei selaimessa ole. Yleisiä back-end ohjelmointikieliä ovat mm. Python, Ruby, PHP, Java ja Javascript (Back4App 2022), jota voidaan hyödyntää myös serveriympäristössä Node.js-frameworkin avulla (Clark 2022).

Back-end on oma erillinen sovelluksensa, jonka tarjoamia palveluita käytetään usein erilaisin sovellusrajapintojen (API) avulla. Perustasolla sovellusrajapinta on vain ennalta määritelty valikoima toimintoja erilaisten syötteiden ja tuotosten muodossa (Sandoval 2018). Kärjistettynä, web-sovelluksen back-endissä kyse on usein JSON (JavaScript Object Notation) muotoisen tekstidatan pyytämistä (GET) tai lähettämistä (POST). JSON on tässä tapauksessa vain yksi vaihtoehto koneiden väliseen tiedonsiirtoon, mahdollisia formaatteja on useita (del Alba 2016).

Rajapinta kuuntelee pyyntöjä samanlaisten web-osoitteiden takana, kun mikä tahansa internet-sivu, ja sovellus on käytännössä päällä koko ajan. Koska resurssitarpeita ei voi aina ennakoida, on siis täysin mahdollista, että joinakin aikoina serveri on toimeton, ja toisina taas ylikuormitettu. Resurssitarve yleisellä tasolla on myös hankalasti arvioitava määre ja siihen on useita eri laskutapoja riippuen mitä kokonaisuuden osaa tarkastellaan (Amin 2019).

Perinteisen serverin rinnalle ovat nousseet pilvipalveluntarjoajien kehittämät serverless-ratkaisut, kuten Amazon Web Services:in tarjoama Lambda-palvelu ja Azuren Functions-palvelu, jotka perustuvat tapahtumavetoiseen ajattelutapaan (Microsoft 2022). Pilvipalvelussa ei siis ole erillistä omaa käynnissä olevaa serveriä, vaan koodia, joka laukaistaan perustuen johonkin määrättyyn tapahtumaan. Koodin suoritukseen vaadittavat resurssit ovat koodin käytössä ja laskutettavissa vain sen ajan, kun koodia suoritetaan. Tämä tekee serverless-palveluista erittäin kustannustehokkaita (Nnakwue 2021) verrattuna perinteiseen serveriin, joka ovat päällä koko ajan. Pestikoneeseen ne soveltuivat juuri vähäisen

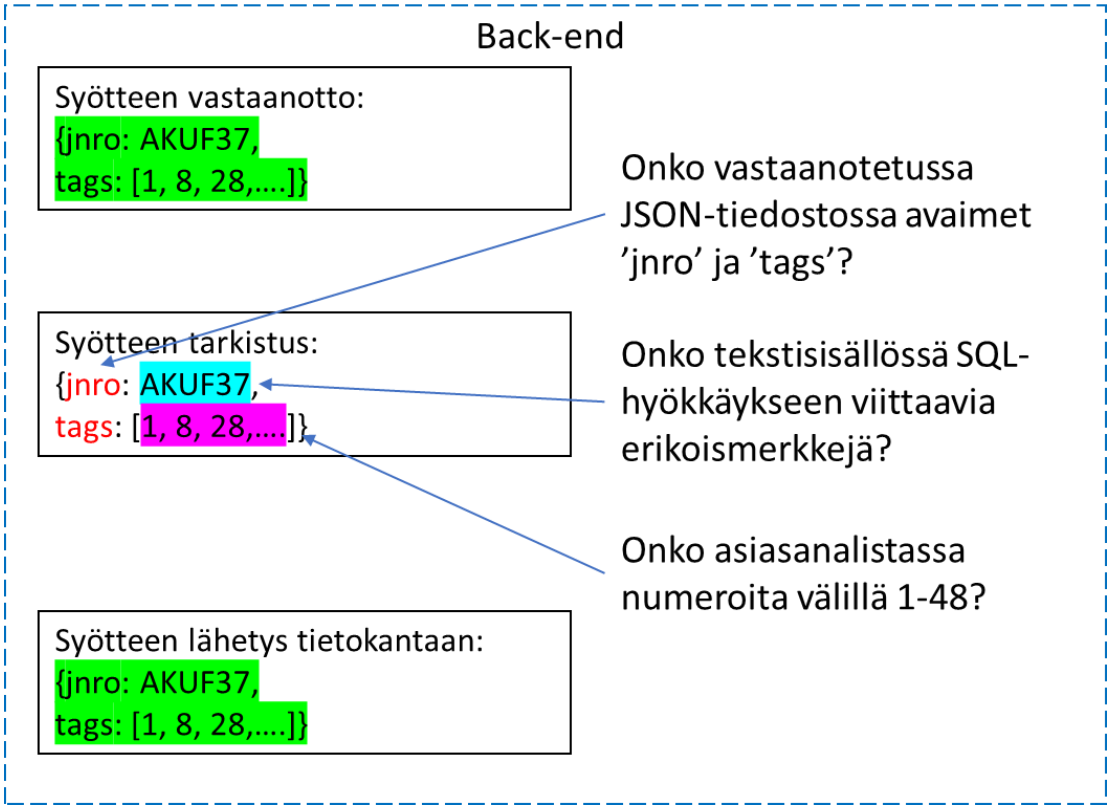
käytön takia. Ainoa ominaisuus, jota Pestikone edellytti, oli kyky ottaa vastaan dataa, jolloin täysimittainen serveri olisi ollut ylimitoitettu ratkaisu.

4.2 Pestikoneen back-end

Pestikoneen tapauksessa back-endin tehtävänä oli syötteen vastaanotto front-endistä, tarkistus ja tietokantayhteyden avaus (Kuva 5). Koska kokonaisuus oli suunniteltu toimimaan JSON-datalla jokaisen kerroksen välissä, back-end ei suorittanut varsinaista tiedonkäsittelyä ollenkaan. Sen päätehtävä olikin syötteen tarkistus, eli varmistus siitä, ettei saapuneessa JSON-datassa ollut vääriä tai sinne kuulumattomia asioita, sekä CORS-perusteella toteutettu matalan tason ’pääsynhallinta’. CORS, eli Cross-Origin Resource Sharing on selaimen tekemä tarkistus siitä, onko juuri tällä sovelluksella oikeudet tehdä pyyntöjä johonkin toisessa osoitteessa sijaitsevaan serveriin (PortSwigger, 2022). Käytännön tasolla tämä tarkoittaa sitä, että serverille voidaan määrittää erikseen osoite, josta se hyväksyy pyyntöjä.

Edellä mainitun toimintojen vähyyden takia back-end toteutettiin serverless-menetelmällä Azuren tarjoaman Functions-palvelun avulla, jossa palvelimelle tallennettava koodi, eli ’funktio’, suoritetaan vain sitä erikseen kutsuttaessa. Tämä suoraviivaisti kehitystyötä myös poistamalla kokonaan tarpeen harkita vaadittavien serveriresurssien määriä, sillä palvelu määrittäisi itse tarvittavat resurssinsa.

Funktio asetettiin toimimaan palvelun HttpTrigger-toiminnolla, eli front-endin lähettämä POST-pyyntö laukaisee automaattisesti funktion suorittamisen. Funktiossa tarkistetaan syöte ja tallennetaan tieto hyödyntäen tietokantaan luotua funktiota, jota back-end kutsuu (Kuva 9).



Kuva 9. Back-endin funktion yleisrakenne

Funktio palauttaa front-endille vastauksena joko 1 tai 0, riippuen syötteen oikeellisuudesta. Tarkistusfunktio tarkistaa ensin, onko POST-pyynnön mukana tulleessa JSON-tiedostossa oikeat avaimet ’jnro’ ja ’tags’, jonka jälkeen ’jnro’-avaimen arvosta puhdistetaan mahdolliset erikoismerkit, kuten *, ” tai / mahdollisten SQL-injektio-hyökkäysten varalta.

SQL-injektio hyökkäyksessä pyritään pääsemään käsiksi tietokannan tietoihin syöttämällä normaalin, esimerkiksi tekstimuotoisen syötteen sijaan SQL-kielistä tekstiä, tavoitteena saada tietoa joko tietokannan sisällöstä tai rakenteesta (Imperva 2022). Syötteiden siistiminen oli tarpeellinen toiminto, sillä nyrkkisääntönä erityisesti web-kontekstin datan käsittelyssä on olla luottamatta ulkoisiin datanlähteisiin sellaisenaan (OWASP 2021). Sama päti myös ’tags’-avaimen arvoon, jonka piti virallisesti olla lista numeroita. Koska myös back-endiin voitiin tuoda tieto samoista tietokanta-avaimista kuin front-endillä oli, tarkoitti se, että lista voitiin tarkistaa myös numeroiltaan. Jos jokin numero listassa ei ollut välillä 1 ja 48, se poistettiin listasta. Tällä vältettiin myös tietokannan sisällä mahdollisesti datan lisäämisvaiheessa tapahtuvat viitevirheet (integrity constraint violation), eli tilanteet, missä tietokantaan olisi lisätty viittaus sellaisen tagin tietokanta-avaimeen, jota tietokannassa ei ole.

5 Tietokanta ja pestialgoritmi

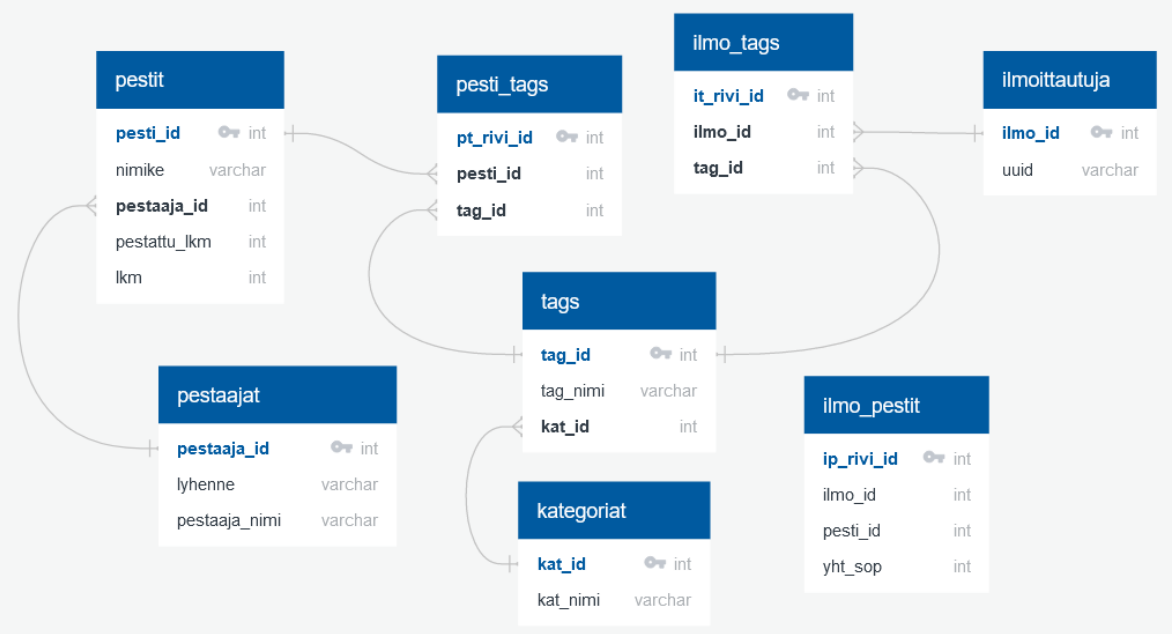
5.1 Pestikoneen tietokannan teoriaa ja käytäntöä

Koska pestikoneen tarkoituksena on nimenomaan tiedon kerääminen tietyltä osallistujajoukolta, perustoimintaedellytyksenä oli kyky tiedon tallennukseen ja analysointiin. Niin perinteisissä kuin moderneissa web-sovelluksissa tiedon tallennus tarkoittaa usein tietokantaa, johon sovellus tekee luku ja kirjoitusoperaatioita. Modernit tietokannat jaetaan yleensä kahteen kategoriaan: perinteiseen rakenteeltaan määriteltyyn relaatiotietokantaan, josta tieto haetaan SQL-kielisillä hauilla, sekä NoSQL-kantoihin, jotka mahdollistava monipuolisen datan tallennuksen samaan paikkaan ilman ennalta määrättyä rakennetta. Tätä strukturoimatonta dataa hyödynnetään usein datajohtoisessa päätöksenteossa, mutta sen analysointi edellyttää kehittyneempiä työkaluja, kuten tekoälyä (Marr 2019), joka kykenee prosessoimaan monissa eri muodoissa. Kappaleessa 2 mainittiin pestikoneen tarkoituksen olevan tehtävien allokointi. Tämä edellytti karkealla tasolla tietoja siitä, mitä allokoidaan ja mihin, mikä taas edellytti tallennus ja analysointikapasiteettia. Kysymys olikin, mikä tietokanta olisi sopiva tarkoitukseen. Tässä tapauksessa front-end lähettää nykyaikaisten web-kehitysperiaatteiden mukaisesti JSON-dataa, jonka voi teoriassa tallentaa sellaisenaan NoSQL-kantaan. Relaatiokannassa JSON tulee purkaa ja viedä SQL-komennoilla erikseen kantaan. Suurin osa moderneista tietokantamoottoreista tukee myös JSON-tietojen purkamista, joten relaatiokantakaan ei ollut mahdoton valinta tallennusperspektiivistä. Jos NoSQL-kannan vahvuus on horisontaalinen skaalautuvuus, niin relaatiokannan vahvuus on datan integriteetti ja varmuus siitä, että kaikki tieto voidaan luotettavasti liittää johonkin muuhun tietoon. Oli tiedossa, että pestikoneen kohderyhmää oli vain murto-osa koko leirin osallistujista, eli vaeltajat ja aikuiset (ikäluokka 18-vuotiaista ylöspäin) eli varsinainen kerätyn tiedon määrä tai luku ja kirjoitusoperaatioiden kuormittavuus ei ollut merkittävän suuri. Käyttäjä kykenee vain tallentamaan tietoa, eikä varsinaista hakutoimintoa ole.

Oli kuitenkin tekijöitä, jotka puolsivat nimenomaan relaationkannan käyttöä. Merkittävä tekijä, jota jouduttiin harkitsemaan, oli kokonaisuuden uudelleenkäytettävyys mahdollisimman vähillä muutoksilla ja edelleen toimivilla analyysirajapinnoilla. Yksi Kajo 2022 tapahtuman pääperiaatteista on ”luoda isoja yleisötapahtumia kestävästi” (Kajo 2021c), mikä periaatteessa tarkoittaa digipalveluiden näkökulmasta mahdollisimman uudelleenkäytettäviä ratkaisuja. Pestikonetta ei välttämättä käytetä sellaisenaan enää koskaan, mutta määritelty kannan rakenne helpottaa uudelleenkäyttöä. Uuden tietokantaserverin pystyttäminen vaatii vain SQL-lauseiden ajamisen ja mahdolliset muutokset voidaan kohdentaa tietokantamoottorin mukaiseen input-funktioon, jonka avulla JSON-data varsinaisesti viedään tietokantaan. Myös tallennettava tieto oli määrämuotoista. Organisaatio oli

määrittänyt aiemmin vaadittavat asiasanat, joiden perusteella pestit luokiteltiin ja joita osallistujat valitsevat. Asiasanoilla oli myös selvästi tunnistettavat kategoriat. Jokaisella pestillä on pestaaja ja jokin organisaation osa, jonka alla pesti on. Relaatiorakenne mahdollisti näiden yksityiskohtien kuvaamisen ilman turhaa toistoa tallennettavassa datassa. Myös vastausduplikaatit, eli monta samaan osallistujaan liittyvää tallennettua vastausta, pystyttiin siivoamaan pois suoraviivaisella SQL-kielisellä vertailulla jo tietokantatasolla. Relaatiokanta myös tarjosi yksiselitteisen totuuden lähteen asiasanojen tunnusluvuille, joita käytettiin sovelluskokonaisuuden kaikissa kerroksissa.

Kun päätös relaatiokannasta oli tehty, seuraava päätös on kannan rakenne. Asiasana-tietokannat relaatiorakenteessa jaetaan yleensä kolmeen eri tyyppiin; yhden taulun denormalisoitu ratkaisu, kahden taulun puolinormalisoitu ratkaisu, sekä kolmen taulun täysinormalisoitu ratkaisu (Keller 2005). Pestikoneen tapauksessa ei ollut käytännön tarvetta varsinaisiin suodatustoimintoihin, joten perusteena on pikemminkin tietokannan rooli siivottuna ja yksiselitteisenä datan lähteenä myös tilastointitarkoituksessa. Pestikoneen tietokanta noudattaa kolmen taulun täysnormalisoitua rakennetta (Kuva 10).



Kuva 10. Pestikoneen tietokantarakenne

Tietokannassa JSON-tiedostossa tuleva ilmoittautujan uniikki tunniste tallennetaan ilmoittautuja-tauluun, jonka luoma ilmo_id yhdistetään JSON-tiedoston tags listan tietoihin taulussa ilmo_tags. Vastaavasti pestaajat, joilla voi olla useampi pesti tarjolla, tunnistetaan ensin taulussa pestaajat, jonka luomiin pestaaja_id-tunnuksiin viitataan taulussa pestit. Pestiin liittyvät tagit tunnistetaan pesti_id:n perusteella taulussa pesti_tags. Pesti_tags ja Ilmo-tags-taulujen tag_id:t viittaavat tags-tauluun, jossa sijaitsevat varsinaiset valittavat tagit, jotka on jaettu kategorioihin viittaamalla kategoriat-tauluun. Ilmo_pestit taulu, joka ei

viittaa mihinkään, toimii analyysin tuloksen tallennuspaikkana, eikä ole välttämätön. Periaatteessa analyysin tulokset voisi tallentaa vaikkapa erilliseen CSV-tiedostoon.

Tietokannan rakenteesta voidaan tehdä kaksi merkittävämpää huomiota, joista ensimmäinen on kannan normalisoinnin taso. Koska kannan rakenne suunniteltiin ennen pestien tietojen keräämistä, ilmeni mahdollisuus vielä lisänormalisointiin. Tässä tapauksessa 'lyhenne'-tietokenttä olisi voitu eriyttää vielä omaksi taulukseen, koska organisaation toimialojen alla on monta pestaajaa. Lisänormalisoinnin hyödyt olivat kuitenkin vähäiset verrattuna sen teettämään käsityöhön tietojen viennin osalta.

Toisena tietokannassa huomiota herättävä tekijä on viittauseheyden sivuuttaminen ilmo_pestit-taulussa. Vaikka kyseessä on relaatiokanta, ilmo_pestit-taulun tietokentät eivät relaatiotasolla viittaa luonnollisiin pääavaimiinsa (esim. ilmo_pestit.ilmo_id >- ilmoittautuja.ilmo_id). Kuten rakenteesta on nähtävissä, jos vastaavat relaatiot olisi toteutettu kannassa, olisi muodostunut ympyräviittaus ilmo_pestit > ilmoittautuja > ilmo_tags > pesti_tags > pestit > ilmo_pestit, joka olisi tehnyt kannasta työlään käyttää. Käytetty ratkaisu on perusteltu relaation päätarkoituksella, eli datan eheydellä. Koska ilmo_pestit-taulu luodaan suoraan pestialgoritmin avulla koneellisesti perustuen pesti_tags- ja ilmo_tags-tauluhin, eheys varmistuu automaattisesti em. taulujen siivoamisella ennen pestialgoritmin ajamista. Toisin sanoen, kyseiseen tauluun ei missään vaiheessa kirjoiteta suoraan dataa muun, kun algoritmin toimesta, joten tilannetta voi verrata ratkaisuihin, missä osa datan eheysvaatimuksista on viety pois tietokantatasolta suoraan businesslogiikkakerrokseen (Wambler 2003). Kyseessä on siis tähän käyttötarkoitukseen soveltuva käytännön ratkaisu, mutta kuten mainittu, tulosten tallennus tietokantaan ei ole välttämätön osa.

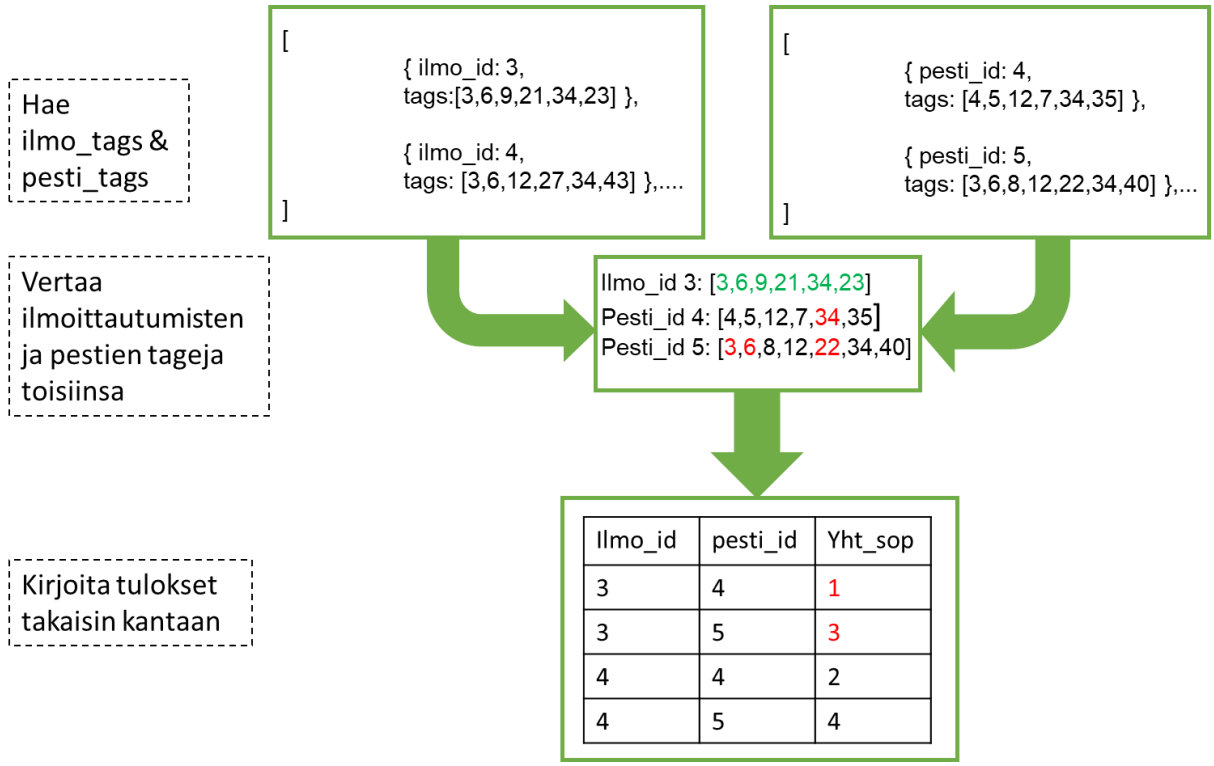
Datan vienti analyysia varten tietokantaan tapahtui kahdella eri tavalla. Osallistujien data vietiin kantaan back-endin toimesta hyödyntämällä tarkoitukseen kirjoitettua insert-funktiota, jolla JSON-data puretaan ilman tarvetta dynaamiselle SQL-lauseelle. Pestien data kerättiin organisaatiolta jaetun Excel-tiedoston avulla, joka siivottiin käsin ja vietiin kantaan Python-skriptillä, jossa ensin tallennettiin kantaan pestaajat, jonka jälkeen jokainen pesti liitettiin johonkin pestaajaan. Lopuksi myös pestin asiasanat tallennettiin kantaan. Prosessia avataan tarkemmin luvussa 6.

Tietokannan tarkoituksen vuoksi se oli myös Euroopan tietosuoja-asetuksen mukainen henkilörekisteri (Tietosuojavaltuutetun toimisto 2021). Kantaan itseensä ei tallenneta todellisia henkilötietoja poislukein pestaajien nimet eikä siinä ole ollenkaan julkista GET-rajapintaa, mutta toiminnan luonteen takia siinä olevat tiedot käsitellään ja yhdistellään ilmoittautumistietoihin jälkeenpäin, tehden siitä henkilörekisterin. Kokonaisuus tulkitaan

kuuluvaksi partion ’Jäsenten ja Huoltajien’ tietosuojaselosteen alle (Suomen Partiolaiset ry 2021) perustuen ’tapahtumien järjestämiseen ja tuottamiseen’. Status henkilörekisterinä asettaa myös vaatimuksia automaattisen päätöksenteon välttämisestä. Tässä tapauksessa pestikoneen tiedonkäsittelyprosessi, jota on tarkennettu osassa 6, sopii hyvin yhteen vaatimuksen kanssa.

5.2 Pestialgoritmi ja sen pohdintoja

Pestikoneen perusajatus on tehtävien asiasanapohjainen allokointi, mikä tässä tapauksessa tarkoitti metodia verrata käyttäjän valitsemia asiasanoja pestien asiasanoihin. Toteutustavaksi valikoitui Python-skripti, jonka perusprosessin kuvaus löytyy kuvasta 11. Algoritmin täysi versio on tarkasteltavissa liitteessä 2. Käytännössä jokaisen ilmoittautujan valintoja verrataan jokaisen pestin asiasanoihin. Jos pestillä ja ilmoittautuneella on samoja asiasanoja, yhteisten asiasanojen määrä lasketaan yhteen, mistä saadaan lopullinen yhteensopivuusluku.



Kuva 11. Pestialgoritmin perusprosessi

Pestialgoritmi on suunniteltu ajettavaksi käytännössä vain kerran, ilmoittautumisen loppuessa. Kehittyneempi pestikone todennäköisesti suorittaisi yhteensopivuusanalyysin heti, ja ehdottaisi sopivaa pestiä käyttäjälle samassa istunnossa, mutta se edellyttäisi huomattavasti älykkäämpiä back-endin business-logiikkaominaisuuksia myös varsinaisen allokoinnin osalta. Ensisijainen syy ominaisuuden poisjättämiseen oli kehitysprosessin aikarajoitteet ja organisaation asettamat vaatimukset. Pestikone oli auki ilmoittautumisen ajan,

joulukuu 2021-helmikuu 2022, mutta ilmoittautumisajan alussa tehty ilmoittautuminen ei saanut vaikuttaa mahdollisuuksiin saada jokin tietty pesti pestikoneesta. Tämä tarkoitti pestialgoritmin tuottamien yhteensopivuuksien arvioimista yhdellä kertaa, eikä useammassa erässä ilmoittautumisen aikana. Käyttäjän istunnossa näkemä mahdollinen pestiehdotus ei täten olisi ollut takuu kyseisen pestin saamisesta, joten väärien odotusten riski oli myös todellinen. Toissijaisena syynä oli organisaation kykenemättömyys pestien ja niihin liittyvien tietojen keräämiseen tarpeeksi ajoissa. Huolimatta useista määräajoista, pesteihin liittyvät tiedot olivat valmiina vasta helmikuussa 2022. Huomioitavaa on myös, että yksittäisen pestin budjetti, eli pestattavien määrä varmistui vasta ilmoittautuneiden määrän ollessa tiedossa.

Pestialgoritmin prosessi tuottaa pelkästään kokonaislukuja yhteensopivuuden määrittämiseksi, mikä oli myös organisaation toive. Potentiaali hienojakoisempaan vertailuun olisi ollut mahdollinen erilaisilla algoritmiin rakennetuilla painotuksilla. Tällä hetkellä jokaisella pestin ja osallistujan yhteisellä asiasanalla on painoarvo 1. Vertailuun olisi ollut mahdollista rakentaa sisään tarkistus, jossa olisi voitu painottaa (esim. painoarvo 1,2) vaikkapa pestin suoritusajankohtaa tai -paikkaa. Toisin sanoen, jos esim. suoritusajankohtaa kuvaava olisi ollut sama osallistujan ja pestin listassa, vertailu olisi kasvattanut yhteensopivuuslukua 0,2 pisteellä.

Pestialgoritmi suunniteltiin kirjoittamaan tulokset takaisin tietokantaan, mikä ei periaatteessa ollut välttämätön askel. Se kuitenkin mahdollisti tietokannan käyttämisen eheänä master-datan lähteenä ja tietojen kokoamisen koko tietokannasta kattavasti ja luotettavasti ilman tarvetta manuaaliseen yhdistelyyn. Se antoi myös reagointivaihtoehtoja kesken kehitysprosessin muuttuneisiin vaatimuksiin. Kehitysprosessin alkaessa ns. pätevyysasiasanoja (ks. Liite 1) ei oltu määritetty ollenkaan mutta pestialgoritmi oli jo valmis. Pestialgoritmi ei vaatinut asian kohdalla muutoksia, vaan ongelma oli pätevyyksien käyttö suodatusperusteena. Esimerkkinä voidaan hyödyntää hygieniapassia. Jos pesti edellyttää hygieniapassia (kuten leirikeittiössä toimiminen tai kahviloiden pestit), ei ole mielekästä ehdottaa sitä sellaiselle osallistujalle, jolla ei ole ko. pätevyyttä. Tämä tarkoitti pätevyyksien perusteella tapahtuvaa suodatusta, jonka olisi voinut periaatteessa implementoida useaan kohtaan prosessia. Koska dataa jouduttaisiin käsittelemään merkittävästi vielä Excelissä, pestikohtaisten pätevyyksien tiedot lisättiin mukaan ja suodatettiin pois Excel-toiminnon avulla.

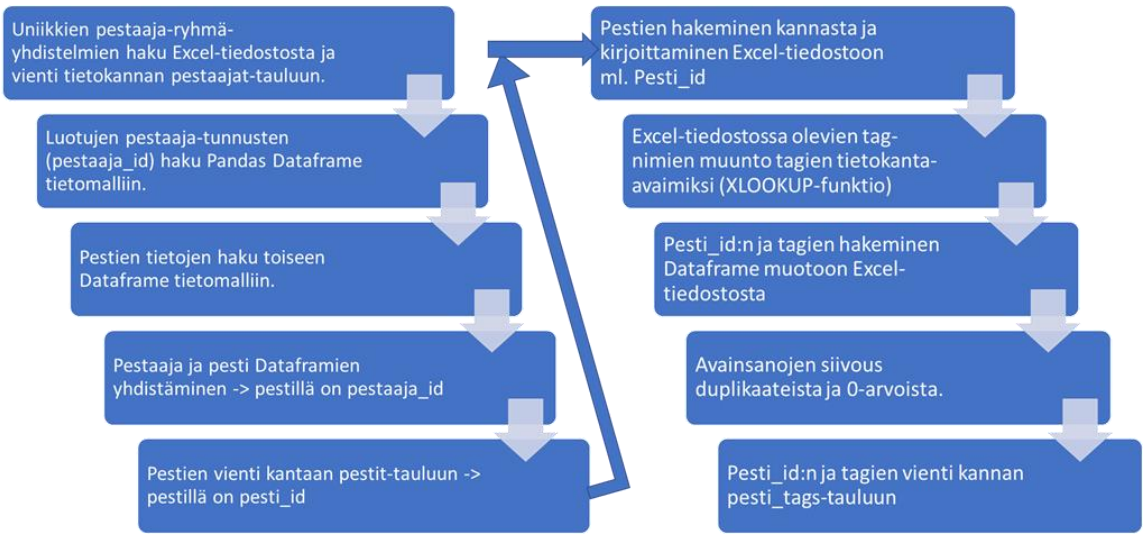
6 Datan käsittely

6.1 Pestidatan käsittely

Merkittävä osa pestikoneen kokonaisuudesta liittyi datan käsittelyyn. Kuten yllä on jo useaan kertaan mainittu, sovelluksen päätarkoitus oli datan kerääminen ja mahdollisimman pitkälle automatisoitu käsittelyprosessi, lopputuloksenaan pestien jakaminen osallistujille. Pestikonekokonaisuus oli jo formalisoinut ilmoittautujien antaman datan helposti käsiteltävään muotoon, mutta varsinainen pestidata edellytti vielä toimenpiteitä.

Päätös pestidatan keräämisestä Excel-muotoisena tehtiin digipalvelutiimin ja HR-tiimin yhteistoimintana. Koska koko organisaatio toimi samassa Microsoft Sharepoint ympäristössä, jaetun Excel-tiedoston luominen oli vaivaton prosessi, joka mahdollisti myös samanaikaisen käytön ja vaivattoman tietojen muokkaamisen jälkeenpäin. Se vaati myös vähiten organisaation erillistä ohjeistusta. Pohjaksi luotiin Excel-tiedosto, jossa kerättiin seuraavat tiedot: ryhmä (eli osa-alue tai muu alatunnus), pestaajan nimi/nimet, pestin nimi, pestattavien lukumäärä sekä avainsanat (sanallisessa muodossa) noudattaen samoja rajoitteita kuin pestikoneessa itsessään. Tiedoston täyttöön luotiin erillinen ohje, joka jaettiin organisaatiolle määräpäivän kanssa. Alkuperäinen tavoite oli pestitietojen koonti ennen pestikoneen avaamista julkiseksi, vaikka varsinaiset käyttäjätoiminnot eivät sitä vaatineet. Jos kyseessä olisi edellisessä luvussa keskusteltu kehittyneempi versio, jossa yhteensopivaa pestiä ehdotettaisiin samassa istunnossa, olisi pestien vieminen tietokantaan ollut pakollinen askel ennen sovelluksen julkaisua. Kuten jo aiemmin on jo mainittu, organisaation muut osat eivät kuitenkaan kyenneet pysymään aikataulussa, joten ratkaisuksi päätynyt, ilmoittautumisen jälkeen tapahtuva yhteensopivuuksien arviointi osoittautui ainoaksi toimivaksi ja ennen kaikkea tasavertaiseksi vaihtoehdoksi.

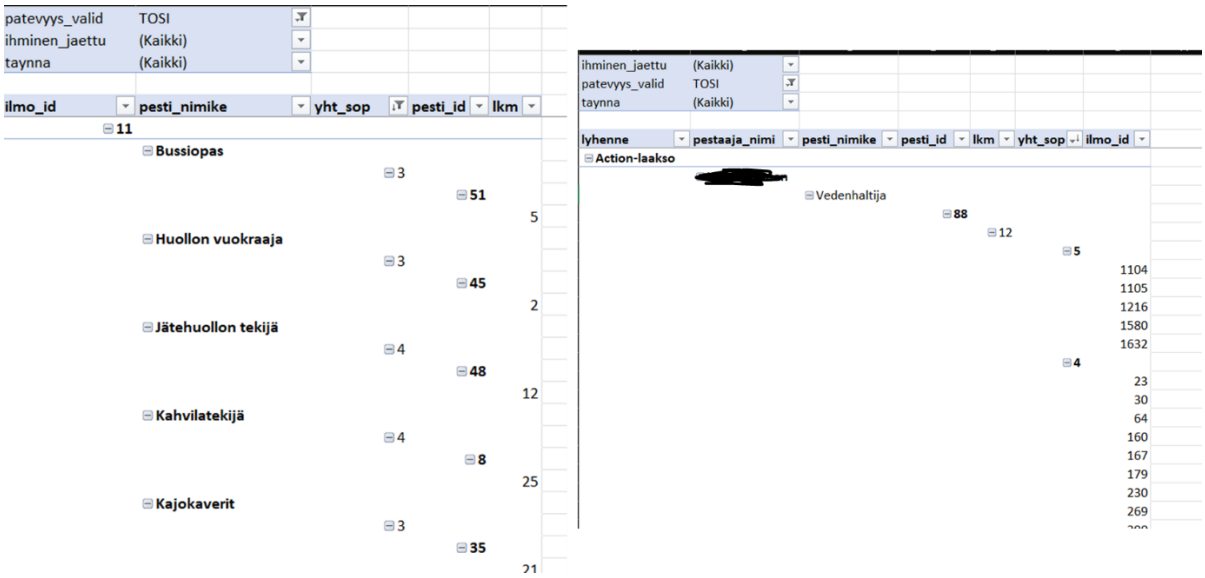
Ennen kuin data voitiin viedä tietokantaan pestialgoritmin hyödynnettäväksi, se vaati käsin tehtävää siivousta. Excel-tiedoston tasolla datan mahdolliset epäsäännöllisyydet ja puutteet korjattiin ensin, jonka jälkeen eriteltiin uniikit pestaaja-ryhmä yhdistelmät. Tässä vaiheessa prosessia suoraviivaistettiin toisella Python-skriptillä (prosessi kuvassa 12).



Kuva 12. Kuvaus pestien tietokantaan viennin vaiheista

Pestien viennin jälkeen ajettiin pestialgoritmi yhteensopivuuksien selvittämiseksi ja tietojen tallentamiseksi takaisin tietokantaan. Tiedot yhdistettiin tietokantatasolla tarkoituksenmukaisella näkymällä, jossa yhdistettiin jokaiseen pestiin jokainen osallistuja yhteensopivuuslukuineen ja pätevyystietoineen. Pätevyysvaatimusten suodattaminen näkymän tasolla suoraan osoittautui hankalaksi puhtaalla SQL komennoilla, joten tiedot päätettiin suodattaa Excel-tasolla. Se myös mahdollisti laajemman tietojen tarkastelun loppukäyttäjän toimesta helpommin verrattuna tilanteeseen, jossa tieto epäsopivista pätevyyksistä olisi jätetty kantatasolla. Toisin sanoen, tietoa voitaisiin käsitellä kokonaisvaltaisemmin mahdollisissa ristiriitatilanteissa pysyen Excel-tasolla, ilman että tulisi tarvetta viitata tietokantaan.

Varsinainen pestien jako tapahtui seuraavaksi. Tietokannan näkymädata vietiin sellaisenaan Exceliin, jossa varsinainen jako tapahtui käsityönä hyödyntämällä kahta datasta muodostettua Pivot-taulukkoa, toinen pestien näkökulmasta ja toinen osallistujien näkökulmasta (Kuva 13). Pestien allokoinnin jälkeen henkilötiedot yhdistettiin samaan tiedostoon ja kokonaisuus luovutettiin HR-tiimin käyttöön, joka toimitti yhteystiedot pestaajille eteenpäin.



Kuva 13. Pestidata pestien ja osallistujien näkökulmasta

6.2 Pestidatan ja jakoprosessin mietintöjä

Ilmoittautumisen päättyessä pestidata siivottiin ja tallennettiin tietokantaan, jonka jälkeen ajettiin varsinainen pestialgoritmi. Tämä jälkeen varsinainen pestien jakoprosessi suoritettiin käsin yhteensä n. 10 tuntia kestäneessä prosessissa. Jo pestidatan siivous osoittautui suhteellisen työlääksi. Avoimen Excel-tiedoston käyttö keräystarkoitukseen osoittautui virhealttiiksi ja vaivalloisesti tarkastettavaksi ratkaisuksi. Datan kerääminen esimerkiksi lomaketyyppisellä ratkaisulla olisi vähentänyt ongelmia merkittävästi, mm. rajaamalla vaihtoehtoja ja tarkemmalla ohjeistuksella, mutta olisi tuonut mukanaan muokkausongelmat. Pestien budjetit määräytyivät lopullisesti vasta ilmoittautumisen päättymisen jälkeen, joten jaettavissa pesteissä olivat taattuja lukumäärien ja itse pestien osalta. Myös mahdollisiin virhesyötteisiin olisi pitänyt tarjota muokkausmahdollisuus, joka olisi ollut käyttäjäystävällinen ja selkeästi ohjeistettu. Toisin sanoen, käytettävyysnäkökulmasta tilanne olisi edellyttänyt kokonaisen käyttöliittymän rakentamista kaikkine tallennus- ja muokkaustoimintoineen. Tähän verrattuna keskitetty ja kerralla suoritettu datan käsin muokkaus ja siivous arvioitiin merkittävästi vähemmän aikaa vieväksi prosessiksi. On kuitenkin todennäköistä, että hyvin ohjeistettu lomakkeen ja Excel-tiedoston yhdistelmä olisi sallinut mahdollisimman virheettömän ensimmäisen tallennuksen, muokkauksen ja normalisoinut datasyötteet.

Myös itse jakoprosessin oli työläs. Se edellytti pesti_id:n lisäämisen käsin ilmo_id:n kohdalle Excel-tiedostossa jokaista jaettavaa pestiä, yhteensä n. 960 kpl, kohden. Vaikka Pivot-taulukot automatisoitiin pitkälle erilaisilla ehtolauseilla, varmistaen sen, että yhden pestin ja osallistujan voi jakaa vain kerran, vei prosessi silti paljon aikaa. Sen aikana huomattiin vielä pestidataan jääneitä virheitä, eli pestaajan käyttämiä vääriä tai olemattomia

asiasanoja, jotka lopulta tarkoittivat osallistujan kannalta matalia yhteensopivuuslukuja ja täten käytännössä sattumanvaraista pestien allokointia.

7 Pohdinta

Pestikoneen kehitysprosessi oli ennen kaikkea mielenkiintoinen oppimiskokemus yhteistyössä tehdystä ohjelmistokehityksestä. Ohjelmistotuotannon näkökulmasta tarkasteltuna produktin yksittäiset palaset, front-end, back-end ja tietokanta eivät olleet kovin monimutkaisia toteuttaa vaan pikemminkin haaste oli niiden sovittamisessa yhteen erilaisten vaatimusten muuttuessa. Erilaisten teknologiavaihtoehtojen kirjo oli myös mielenkiintoinen ja opettavainen kokemus. Esimerkiksi back-endin toteutus serverless-teknologialla ei ollut ollenkaan itsestäänselvyys ja opinkin paljon Azuren eri palveluista ja niiden mahdollisuuksista päätösprosessin edetessä. Opin myös valtavan paljon uutta Pythonin data-analyysityökaluista, vaikka niiden tarve osoittautui hyvin rajalliseksi.

Erityisen paljon sain lisäkokemusta erilaisten tietokantamoottoreiden dokumentaatioiden lukemisesta ja toiminnallisuuksien eroista. Tuotantoon päätynyt PostgreSQL oli kolmas testattu vaihtoehto, ja jokaisella testatulla moottorilla oli erilainen tapa käsitellä sisään tulevaa JSON-dataa. Tähän liittyvä tiedonhankinta oli erityisesti PostgreSQL:n kohdalla työläs kokemus. Opin myös paljon sovellusten julkaisuprosessista pilvipalveluihin, mikä oli minulla täysin uutta ennen produktin tekemistä. Myös organisaation kanssa käydyt keskustelut tietokannan roolista henkilötietorekisterinä olivat opettavainen kokemus ennen kaikkea teknisten yksityiskohtien viestinnästä ei-tekniselle yleisölle.

Kuten varmasti on jo tullut selville, pestikone oli sisällöllisesti ja ajallisesti rajattu kokonaisuus, jota ei välttämättä koskaan enää käytetä uudestaan. Jatkokehitys laaja-alaisemmaksi ja älykkäämmäksi kokonaisuudeksi saattaisi olla mielenkiintoinen haaste erityisesti juuri tiedon analysoinnin näkökulmasta. Siitä olisi myös mahdollisesti hyötyä partio-organisaatiolle laajemmin, sillä vastaavia suuria määriä vapaaehtoisten väliaikaispestejä jaetaan jokaisen suurleirin yhteydessä aina uudestaan. Prosessin vieläkin pidemmälle viety automatisointi säästäisi merkittävästi vapaaehtoisresursseja jokaisen suurleirin organisaatiossa.

Merkittävin asia, minkä olisi tehnyt toisin ajan niin salliessa olisi ollut front-endin toteutus jollain toisella teknologialla. Se olisi ollut mielenkiintoinen oppimiskokemus, mutta olisi toisaalta rajoittanut tiimin toisten jäsenten osallistumismahdollisuuksia merkittävästi. On myös totta, että tietokannan valinnassa olisi voinut käyttää enemmän harkintaa. Huolimatta kerätyn tiedon suhteellisesta soveltuvuudesta relaatiotietokantaan, tallennuksen toteutus noSQL-kannalla olisi ollut täysin varteenotettava vaihtoehto. Se olisi ollut myös kehityksen kannalta nopeampi vaihtoehto, vaikka erilaisten vaihtoehtojen testaaminen olikin erittäin opettavainen kokemus.

Lähteet

Adobe Workfront. 2022. Waterfall Methodology. Luettavissa: https://www.workfront.com/project-management/methodologies/waterfall Luettu: 12.2.2022

Amin, O. 2019. Calculating server capacity and planning for future user growth. Luettavissa: https://devblogs.microsoft.com/premier-developer/calculating-server-capacity-and-planning-for-future-user-growth/ Luettu 19.2.2022.

Back4App. 2022. The Best Ten Backend Languages. Luettavissa: https://blog.back4app.com/best-backend-language/ Luettu 25.4.2022.

Clark, J. 2022. The Best Ten Backend Languages. Luettavissa: https://blog.back4app.com/best-backend-language/ Luettu 3.2.2022.

Create React App. 2021. Create React App. Luettavissa: https://create-react-app.dev/ Luettu: 20.12.2021

del Alba, L. 2016. Data Serialization Comparison: JSON, YAML, BSON, MessagePack. Luettavissa: https://www.sitepoint.com/data-serialization-comparison-json-yaml-bson-messagepack/ Luettu: 19.2.2022.

Eby, K. 2017. The Ultimate Guide to Understanding and Using a System Development Life Cycle. Luettavissa: https://www.smartsheet.com/system-development-life-cycle-guide Luettu 11.11.2021.

Flyaps 30.6.2020. 'Front-end' & 'back-end': What do these terms mean in different applications? Luettavissa: https://flyaps.com/blog/difference-front-end-back-end-development-in-different-applications/. Luettu: 9.10.2021

IBM. 2020. Three-Tier Architecture. Luettavissa: https://www.ibm.com/cloud/learn/three-tier-architecture Luettu: 3.2.2022

IBM. 2022 What is software development. Luettavissa: https://www.ibm.com/topics/software-development Luettu 23.2.2022

Imperva. 2022. SQL (Structured query language) Injection. Luettavissa: https://www.imperva.com/learn/application-security/sql-injection-sqli/ Luettu: 19.2.2022.

Inflectra. 8.1.2022. Introduction to Agile Software Development Methods. Luettavissa: https://www.inflectra.com/ideas/whitepaper/introduction%20to%20agile%20development%20methods.aspx Luettu: 12.1.2022

Kajo, 2021c. Kajon tavoitteet. Luettavissa: https://kajo2022.fi/mika-kajo/kajon-tavoitteet/ Luettu: 11.12.2021

Kagga, J. 2018. Understanding React Components. Luettavissa: https://medium.com/the-andela-way/understanding-react-components-37f841c1f3bb Luettu: 5.3.2022

Keller, P. 2005. Tags: Database schemas. Luettavissa: http://howto.philippkeller.com/2005/04/24/Tags-Database-schemas/ Luettu: 12.12.2021

Marcotte, E. 2010. Responsive Web Design. Luettavissa: https://alistapart.com/article/responsive-web-design/ Luettu 20.3.2022.

Marr, B. 2019. What Is Unstructured Data And Why Is It So Important To Businesses? An Easy Explanation For Anyone. Luettavissa: https://www.forbes.com/sites/bernardmarr/2019/10/16/what-is-unstructured-data-and-why-is-it-so-important-to-businesses-an-easy-explanation-for-anyone/?sh=388c152d15f6 Luettu 1.3.2022

Microsoft. 2022. Azure Functions. Luettavissa: https://azure.microsoft.com/en-us/services/functions/ Luettu 19.2.2022.

Mozilla. 2022. SPA (Single-page application). Luettavissa: https://developer.mozilla.org/en-US/docs/Glossary/SPA Luettu: 27.3.2022

Node.js. 2021. About Node.js. Luettavissa: https://nodejs.org/en/about/ Luettu: 9.10.2021

Nnakwue, A. 2021. Going serverless with Node.js apps. Luettavissa: https://blog.logrocket.com/going-serverless-node-js-apps/ Luettu 19.2.2022.

OWASP. 2021. Input Validation Cheat Sheet. Luettavissa: https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html Luettu: 19.2.2021.

Partio.fi 17.8.2019. Suomen Partiolaisten seuraavan Finnjamboreen johtajat on valittu. Luettavissa: https://www.partio.fi/ajankohtaista/suomen-partiolaisten-seuraavan-finnjamboreen-johtajat-on-valittu/ Luettu: 9.10.2021

PortSwigger. 2022. Cross-origin resource sharing (CORS). Luettavissa: https://portswigger.net/web-security/cors Luettu: 19.2.2022.

React. 2022a. Components and Props. Luettavissa: https://reactjs.org/docs/components-and-props.html Luettu 19.2.2022

React. 2022b. Fragments. Luettavissa: https://reactjs.org/docs/fragments.html Luettu 26.2.2022.

RisingStack. 2021. The History of React.js on a Timeline. Luettavissa: https://blog.risingstack.com/the-history-of-react-js-on-a-timeline/ Luettu: 19.2.2022

Sandoval, K. 2018. Who Invented the API? Luettavissa: https://nordicapis.com/who-invented-the-api/ Luettu 19.2.2022

StackOverflow. 2021. 2021 Developer Survey. Luettavissa: https://insights.stackoverflow.com/survey/2021 Luettu: 19.2.2022

Suomen Partiolaiset ry. 2021. Jäsenten, huoltajien ja vapaaehtoisten tiedot. Luettavissa: https://www.partio.fi/wp-content/uploads/2021/07/Jasenten-ja-huoltajien-tiedot-tietosuoja-seloste.doc.pdf Luettu: 15.12.2021

Techopedia, 2021. ”Full Stack” Luettavissa: https://www.techopedia.com/definition/33963/full-stack. Luettu: 9.10.2021

Techopedia, 2017. ”Tag”. Luettavissa: https://www.techopedia.com/definition/5240/tag-metadata Luettu: 10.3.2022.

Tietosuojavaltuutetun toimisto 2021. Usein kysyttyä EU:n tietosuoja-asetuksesta. Luettavissa: https://tietosuoja.fi/gdpr Luettu: 9.10.2021

Tucker, A., Morelli, R. & de Silva, C. Software Development. Hoboken: CRC Press, 2011.

Wambler, S. 2003. Implementing Referential Integrity and Shared Business Logic in a RDB Luettavissa: http://www.agiledata.org/essays/referentialIntegrity.html#BusinessLogicImplementationOptions Luettu: 14.12.2021

Wanli, N. 2020. Software Development Lifecycle Management. Luettavissa: https://codecoda.com/en/blog/entry/software-development-lifecycle-management. Luettu: 11.11.2021.

Liitteet

Liite 1. Lista avainsanoista ja niiden luokitteluista

Kategoria (määrä)	Avainsanan	
Missä (1)	Näe leirialuetta	Leirin ulkopuolella
	Siisti sisätyö	Varmasti ulkona
	Yhdessä paikassa	
Mitä (2)	Jotain ihan uutta	Selkeitä suunnitelmia
	Muiden ohjaamista	Monimutkaista
	Paljon uusia ihmisiä	Liikuntaa
	Auttaa ja palvella	Vieraita kieliä
	Reilusti kahvia	Tiimityö
	Luovuutta	Soolotekijä
	Yllättäviä käänteitä	Paljon duunia
	Yksinkertaista	Paljon vapaa-aikaa
	Ohjattua ja systemaattista	lapsiystävällinen
	Tee-se-itse	Saatan käyttää kesätyöturvaa
Milloin (1)	Päivätyö	Yötä myöten
	Iltahommia	Aina valmiina - ilman työvuoroja
Osaaminen (2)	Yhteistyö- ja ihmissuhdetaidot	Kansainvälinen osaaminen
	Joustavuus ja aloitekyky	Elämänhallintataidot
	Kehittämisosaaminen	Talousosaaminen
	Viestintätaidot	Johtamistaidot
	Tekniset työelämätaidot	Projektiosaaminen
	Strateginen ajattelu	Kehittämisosaaminen
Pätevyydet (N)*	EA1	C-ajokortti
	EA2	Hygieniapassi
	Järjestyksenvalvojakortti	Työturvallisuuskortti
	B-ajokortti	

(* Pätevyyksiä valitaan tarpeen mukaan, eikä niiden määrää ole rajoitettu.)

Liite 2. Pestialgoritmi

```
import pyodbc
from collections import defaultdict
import csv
import psycopg2

def vertaa(a, b):
    r = len(set(a) & set(b))
    return r

def noutaja(a): #palauttaa annetun lauseen mukaisen datasetin (ks pestikone DB rakenne)
    lista = defaultdict(list)
    conn = psycopg2.connect(database="pestikone_remote", user="postgres", password="normal", host="localhost", port="5432")
    cursor = conn.cursor()
    cursor.execute(a)
    rows = cursor.fetchall()
    for row in rows:
        lista[row[0]].append(row[1])
    conn.close()
    return lista

def vie(arr):
    conn = psycopg2.connect(database="xxxxxxxxx", user="xxxxxx", password="xxxxxxx", host="localhost", port="5432")
    cursor= conn.cursor()
    cursor.executemany("""INSERT INTO ilmo_pestit (ilmo_id, pesti_id, yht_sop) values (%(ilmo_id)s,%(pesti_id)s,%(yht_sop)s)""", arr)
    conn.commit()
    conn.close()


ilmo_sql = 'select ilmo_id, tag_id from ilmo_tags'
pesti_sql = 'select pesti_id, tag_id from pesti_tags'

ilmot = noutaja(ilmo_sql)
pestit = noutaja(pesti_sql)

def printout(c, d):
    #print('ilmo_id', 'pesti_id', 'yhteensop_luku')
    lista = []
    for k, v in d:
        lista.append({"ilmo_id": c, "pesti_id": k, "yht_sop": v})
    vie(lista)

for k, v in ilmot.items():
    mix = defaultdict(list)
    for key, value in pestit.items():
        mix[key] = vertaa(v,value)
    printout(k, mix.items())
```