jamk

Mikropalveluarkkitehtuurin suunnittelu ja toteutus kuvantunnistusrajapinnalle

Ville-Veikko Viertola

Opinnäytetyö, AMK
Toukokuu 2022
Tieto- ja viestintätekniikka
Ohjelmistotekniikka

jamk | Jyväskylän ammattikorkeakoulu
University of Applied Sciences

jamk

Kuvailulehti

Viertola, Ville-Veikko

Mikropalveluarkkitehtuurin suunnittelu ja toteutus kuvantunnistusrajapinnalle

Jyväskylä: Jyväskylän ammattikorkeakoulu. Toukokuu 2022, 58 sivua.

Tietojenkäsittely ja tietoliikenne. Tieto- ja viestintätekniikan tutkinto-ohjelma. Opinnäytetyö AMK

Julkaisun kieli: suomi

Julkaisulupa avoimessa verkossa: kyllä

Tiivistelmä

Opinnäytetyön toimeksiantajana toimi konsultaation ja ohjelmistotuotannon yritys KovaKoodarit Oy. Toimeksiantona oli suunnitella ja toteuttaa verkkopalvelu asiakasyrityksen kuvantunnistusrajapinnan ja statistiikan työkaluksi. Opinnäytetyön tavoitteena oli toteuttaa nykyaikaisia menetelmiä ja teknologioita hyödyntävä verkkopalvelu, joka vastasi asiakkaan palvelulle asettamiin vaatimuksiin.

Palvelun suunnittelu aloitettiin tarkastelemalla palvelulle asetettuja vaatimuksia, joiden perustella arkkitehtuurimalli valittiin. Palvelu päädyttiin toteuttamaan mikropalveluarkkitehtuurin mukaisesti hyödyntäen Docker-konttiteknologiaa. Mikropalveluiden rajapinnat toteutettiin REST-rajapintatyylin mukaisesti Node.js-ajoympäristössä hyödyntäen Express.js-sovelluskehystä. Mikropalveluiden välityspalvelin toteutettiin Nginx-ohjelmistolla, jonka avulla mikropalvelut yhdistettiin käyttöliittymän web-sovellukseen. Verkkopalvelun tietokantoina käytettiin MongoDB- ja Redis-tietokantoja. MongoDBtä hyödynnettiin kuvantunnistusrajapinnasta saatavien tietojen tallennukseen ja Redistä hyödynnettiin käyttäjien sessiohallinnassa.

Palvelun käyttöliittymä toteutettiin SPA-sovelluksena, jossa käytettiin Vue.js-sovelluskehystä ja Bootstrap-Vue-tyylikirjastoa. Vue-ekosysteemiä laajennettiin Vuex-tilahallintakirjastolla sekä Vue Router -lisäosalla, joka mahdollisti SPA-tyylisen Vue-sovelluskehityksen.

Palvelun tuotantoalusta toteutettiin GitLabiin, jonka tarjoamilla jatkuvan integraation toiminnallisuuksilla mikropalveluiden lähdekoodeista luotiin Docker-levykuvat ja tallennettiin konttirekisteriin. Palvelun julkaisussa hyödynnettiin Docker Composea jonka avulla konfiguraation mukaisten mikropalveluiden Docker-levykuvat noudettiin Gitlab-konttirekisteristä ja käynnistettiin palvelinympäristössä.

Opinnäytetyön tuloksena saatiin vaatimusten mukainen mikropalveluarkkitehtuuria mukaileva verkkopalvelu, jonka käyttöliittymällä ratkaistiin asiakkaan tarpeet ja tuotettiin lisäarvoa. Lisäksi toteutukseen valittu arkkitehtuuri mahdollistaa asiakkaalle tarpeen vaatiessa skaalautuvan verkkopalvelun ja teknologioista riippumattoman jatkokehityksen. Asiakas otti palvelun käyttöönsä osana tuotekokonaisuutta.

Avainsanat (asiasanat)

Node.js, Express, Vue, JavaScript, Docker, Mikropalvelut

Muut tiedot (salassa pidettävät liitteet)

jamk

Description

Viertola, Ville-Veikko

Design and implementation of a microservice architecture for an image recognition API

Jyväskylä: JAMK University of Applied Sciences, May 2022, 58 pages.

Engineering and technology. Degree Programme in Information- and Communication Technology. Bachelor´s thesis.

Permission for open access publication: Yes

Language of publication: Finnish

Abstract

The thesis was commissioned by KovaKoodarit Oy, a consulting and software production company. The assignment was to design and implement a web application as a tool for the customers image recognition API and statistics. The goal of the thesis was to implement a web application using modern methods and technologies that would meet the customers' requirements for the application.

The designing process began with an examination of the requirements for the application on which the software architecture was chosen. The web application was implemented in accordance with the microservice architecture, utilizing Docker container technology. The microservice interfaces were implemented according to the REST-style in the Node.js environment utilizing the Express.js framework. The proxy server for the microservices was implemented with Nginx-software, which was used to connect the microservices to the web application. MongoDB and Redis databases were used as databases for the microservices. MongoDB was used to store data from the image recognition API and Redis was utilized in user authentication and session control.

The web application was implemented as a SPA-application using the Vue.js-framework and the BootstrapVue-library. The Vue-ecosystem was expanded with the Vuex state management library and the Vue Router add-on, which enabled SPA-style application development.

The development platform was implemented using GitLab, which provided CI-toolchains for creating Docker-images from the source code and storing them into the container registry. Docker Compose was used for retrieving the Docker-images from the registry and launching them in the production environment.

As a result of the thesis, a web application following microservice guidelines was created. Web application successfully satisfied the customer's needs and provided added value to the product. The architectural decision enables the customer to have a scalable application and technology-independent development. The customer introduced the application as part of their product.

Keywords/tags (subjects)

Node.js, Express, Vue, JavaScript, Docker, Microservices

Miscellaneous (Confidential information)

Sisältö

Kuviot

Lyhenteet

API	Application Programming Interface
CI/CD	Continuous Integration/Continuous Delivery
CLI	Command-line Interface
CRUD	Create Read Update Delete
CSS	Cascading Style Sheets
DOM	Document Object Model
HTML	Hypertext Markup Language
HTTP/S	Hypertext Transfer Protocol / Secure
JSON	JavaScript Object Notation
JWT	Json Web Token
NoSQL	Not only SQL
NPM	Node Package Manager
REST	Representational State Transfer
SAST	Static Application Security Testing
SFC	Single File Component
SPA	Single Page Application
SQL	Structured Query Language
SSH	Secure Shell
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WSL	Windows Subsystem for Linux

1 Johdanto

1.1 Toimeksianto

Opinnäytetyön toimeksiantaja toimi Kovakoodarit Oy, joka on vuonna 2007 perustettu rovaniemeläinen konsultaatio ja ohjelmistotuotannon yritys. Kovakoodarit Oy on erikoistunut verkkosivu, verkkokauppa ja yritysintra kehitykseen (KovaKoodarit n.d). Toimeksiantona oli suunnitella ja toteuttaa verkkopalvelu asiakasyritys Kalasydän Oy:lle, joka kehittää innovatiivista Fishheart-kalatietä. Kalatiellä pyritään parantamaan kalalajien kutuliikehdintää rajoitetuissa virtavesistöissä. Fishheart on kelluva ja liikuteltava hydraulinen kalatiejärjestelmä, jossa hyödynnetään tekoälypohjaista kuvantunnistusta reaaliaikaisessa lajitunnistuksessa. Kehitettävän palvelun on tarkoitus rakentua osaksi Fishheart tuotekokonaisuutta ja toimia työkaluna yrityksen työntekijöille sekä palvella yrityksen asiakkaita portaalina.

1.2 Tavoitteet ja rajaukset

Kalasydän Oy:n ongelmana oli, että heidän käytössään oleva käyttöliittymä tunnistustietojen hallintaan oli monimutkainen ja työläs käyttää. Tämän seurauksena järjestelmän operaattorilta kului paljon aikaa tunnistustietojen läpikäyntiin ja uuden opetusmateriaalin luontiin. Kalasydän Oy:llä ei myöskään ollut tapaa esitellä kalatien tuottamaa statistiikka ja kuvia asiakkailleen.

Opinnäytetyön tavoitteena oli kehittää ja toteuttaa Kalasydän Oy:lle nykyaikaisia sovellustekniikoita hyödyntävä verkkopalvelu, joka antaa näkymän tunnistusjärjestelmään sekä tuo asiakkaille lisäarvoa statistiikan muodossa. Opinnäytetyön tärkein tavoite oli rakentaa työkalukokonaisuus, jonka avulla järjestelmän operaattorit voivat visuaalisesti todentaa kuvakohtaisia tunnistustietoja, korjata mahdollisia tunnistusvirheitä ja tuottaa opetusmateriaalia kuvantunnistusjärjestelmälle. Lisäksi tavoitteena oli toteuttaa helppokäyttöinen asiakasportaali, josta laitekohtainen statistiikka olisi selkeästi näytettävissä asiakkaille.

Käyttöliittymän toteutuksen esittely rajattiin vain opinnäytetyön tavoitteiden kannalta merkittäviin alueisiin ja näkymiin. Opinnäytetyöstä myös rajattiin pois palvelinympäristön shell-skriptit sekä TLS-protokollaan ja käyttäjä autentikointiin käytettyjen sovellusten toteutusten esittelyt.

2 Palvelun vaatimukset

Vaatimukset ovat keskeinen osa ohjelmistokehitystä ja ne kuvaavat asiakkaan vaatimuksia palvelulle. Vaatimukset jakautuvat kahteen kategoriaan, toiminnalliset- ja ei-toiminnalliset vaatimukset. Toiminnalliset vaatimukset kuvaavat mitä asioita palvelulla voidaan tehdä ja ei-toiminnalliset vaatimukset puolestaan kuvaavat palveluun kohdistuvia laadullisia vaatimuksia kuten tietoturva, käytettävyys, skaalautuvuus ja teknologiset vaatimukset. (Luukkainen 2021.)

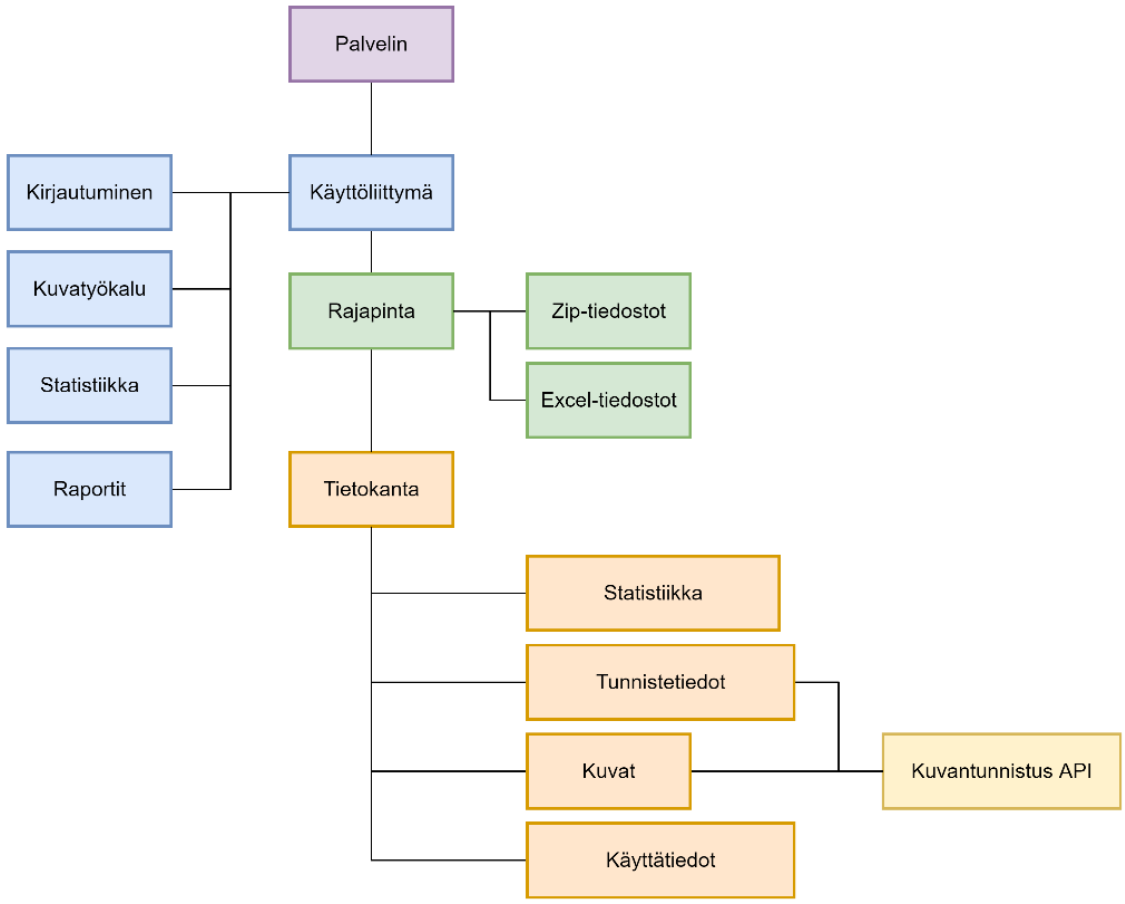
Opinnäytetyön lähtötietoina asiakkaan vaatimukset palvelulle olivat vähäiset ja kuvasivat yleisellä tasolla mitä palvelulla haluttiin tehdä ja mihin ongelmiin haluttiin ratkaisu. Palvelun ainoa toimintaympäristöön liittyvä rajoite oli, että palvelun tulee hyödyntää asiakkaan kuvantunnistusrajapintaa. Tästä syystä toimeksiantaja ei nähnyt erillisen vaatimusmäärittelydokumentin kirjoittamista tarpeelliseksi vaan kehitystyö voitiin aloittaa asiakkaan antamien lähtövaatimusten perusteella.

Vaatimukset perustuivat asiakkaan ja toimeksiantajan kanssa käytyihin keskusteluihin palvelun toiminnallisuuksista. Kehitysprojektin edetessä vaatimuksia tarkennettiin ja uusia vaatimuksia lisättiin työhön. Muutokset vaatimuksiin eivät vaikuttaneet lähtövaatimusten perustella tehtyihin arkkitehtuurisiin- ja teknisiin suunnitelmiin. Luvussa esitellään merkittävät vaatimukset, joiden pohjalta kehitystyö aloitettiin. Palvelun toiminnallisiksi vaatimuksiksi tunnistettiin seuraavia asioita.

- Operaattori voi selata kuvantunnistusrajapinnan kuvia ja tunnisteita visuaalisesti.
- Operaattori voi hylätä huonolaatuiset kuvantunnistusrajapinnan kuvat.
- Operaattori voi hyväksyä yksittäisen kohdetunnistuksen luokat.
- Operaattori voi muokata yksittäisen kohdetunnistuksen luokkia.
- Operaattori voi generoida raportin tunnistusluokista.
- Operaattori voi ladata kuvat tunnistusluokista Zip-tiedoston.
- Operaattori voi ladata raportin tunnistusluokista Excel-tiedoston.
- Käyttäjä voi nähdä laitekohtaisen statistiikan tunnistusluokista.
- Käyttäjä voi nähdä laitekohtaiset parhaat kuvat.
- Käyttäjien tulee rekisteröityä ja kirjautua palveluun.

Asiakkaalla ei ollut projektin alkaessa ei-toiminnallisia vaatimuksia palvelulle, joten ei-toiminnalliset vaatimukset perustuvat toimeksiantajan kanssa käytyihin keskusteluihin kehitettävän palvelun laadullisista seikoista. Tärkeimpinä asioina esille nousi se, että palvelun tulee olla helposti jatkokehitettävissä ja skaalautuva tulevaisuuden tarpeisiin. Vaatimusten perusteella palvelusta tehtiin

alustava miellekartta, johon pyrittiin tunnistamaan kaikki palvelun komponentit huomioiden palvelun luonne (ks. kuvio 1).



Kuvio 1. Palvelun miellekartta

3 Kehitystyön näkökulmia

Verkkopalvelukehitys on laaja aihe ja se pitää sisällään valtavan määrän käsitteitä, työkaluja ja teknologioita (ks. Huffine 2020). Opinnäytetyössä kehitettävä verkkopalvelu toteutetaan ja toimitetaan valmiina palveluna asiakkaalle, joten kehitysprosessia on syytä tarkastella eri työtehtävien ja osaamisalueiden näkökulmista. Eri näkökulmia tarkastelemalla saadaan kokonaiskuva vaadittavista teknologioista ja osaamisesta, joiden avulla verkkopalvelu voidaan viedä tuotantoon.

Sovellustasolla työtehtävät voidaan jakaa kahteen kehitysalueeseen, joista käytetään termejä front-end ja back-end. Palvelun julkaisu- ja tuotantoprosessitasolla kehitystyöstä voidaan käyttää

termiä DevOps. Termit kuvaavat korkealla tasolla työn luonnetta sekä teknologista suuntautumista.

Front-end viittaa käyttäjälle näkyvän osan eli käyttöliittymän kehitystyöhön ja siihen liittyviin teknologioihin. Verkkosovellusten käyttöliittymien perusteknologioihin kuuluvat selaimille tarkoitetut kehityskielet HTML, CSS ja JavaScript (Choudhary 2019). Tietoturvaratkaisut ja JavaScript- ja CSS-kehykset ovat myös tärkeässä roolissa nykypäivän front-end kehityksessä.

Back-end puolestaan viittaa verkkopalvelun koodiin, joka suoritetaan palvelimella. Back-endin tehtävänä on tarjota toiminnallisuuksia käyttöliittymälle rajapintojen avulla. Tyypillisiä toiminnallisuuksia ovat esimerkiksi tietokantaoperaatiot tai algoritmit. Back-end tasolla teknologiat eivät rajoitu tiettyihin ohjelmointikieliin kuten front-end kehityksessä. Tyypillisiä ohjelmointikieliä back-end kehityksessä ovat esimerkiksi PHP, Ruby, Python, C#, Java ja JavaScript. (Filipova & Vilão 2018.)

DevOps puolestaan viittaa tuotantoinfrastruktuuriin sekä kehitys- ja julkaisuympäristöihin liittyvään kehitystyöhön ja hallinnointiin. Tyypillisiä DevOps-tehtäviä ovat esimerkiksi palveluiden monitorointi, testausprosessien automatisointi, jatkuva integraatio (Continuous Integration) sekä jatkuva julkaisu (Continuous Delivery). DevOpsin pohjalla on ajatus infrastruktuurin kehityksestä, joka mahdollistaa sovellusten nopean julkaisu- ja kehityssyklin hyödyntäen ketteriä menetelmiä. Tyypillisiä DevOps-työkaluja ovat esimerkiksi Docker, Kubernetes, Jenkins ja eri pilvipalvelu ja tuotantoalustat kuten AWS, Microsoft Azure, GitLab, GitHub ja Jira. (What is DevOps n.d; DevOps Tools n.d)

Palvelun toteutuksessa hyödynnetään edellä mainittuihin termeihin liittyviä teknologioita, työkaluja ja metodeja. Niiden avulla pyritään luomaan nykyaikaisia teknologioita hyödyntävä verkkopalvelu sekä tuotantoympäristö tukemaan palvelun kehitystyötä ja julkaisua.

4 Arkkitehtuurisuunnitelma

4.1.1 Yleistä

Ohjelmistokehitykseen on tarjolla useita eri arkkitehtuurimalleja, joita voidaan hyödyntää riippuen kehitettävän sovelluksen luonteesta. Arkkitehtuurin valintaan vaikuttavat sovellukselle asetetut ei-toiminnalliset vaatimukset kuten esimerkiksi tekniset, liiketoiminnalliset ja laadulliset vaatimukset. (Wilson, C 2020; Gorton 2011.)

Kansainvälisen tietotekniikanalan järjestön IEEE:n jäsen Gorton (2011) mukaan ohjelmistoarkkitehtuuri kuvaa järjestelmän rakennetta komponenttien ja niiden vuorovaikutuksen perusteella sekä asettaa säännöt sovelluksen toteutukselle. Myös Wilson (2020) nostaa esiin sen, että ohjelmistoarkkitehtuuri kuvaa yleisellä tasolla palvelun rakennetta ja miten pääkomponentit liittyvät toisiinsa. Gorton (2011) toteaa sovelluskehityksen tyypilliseksi ongelmaksi riippuvuuksien kasvamisen suureksi komponenttien välillä, jolloin sovellukseen voi syntyä ei haluttuja sivuvaikutuksia. Suurien järjestelmien arkkitehtuurissa usein hyödynnetään useita eri arkkitehtuurimalleja tai niiden yhdistelmiä, jotta järjestelmän vaatimukset saadaan toteutettua. (Gorton 2011.)

4.1.2 Monoliittinen arkkitehtuurimalli

Monoliittisessa arkkitehtuurimallissa palvelu rakennetaan yhdeksi suoritettavaksi sovellukseksi. Tämä on helppo ja käytännöllinen tapa implementoida yksinkertaisia sovelluksia. Monoliittiset sovellukset skaalautuvat helposti ja sovellusta monistamalla ja lisäämällä kuorman tasaaja (load balancer), voidaan palvelun käyttäjäkapasiteettiä kasvattaa helposti. Toisaalta monoliittiset sovellukset skaalautuvat heikosti, kun data määrät kasvavat. (Richardson, n.d.)

Monoliittisten sovellusten lähdekoodin kasvaminen suureksi voi tuottaa ongelmia kehittäjille ja hidastaa kehitystyötä koska lähdekoodia voi olla vaikea tulkita eikä siinä ole modulaarisuutta. Lisäksi sovelluksen teknologiat ovat valittava heti kehitysvaiheen alussa ja uuden teknologian käyttöönotto tarkoittaa koko sovelluksen uudelleen kirjoittamista. Lisäksi suuren monoliittisen sovelluksen käyttöönotto voi olla haasteellista. (Richardson, n.d.)

4.1.3 Mikropalveluarkkitehtuuri

Mikropalveluarkkitehtuurissa sovelluksen toiminnallisuudet pilkotaan itsenäisiksi osiksi, jotka muodostavat sovelluskokonaisuuden (Microservices 2021). Mikropalveluarkkitehtuuri muuttaa merkittävästi sovelluksen rakennetta verrattuna monoliittiseen arkkitehtuuriin ja tästä syystä sovelluksesta täytyy tunnistaa mikropalveluiksi sopivat toiminnallisuudet. Mikropalveluiden tunnistamisessa täytyy ottaa huomioon, että jokaisella palvelulla tulisi olla vain yksi tehtävä ja mikäli palvelut joutuvat jatkuvasti keskustelemaan toistensa kanssa, on syytä tarkastella mikropalvelu hierarkiaa ja tehtävärajausta. Mikropalveluiden tulisi olla mahdollisimman riippumattomia toisistaan, jotta niiden kehitystyö ja julkaisu voidaan pitää erillään. (Azure DevOps n.d; Microservices 2021.)

Sovelluksen jakaminen mikropalveluihin mahdollistaa helposti ylläpidettävän, testattavan, käyttäjä- ja data määrään skaalautuvan sekä teknologisesti vapaan arkkitehtuurin. REST-rajapintoja hyödyntäen mikropalveluita voidaan kehittää teknologiasta riippumattomasti. julkaista ja hallita ilman että muutokset vaikuttavat muihin mikropalveluihin. Mikropalvelu arkkitehtuuriin liittyy vahvasti konttiteknologiat kuten Docker, jolla mikropalvelut on mahdollista pakata käyttöjärjestelmästä riippumattomiin kontteihin. Mikropalveluilla voidaan tarjota räätälöityjä palvelukokoonpanoja riippuen asiakkaan toiveista ja käyttäjämääristä. (Microservices 2021.)

4.1.4 Serverless-arkkitehtuuri

Serveless-arkkitehtuuri on pilvipalvelualustoihin perustuva ratkaisu, jossa kaikki sovelluksen backend-toiminnallisuudet ovat funktioina palveluntarjoajan pilvialustalla. Serverless-arkkitehtuurista voidaan käyttää termiä FaaS-palvelumalli, joka on lyhenne sanoista Function-as-a-Service. Serverless-arkkitehtuurin etuna on, että sovellukselle ei tarvita omaa palvelinta vaan pilvipalveluntarjoaja ylläpitää tuotantopalvelimet ja hoitaa palvelun mahdollisen skaalauksen. (What is serverless computing? n.d.)

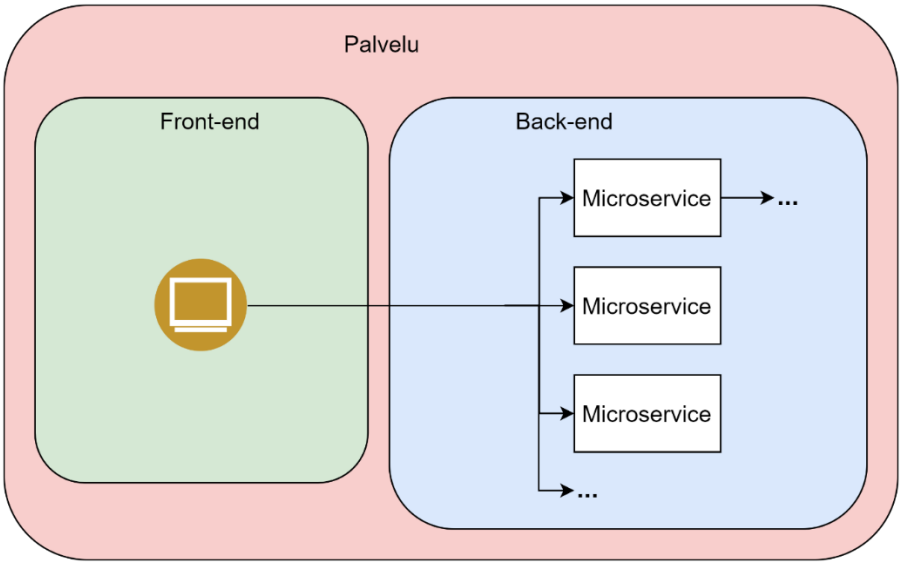
FaaS-palvelumallissa funktiot sidotaan tapahtuma (event) kohtaisesti suoritettaviksi kuten esimerkiksi http-kutsuun tai tietokantaoperaatioon. FaaS-palvelut hinnoitellaan suoritettujen funktioiden perustella ja yksi sen tarkoituksista on pienentää yritysten ja palveluiden palvelinkustannuksia.

4.1.5 Vertailu ja valinta

Arkkitehtuurien vertailussa Serverless-arkkitehtuuri pudotettiin ensimmäisenä pois koska niiden hinnoittelusta ei voitu vetää johtopäätöksiä paljonko lopullinen toteutus olisi maksanut. Kehitettävä palvelu tarvitsee taustaprosesseja, jotka mahdollisesti käyttäisivät suuren määrän FaaS-palvelun funktioita ja näin olleen nostaisivat hintaa. Lisäksi toimeksiantajalla oli käytössään virtuaalipalvelin, jota voitiin hyödyntää tuotantoympäristönä, joten Serverless-arkkitehtuuri olisi luonut ylimääräisiä kuluja.

Mikropalvelu- ja monoliittisen arkkitehtuurin vertailussa mikropalvelut nousivat selvästi edelle. Vaikka opinnäytetyössä kehitettävä palvelu oli kokoluokaltaan pieni ja se olisi ollut hyvin mahdollista toteuttaa monoliittisena arkkitehtuurina, mikropalveluiden teknologinen vapaus nousi ratkaisevaksi tekijäksi.

Palvelun arkkitehtuuriksi valittiin mikropalveluarkkitehtuuri, koska se vastasi parhaiten ei-toiminnallisiin vaatimuksiin, jotka palvelulle oli määritetty. Lisäksi palvelinympäristön kustannukset olivat kiinteät ja helposti perusteltavissa ja kehittäjän näkökulmasta valittu arkkitehtuuri selkeytti projektihierarkiaa. Palvelun skaalaus ei ollut toteutus hetkellä ajankohtainen mutta sen huomioiminen arkkitehtuuri valinnassa oli varautumista tulevaisuuden tarpeisiin. (Ks. kuvio 2.)



Kuvio 2. Mikropalveluarkkitehtuuri

5 Teknologiat ja työkalut

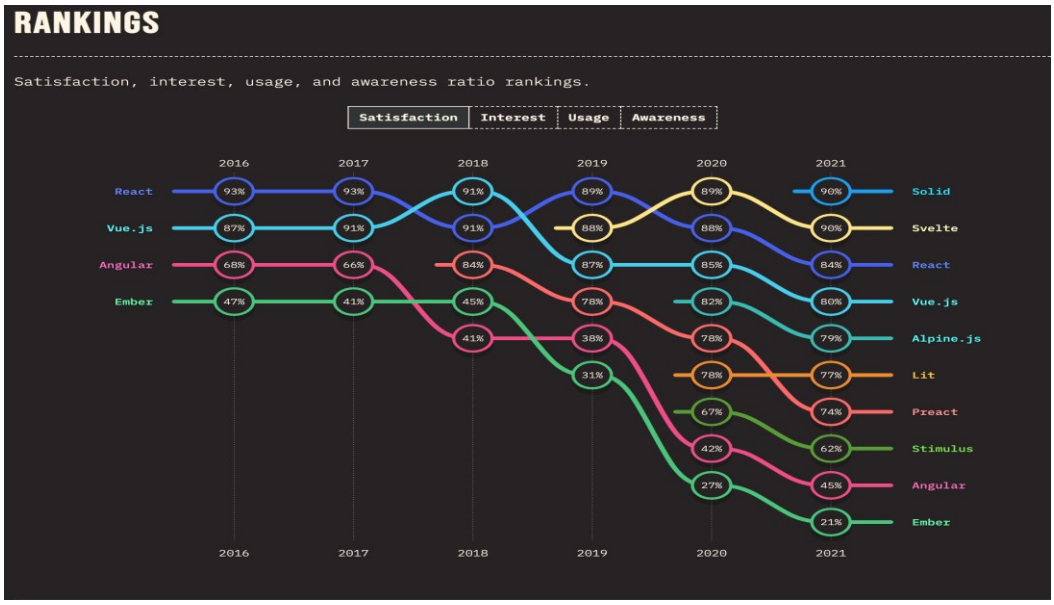
5.1 Front-end

5.1.1 Yleistä

Kuten luvussa 3 mainittiin, verkkosovellusten käyttöliittymien perusteknologioihin kuuluvat selaimille tarkoitetut kehityskielet. HTML-merkintäkielellä ilmaistaan verkkosivun rakennetta ja sisältöä. CSS on tyylisivukieli, jolla voidaan määrittää HTML-elementtien tyylejä ja asettelua. JavaScript on selainsovelluksille tarkoitettu ohjelmointikieli, jonka avulla voidaan luoda verkkosivuille toiminnallisuuksia ja interaktiivisuutta. Lisäksi nykypäivän verkkokehityksessä hyödynnetään JavaScript-sovelluskehyksiä, jotka pyrkivät säästämään aikaa tarjoamalla mahdollisuuden yhdistää HTML-, CSS- ja JavaScript-syntaksia samaan kooditiedostoon. (Choudhary 2019.)

5.1.2 JavaScript-kehykset

Verkkosovellusten käyttöliittymäkehitykseen on tarjolla useita JavaScript ja CSS-sovelluskehyksiä, jotka pyrkivät nopeuttamaan ja yhtenäistämään verkkosivujen kehitysprosessia sekä parantamaan laatua. Vuonna 2021 suoritetun State of JavaScript Surveyn -kyselyn tuloksesta voidaan nähdä, että JavaScript-sovelluskehyksiä on useita ja niiden suosio kehittäjien keskuudessa vaihtelee hyvinkin voimakkaasti. (Ks. kuvio 3.)



Kuvio 3. JavaScript Survey -tyytyväisyyskysely (StateOfJS 2021)

JavaScript-kehykset tyypillisesti tarjoavat omia syntakseja tai syntaksirakenteita yhdistämällä HTML-, CSS- ja JavaScript-kieliä samaan tiedostoon tai jopa samalle koodiriville. CSS-kehykset puolestaan tarjoavat valmiita tyyliluokkia HTML-elementeille. Monet JavaScript-sovelluskehyksistä noudattavat komponenttimallia, jossa käyttöliittymän sisältö koostuu uudelleen käytettävistä komponenteista tai komponenttikokoelmista, joille voidaan määrittää yksilölliset tyylit ja toiminnallisuudet.

5.1.3 Vue.js

Käytettäväksi JavaScript-sovelluskehykseksi valittiin Vue.js aikaisemman Vue-ekosysteemi kokemuksen perusteella. Vue.js on progressiivinen JavaScript-kehys, joka mahdollistaa deklaratiivisen ja komponenttimallisen verkkosovelluskehityksen. Vue käyttää SFC (Single-File Component) -formaattia, joka mahdollistaa JavaScript-, HTML- ja CSS-syntaksin kirjoittamisen saman tiedoston sisälle (ks. kuvio 4). (Vue.js n.d.)

```
//HTML//
<template >
</template>


//JAVASCRIPT//
<script>
export default {
  name: 'Component name',
}
</script>


//CSS//
<style>
</style>
```

Kuvio 4. Vue.js komponenttimalli

Vuen progressiivisuudella tarkoitetaan sitä, että Vue-sovellusta voidaan laajentaa ja mukauttaa lisäominaisuuksilla tarpeiden mukaan. Lisäominaisuuksia voivat olla esimerkiksi staattisen sivuston generointi, palvelinpuolen renderöinti ja tilahallinta (Vue.js n.d).

Vue hyödyntää virtuaalista DOM (Document Object Model) -mallia komponenttien ja tilojen päivitykseen. Vuen virtuaalinen DOM sisältää puurakenteen Vue-komponenteista ja se luodaan selaimen HTML DOM:in rinnalle, jotta Vue sovellus pystyy hallitsemaan tehokkaasti verkkosivun uudelleen renderöintiä tilojen muuttuessa. (Lotanna 2020.)

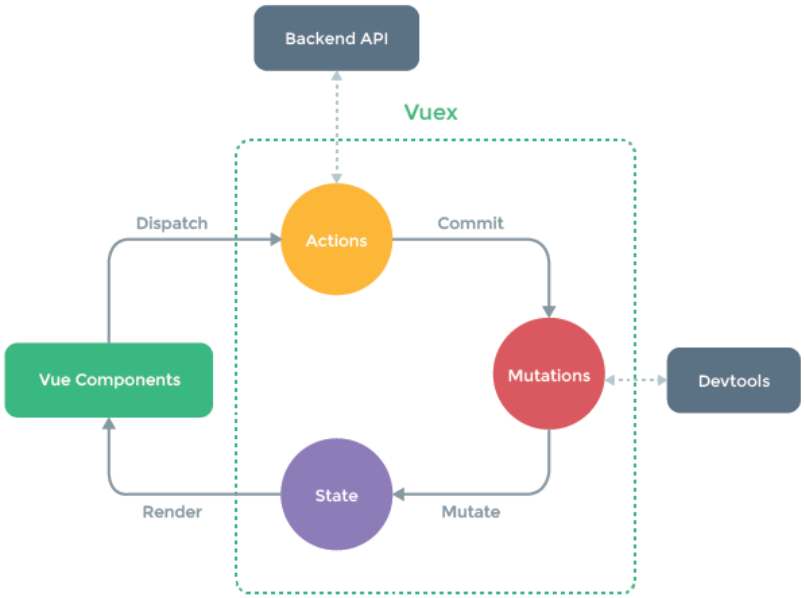
Vue Router

Vue ympäristöä laajennettiin Vue Router -lisäosalla, joka mahdollistaa SPA (Single Page Application) -sovellusten toteuttamisen. SPA-sovelluksessa on vain yksi HTML-sivu, jonka käyttäjä lataa selaimeen ja sivuston sisältö muuttuu dynaamisesti eri näkymiä valittaessa. Vue Router mahdollistaa näkymien navigoinnin, sivuhistorian ja tarjoaa erilaisia uudelleen ohjauksia. Lisäksi Vue Router tarjoaa näkymä vahteja, joilla voidaan rajoittaa käyttäjien pääsyä tiettyihin näkymiin. (Vue Router n.d.)

Vuex

Vue ympäristöä laajennettiin myös Vuex-tilahallintakirjastolla, jonka avulla komponentit voivat keskitetysti jakaa tiloja. Ongelma komponenttipohjaisessa verkkosovelluksessa on se, että tiedon välittäminen komponenttien välillä voi olla haasteellista, mikäli kyseessä on kompleksi komponenttirakenne. Tilannetta voidaan kuvata puuna, jonka lehdet ovat komponentteja ja eri oksilla olevat lehdet ovat riippuvaisia samasta tiedosta. Vuex ratkaisee tämän ongelman eristämällä tiedot yhteen paikkaan, josta komponentit voivan hakea ja muokata tietoja. (What is Vuex? 2022.)

Kuten edellä mainittiin Vuex on erityisen hyödyllinen, kun useampi Vue-sovelluksen komponentti on riippuvainen samasta muuttujasta tai tietorakenteesta kuten esimerkiksi jokin sovelluksen asetus. Vuex tarjoaa dispatch-commit-mutate hallintamallin käytettäväksi Vue-sovelluksen tilahallinnassa ja renderöinnissä (ks. kuvio 5).



Kuvio 5. Vuex tilahallinta (Vuex 2022)

BootstrapVue

CSS-kehykseksi toteutukseen valittiin suosittuun Bootstrap CSS-kehykseen pohjautuva Bootstrap-Vue. BootstrapVue tarjoaa käytettäväksi suuren kokoelman valmiiksi tyyliteltyjä ja interaktiivisia Vue-komponentteja. BootstrapVue-komponentit ovat responsiivisia, joten ne skaalautuvat hyvin näyttölaitteen mukaan ja ovat parametreilla monipuolisesti muokattavissa (ks. kuvio 6.). (BootstrapVue n.d.)


```
<div>
  <b-button>Button</b-button>
  <b-button variant="danger">Button</b-button>
  <b-button variant="success">Button</b-button>
  <b-button variant="outline-primary">Button</b-button>
</div>
```

Kuvio 6. BootstrapVue painike (BootstrapVue n.d)

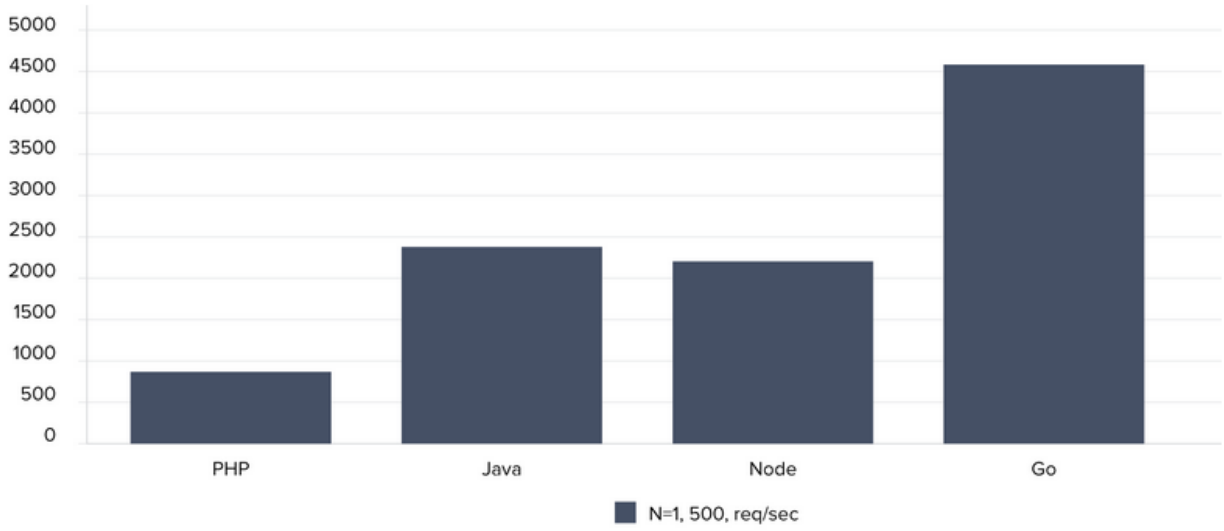
5.2 Back-end

5.2.1 Node.js

Node.js on avoimen lähdekoodin asynkroninen JavaScript-ajoympäristö, joka mahdollistaa selaimille tarkoitetun JavaScript-ohjelmointikielen suorittamisen selaimen ulkopuolella alustasta riippumatta. Node.js käyttää Googlen Chrome-selaimen V8 JavaScript-kääntäjää, jolla JavaScript-ohjelmien suorittaminen selaimen ulkopuolella on mahdollista. Lisäksi Node.js-ympäristöön kuuluu NPM (Node Package Manager) -kirjastohallintatyökalu, jonka avulla kehittäjät voivat asentaa avoimen lähdekoodin lisäkirjastoja. (Introduction to Node.js n.d.)

Node.js on V8-kääntäjän ansiosta suorituskykyinen ja sitä voidaan hyödyntää verkkosovelluksissa monilla alueilla kuten esimerkiksi suoratoistosovelluksissa, suuren tietomäärän reaaliaikaisissa sovelluksissa ja API-sovelluksissa. Node.js käyttää yhtä säiettä (thread) sovelluksen suoritukseen ja hyödyntää tapahtumasilmukkaa (event loop), jonka ansiosta Node.js-sovellukset voivat käsitellä suuria määriä pyyntöjä. (What is Node.js? n.d.)

Vaikka Nodea voidaan pitää erittäin suorituskykyisenä, on syytä nostaa esille vertailu muihin ohjelmointikieliin. Kun Noden suorituskykyä verrataan Go-ohjelmointikieleen, nähdään että Go:n suorituskyky on aivan omaa luokkaansa ja se pystyy käsittelemään noin kaksi kertaa enemmän pyyntöjä kuin Node. Vastaavasti Java ja Node ovat hyvin lähellä toisiaan suorituskyvyllisesti ja puolestaan päihittävän PHP:n yli kaksi kertaisesti. (Ks. kuvio 7.)



Kuvio 7. Node.js suorityskyky vertailu (Peabody n.d)

Node.js valittiin back-end ohjelmointikieleksi koska se tarjoaa riittävän suorituskykyisen ajoympäristön verkkosovelluksille ja sen avulla JavaScript-ohjelmointikieltä voitiin hyödyntää myös back-end kehityksessä, joka oli merkittävä etu aikataulullisesti aikaisemman JavaScript kokemuksen perusteella.

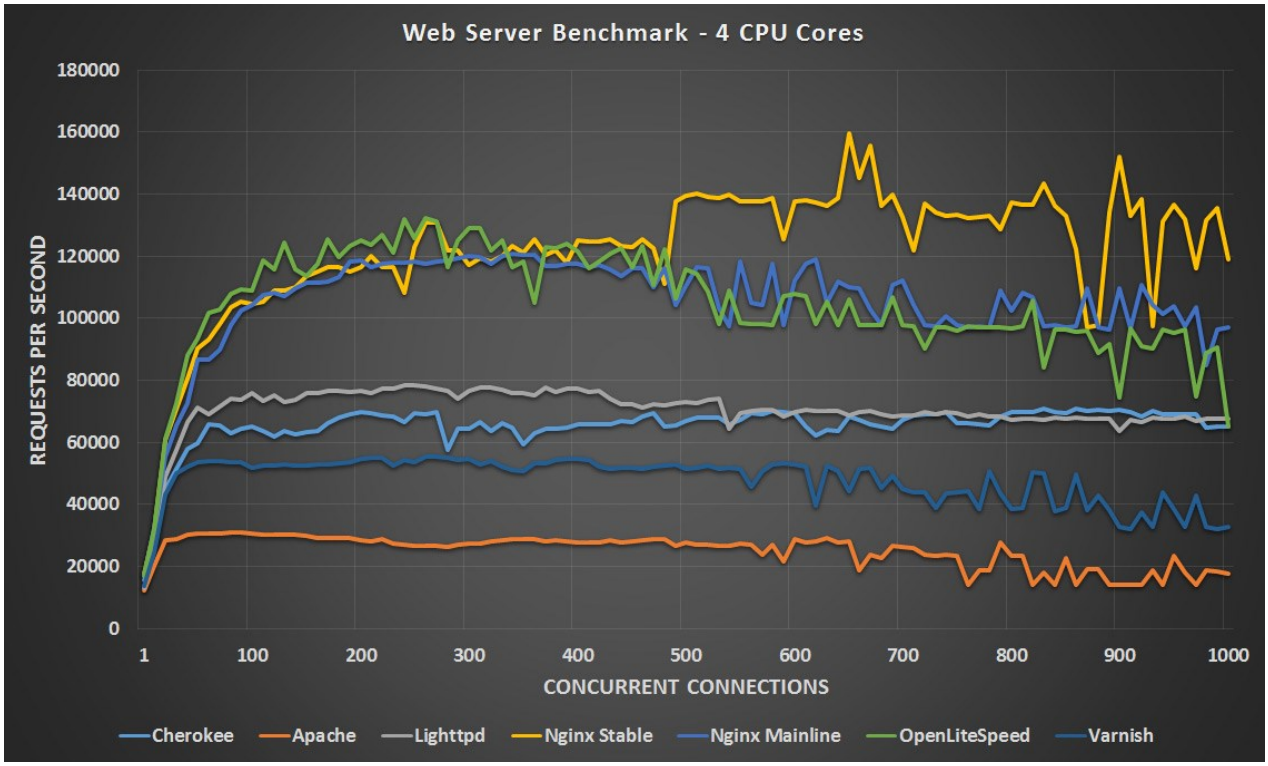
5.2.2 Express.js

Express.js on Node-verkkokehys, joka on tarkoitettu verkkosovellusten ja rajapintojen toteuttamiseen. Se laajentaa Node ympäristöä ja mahdollistaa URL-polkujen reitityksen, HTTP-metodi käsittelijöiden luomisen ja väliohjelmistojen (middleware) lisäämisen HTTP-käsittelijöihin. Vaikka Express tarjoaa toiminnallisuuksia verkkosovelluskehitykseen, on se silti minimalistinen verkkokehys ja useimmat tarpeelliset kirjastot on ladattava NPM-pakettihallinnasta kuten esimerkiksi cookie-parser ja tietokantakirjastot. (Express/Node introduction 2022.)

Express.js valittiin verkkokehykseksi niin ikään aikaisemman kokemuksen perusteella. Se tarjoaa yksinkertaisen ja räätälöitävän sovelluskehyksen, joka sopii hyvin käytettäväksi mikropalveluarkkitehtuurissa. Työssä Express.js-kehystä hyödynnetään mikropalveluiden rajapintojen rakentamiseen.

5.2.3 Nginx

Nginx on vuonna 2004 julkaistu avoimen lähdekoodin ohjelmistoprojekti, joka on tarkoitettu verkko- ja välityspalvelimeksi sekä kuorman tasaajaksi. Nginx on yksi suorituskykyisimmistä verkkopalvelinohjelmistoista ja se on ollut yksi projektin keskeisin tavoite alusta alkaen (ks. kuvio 8). Nginx tukee monia verkkotekniikoita kuten WebSocket, HTTP/2, gRPC sekä useiden videoformaattien striimausta. (NGINX n.d.)



Kuvio 8. Verkkopalvelin suorituskyky vertailu (Jarrod 2016)

Nginx valittiin toteutukseen sen suorituskyvyn ja avoimen lähdekoodin perustella. Nginx:llä on erityinen tehtävä toteutettavassa palvelussa, sitä käytetään välityspalvelimena jakamaan käyttöliittymästä tulevia http-pyyntöjä eri mikropalveluille.

5.2.4 REST-rajapintatyyli

REST (Representational State Transfer) on rajapintakehitystä varten tarkoitettu tyyli, jonka Roy Fieldingin esitteli vuonna 2000. Ajatus REST-rajapintatyylin takana oli, että se parantaa hajautetun

sovelluksen suorituskykyä, skaalausta ja tarjoaa teknologia vapaan kehityksen. REST tukee URI (Uniform Resource Identifier) -järjestelmää käyttäviä protokollia. (Doglio 2018.)

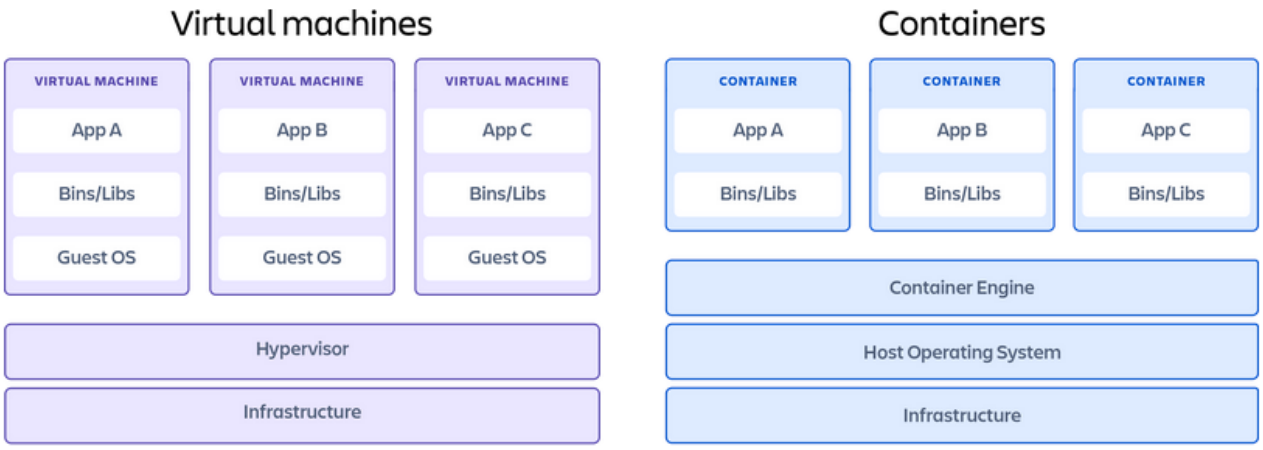
REST-rajapinnat ovat tilattomia, joka tarkoittaa, että asiakkaan ja rajapinnan ei tarvitse tietää toisistaan entuudestaan mitään vaan ne pystyvät käsittelemään toistensa pyyntöjä ja vastauksia ns. tilattomasti. REST-rajapinnoissa käytetään tyypillisesti HTTP/S-protokollaa ja hyödynnetään neljää HTTP-metodia, jotka määrittävät rajapinnan toimintojen luonteen. GET-metodilla rajapinta palauttaa tietoja, POST-metodilla rajapinta tallentaa uuden tiedon, PUT-metodilla rajapinta päivittää tietoja ja DELETE-metodilla tietoja voidaan poistaa. REST-rajapinnoissa HTTP-metodit voidaan linkittää suoraan tietokantojen CRUD (Create Read Update Delete) -toimintoihin. (What is REST? n.d.)

Asiakkaan lähettämissä pyynnöissä tulee määritellä HTTP-metodi, resurssipolku ja vaihtoehtoiset header ja body-tietokentät, joilla asiakas voi sisällyttää tietoa pyyntöön. REST-rajapinnan vastauksessa hyödynnetään HTTP-vastauskoodeja ja JSON- tai XML-formaatteja tiedon palauttamisessa. Mikäli REST-rajapinta palauttaa dataa silloin header-tietokentässä lähetetään *Content-Type* otsikko, jotta asiakas tietää mikä tietotyyppi on kyseessä kuten esimerkiksi ”Content-Type: text/html”. REST-rajapintatyyliä hyödynnetään Express.js:llä toteutettavissa mikropalveluiden rajapinnoissa. (What is REST? n.d.)

5.3 Kontitus

5.3.1 Yleistä

Kontitus (containerization) tarkoittaa ohjelmakoodin paketointia siten että se sisältää vain vaadittavat käyttöjärjestelmän kirjastot ja riippuvuudet ohjelman suorittamiseen. Kontitus on tärkeässä roolissa toteutettavassa palvelussa koska sen avulla mikropalveluarkkitehtuuri saadaan helposti skaalautuvaksi ja tuotantoympäristön pystyttäminen on nopeaa. Kontituksessa ei paketoida koko käyttöjärjestelmää vaan hyödynnetään konttimoottorisovellusta, joka jakaa konteille Linux-käyttöjärjestelmäytimestä tarvittavat riippuvuudet (ks. kuvio 9.). Tämä johtaa siihen, että kontit käyttävät huomattavasti vähemmän tietokoneen resursseja kuin virtuaalikoneet, jotka ovat kopioita koko käyttöjärjestelmästä. Pienen resurssijalanjäljen ansiosta kontteja voidaan suorittaa yhdellä palvelimella huomattavasti enemmän kuin virtuaalikoneita. (Containerization 2021.)



Kuvio 9. Virtual machines vs. containers (Buchanan n.d)

Konttiteknologiat ovat monella tapaa mullistaneet nykyaikaisen ohjelmistotuotannon sillä niiden ansiosta isoja palvelukokonaisuuksia voidaan hallita tehokkaasti. Kontit mahdollistavat mikropalveluarkkitehtuurien toteuttamisen helposti ja palvelinympäristö voidaan pitää yksinkertaisena. Kontit ovat myös turvallisia sillä niiden sisällä suoritettava ohjelmakoodi eristetään isäntäkoneesta ja lisäksi konttien laiterajapintaoikeuksia voidaan tarvittaessa rajoittaa. (Containerization 2021.)

5.3.2 Docker

Docker on avoimen lähdekoodin konttimoottorisovellus, jonka avulla käyttäjät voivat luoda ja hallinnoida Docker-levykuvia (Docker-image) ja Docker-kontteja. Docker hyödyntää Linux-käyttöliittymän eristys ja virtualisointi ominaisuuksia, joiden avulla kontit eristetään isäntäkäyttöjärjestelmästä ja toisistaan. Merkittävä etu Docker-konteissa on, että se tekee hajautettujen mikropalveluiden hallinnoinnista helppoa ja kontit tarjoavat aina saman ympäristön sovelluksille. Docker-konteilla on helppo luoda eri ympäristöt esimerkiksi tuotanto, testaus ja kehitysversioille, jotka voivat pitää sisällään toisilleen tarpeettomia paketteja. (Docker 2021.)

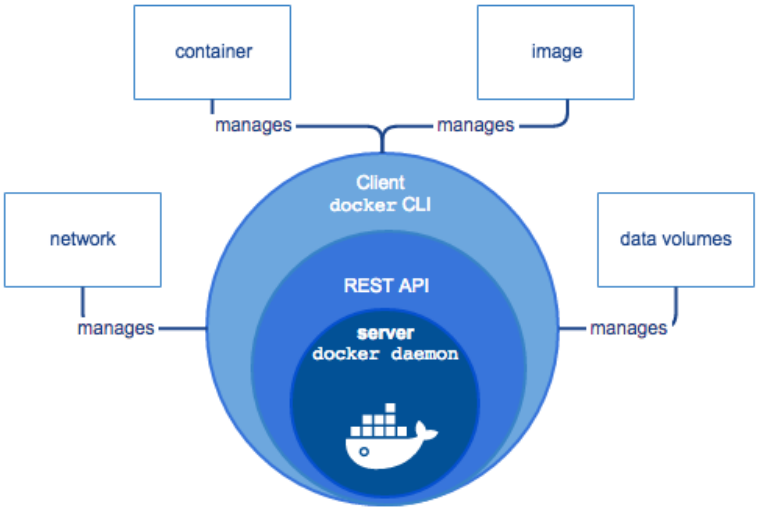
Docker-levykuva pitää sisällään suoritettavan ohjelmakoodin sekä tarvittavat käyttöjärjestelmän kirjastot ja riippuvuudet. Docker-kontti on suoritettava instanssi Docker-levykuvasta. Docker-levykuvan rakentaminen aloitetaan *Dockerfile*-tiedostolla, johon määritellään komentoina levykuvan rakentamisvaiheet (ks. Dockerfile reference n.d). Docker-levykuva on mahdollista rakentaa itse

mutta usein valmiin levykuvapohjan käyttäminen on paras ratkaisu ja samaa pohjakuvaa voidaan hyödyntää useassa Docker-levykuvassa. (Docker 2021.)

Docker-kontit eivät lähtökohtaisesti säilytä mitään tietoa, kun kontti suljetaan. Docker kuitenkin tarjoaa mahdollisuuden kiinnittää isäntäkoneella sijaitsevia kansiota Docker-kontteihin niin että tieto voidaan säilyttää isäntäkoneella. Toinen vaihtoehto on Docker-volyymi (volume), joka toimii lähes samalla tavalla kuin kansioiden kiinnittäminen. Docker-volyymi luo kansion isäntäkoneelle ja se on mahdollista nimetä, nimetty volyymi tallennetaan Docker daemoniin objektina. Volyymi-objekti voidaan yhdistää Docker-konttiin pelkän nimen perusteella eikä polkuja tarvita. (Use Volumes n.d.)

Docker-kontteja on mahdollista yhdistää toisiinsa tai muihin palveluihin Docker-verkkojen avulla. Tämä tarkoittaa sitä, että Docker-verkkoon liitetyt kontit ja palvelut voivat kommunikoida keskenään. Docker tarjoaa useita verkkoajureita käytettäväksi Docker-verkkojen reitityksessä. (Networking overview. n.d.)

Docker on kokonaisuus, joka koostuu useasta työkalusta ja sitä voidaan kutsua Docker Engineksi. Työkalujen avulla käyttäjät voivat rakentaa, suorittaa ja ladata Docker-levykuvia. Docker Engine koostuu kolmesta komponentista Docker Daemon, CLI ja REST API. Docker CLI on käyttöliittymä, jonka kautta käyttäjät voivat antaa komentoja Docker REST API:lle. REST API puolestaan välittää komennot Docker daemon taustaprosessille, joka vastaa Docker-levykuvista, konteista, verkoista ja varastoista. (Ks. kuvio 10.) (Docker Architecture n.d; Wilson 2022)



Kuvio 10. Docker Engine (Docker Architecture n.d)

5.3.3 Docker Compose

Docker Compose on työkalu, jota voidaan käyttää Docker-konttiympäristön määrittämiseen ja suorittamiseen. Sitä hyödynnetään opinnäytetyössä tuotanto ja kehitysympäristöjen luomiseen. Konttiympäristön konfiguraatio määritellään *docker-compose.yml* -tiedostoon ja se voidaan suorittaa yhdellä komennolla, jolloin Docker Compose käynnistää kaikki tiedostossa määritellyt Docker-kontit (ks. Docker-compose cheatsheet. n.d). Docker Composella voidaan käynnistää useita konttiympäristöjä samanaikaisesti ja niitä on mahdollista eristää toisistaan projektinimien perusteella. Tämä tarkoittaa sitä, että sama palveluympäristöt voidaan käynnistää eri projektinimillä ja ympäristöt eivät ole tietoisia toisistaan. (Docker 2021.)

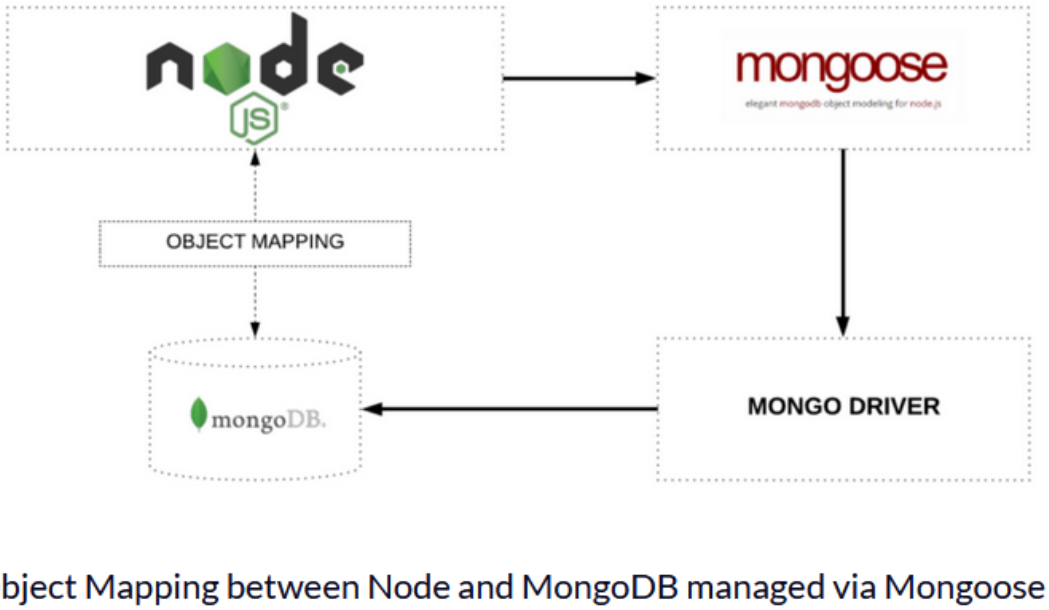
Kun *docker-compose.yml* -tiedosto suoritetaan ensimmäisen kerran, Docker Compose lataa kaikkien määriteltyjen konttien Docker-levykuvat ja käynnistää ne kontissa. Docker-levykuvan muuttuessa, Docker Compose rakentaa vain muuttuneen levykuvan kontin uudelleen. Docker Compose konfiguraatiotiedostoon voidaan myös määrittää Docker-verkkoja sekä volyymeja. Lisäksi Docker Compose mahdollistaa ympäristömuuttujien määrittämisen konteille. Ympäristömuuttujilla palveluille voidaan käynnistyksen yhteydessä antaa esimerkiksi salausavaimia tai kirjautumistunnuksia. (Overview of Docker Compose n.d; Docker 2021.)

5.4 Tietokannat

5.4.1 MongoDB

MongoDB on korkean volyymin NoSQL-tietokanta joka SQL-relaatiotietokannan taulukkorakenteesta poiketen käyttää dokumenttikokoelmia tiedon tallentamiseen JSON (JavaScript Object Notation) -formaattiin. Dokumentit ovat dynaamisia ja niiden kentät koostuvat avain-arvo-pareista, jotka ovat MongoDB-tietokannan perusyksikköjä. Koska MongoDB tarjoaa joustavan ja dynaamisen tavan tallentaa tietorakenteita, yksittäisen kokoelman sisällä olevat dokumentit voivat pitää sisällään erilaisia avain-arvo-pareja. (Taylor, D 2022.)

MongoDB valittiin toteutuksen tietokannaksi siitä syystä, että se tarjoaa yksinkertaisen, joustavan ja suorituskykyisen tietokannan suurille tietomäärille ja lisäksi kuvantunnistusrajapinnasta saatavat tietorakenteet sopivat hyvin JSON-formaattiin. Toteutuksessa hyödynnettiin lisäksi Mongoose JavaScript-kirjastoa, joka on ODM (Object Data Modeling) -kirjasto ja se tuo Node.js ympäristöön funktioita MongoDB-tietokannan hallintaan sekä mahdollistaa objektikaavojen käytön. (Ks. kuvio 11.)



Kuvio 11. MongoDB, Mongoose diagram (Karnik 2018)

5.4.2 Redis

Redis (Remote Dictionary Server) on avoimen lähdekoodin avain-arvo-pari-tietokanta, joka tallentaa tiedot keskusmuistiin ja luo keskusmuistista tilannekuvia (snapshot). Redis on suorituskyvyltään erittäin nopea ja sitä voidaan hyödyntää jonona tai välimuistina toisille tietokannoille. Redis tarjoaa myös mahdollisuuden säilyttää tieto vain keskusmuistissa ja tämän ominaisuuden ansiota se sopii hyvin tietokannaksi tiedoille, joilla on voimassaoloaika kuten autentikointitunnukset. (What is Redis n.d.)

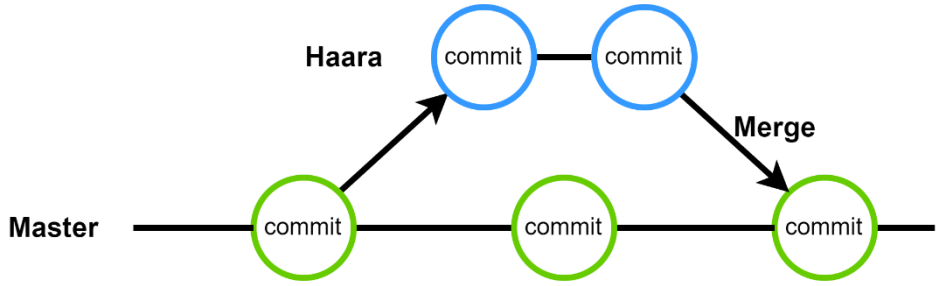
Redis valittiin toteutukseen siitä syystä, että se tarjoaa valmiita toimintoja ajastettuun tiedon säilytyksen ja koska tiedot ovat tallennettu keskusmuistiin, mitään tietoja ei voida enää palauttaa poistamisen jälkeen. Toteutettavassa verkkopalvelussa edellä mainitut asiat ovat tärkeässä roolissa käyttäjien kirjautumis- ja sessiohallinnassa.

5.5 Projektihallinta ja DevOps

5.5.1 Git-versiohallinta

Git on versionhallintatyökalu, jonka Linux yhteisö kehitti vuonna 2005 tukemaan käyttöjärjestelmän kehitysprosessia. Git luo arkistorakenteen haluttuun kansioon ja seuraa arkiston sisällä olevien tiedostojen muutoksia ja tallentaa tilannekuvia jokaisesta muutoksesta. Arkiston tilannekuviin voidaan palata takaisin tai vertailla aikaisempien tilannekuvien välisiä eroja. Lisäksi Git seuraa ja tallentaa tiedon kuka muutokset arkiston tiedostoihin on tehnyt. Git on suunniteltu siten että arkistosta on vaikea poistaa tilannekuvia, joten se minimoi tilanteita, joissa lähdekoodi olisi mahdollista kadota pysyvästi. (Chacon & Straub 2022, 14–26.)

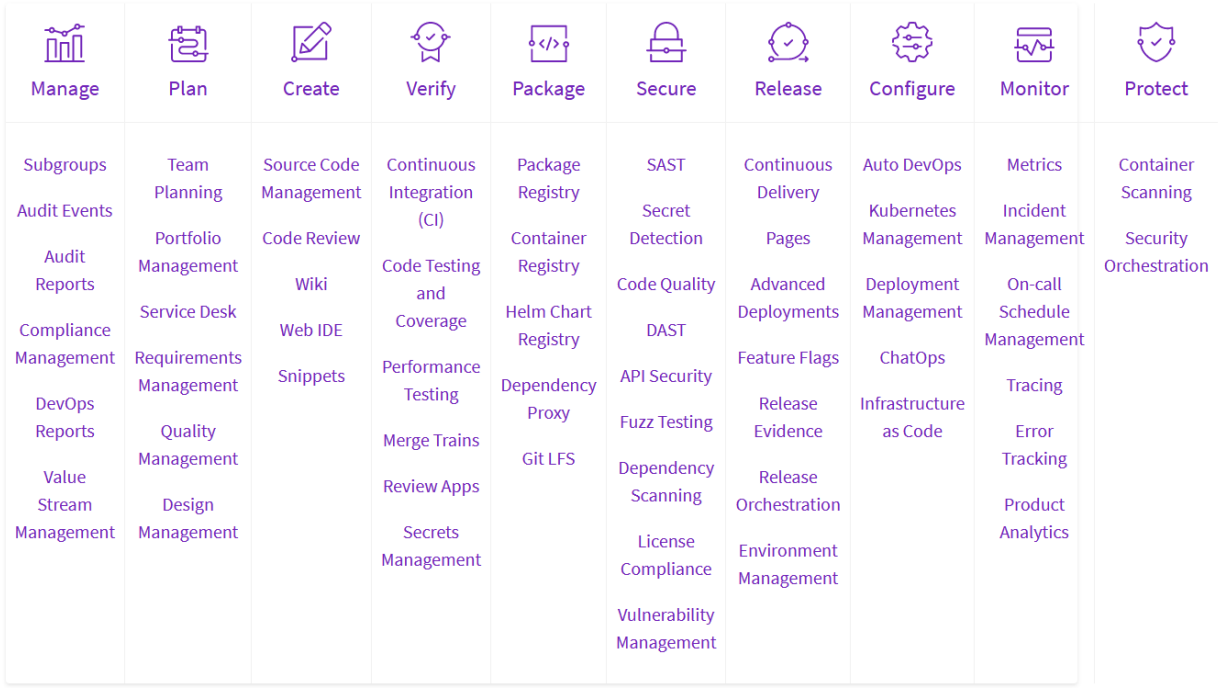
Yksi Gitin hyödyllisistä ominaisuuksista on haarautuminen (branching) joka tarkoittaa, että arkistosta voidaan erottaa uusi arkistohaara, johon tehtävät muutokset eivät vaikuta alkuperäiseen arkistohaaraan. Luotu haara voidaan yhdistää (merge) takaisin alkuperäiseen haaraan, jolloin tallennetut muutokset siirtyvät alkuperäiseen arkistohaaraan. Tämä ominaisuus mahdollistaa ohjelmistokehityksessä sen, että päähaaran lähdekoodin laatua voidaan valvoa tarkemmin. (Ks. kuvio 12.)



Kuvio 12. Git branching

5.5.2 GitLab

GitLab on vuonna 2011 alkanut avoimen lähdekoodin DevOps-alusta, joka tarjoaa työkaluja palvelutuotantoon. GitLab on ennen kaikkea tallennuspaikan Git-arkistoille ja se on täysin integroitu Gitin kanssa sekä tarjoaa verkkosivupohjaisen käyttöliittymän, jonka kautta käyttäjät voivat selata yksityisiä tai julkisia Git-arkistoja tai GitLab projekteja. Arkistohallinnan lisäksi GitLab tarjoaa valtavan määrän erilaisia työkaluja esimerkiksi projektihallintaan, julkaisuun ja monitorointiin (ks. kuvio 13). (About GitLab n.d)



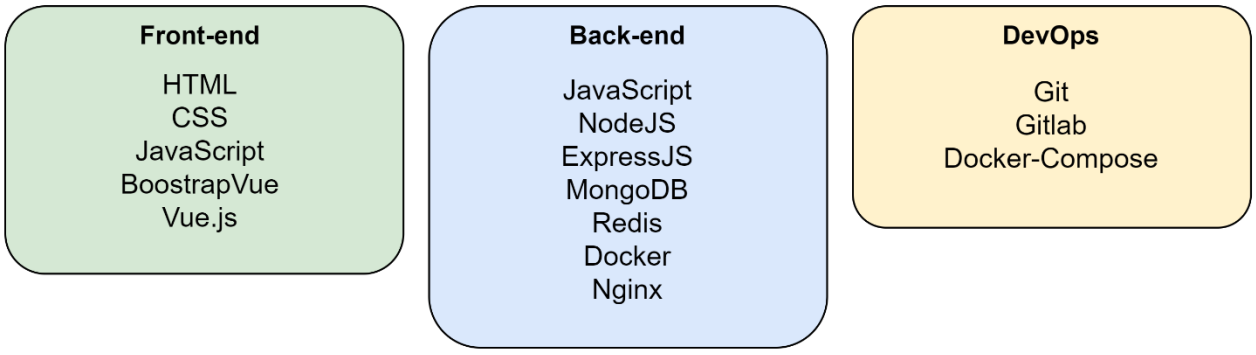
Kuvio 13. Gitlab ominaisuudet (Gitlab n.d)

GitLab valittiin työhön koska se on avoimen lähdekoodin alusta ja tarjoaa laajan työkalu tarjonnan. Työssä hyödynnettiin Git-arkistohallinnan lisäksi konttirekistereitä sekä CI/CD-tehtäväputkia, joilla palvelu lähdekoodi rakennetaan Docker-levykuvaksi automaattisesti lähdekoodin muuttuessa. Gitlabin Docker-konttirekisteri on työn kannalta erityisen merkittävä, sillä sen avulla tuotanto- ja kehitysympäristöjen pystytys on helposti hallittavissa.

5.6 Yhteenveto

Palveluun valitut teknologiat muodostavat MEVN-ohjelmistopinon, jota voidaan pitää yhtenä JavaScript-ohjelmistopinoista. JavaScript-pinoilla viitataan sovelluksessa käytettyihin teknologioihin ja kun front-end ja back-end kehityksessä käytetään JavaScript-ohjelmointikieltä, voidaan puhua JavaScript-ohjelmistopinosta. Lyhenne MEVN tulee käytettyjen teknologioiden nimistä MongoDB, Express, Vue ja Node. (Ks. kuvio 14.)

JavaScript-ohjelmistopino yksinkertaistavat kehitysympäristöä ja mahdollistaa JavaScript-ohjelmointikieltä osaavan henkilön työskentelyn sekä front-end että back-end kehityksessä. Edellä mainittu on suuri etu kehitettävän palvelun jatkokehityksen kannalta.



Kuvio 14. Teknologiavalinnat

6 Toteutus

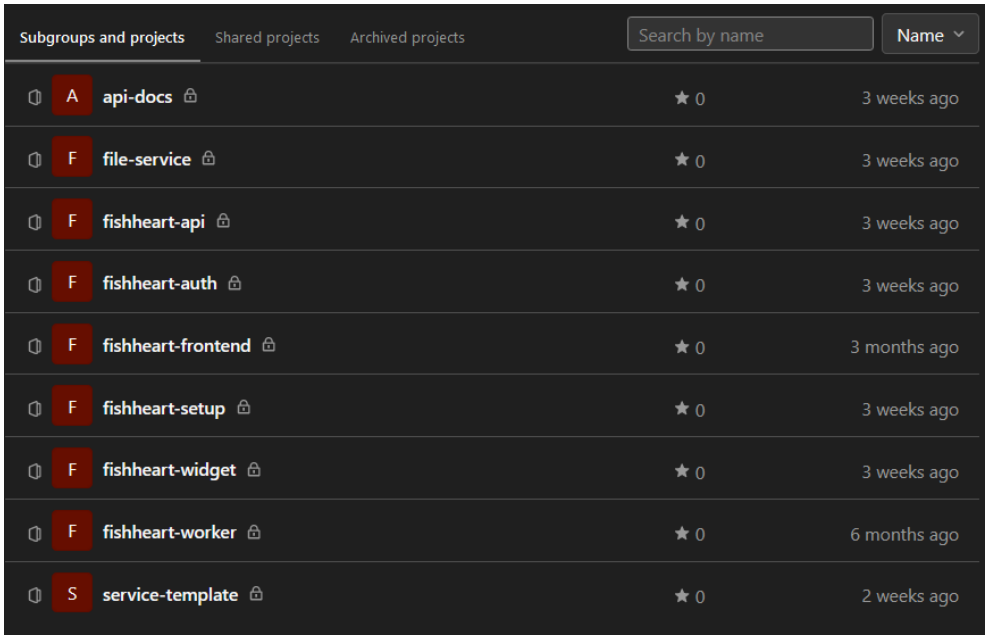
6.1 Sovelluskehitysympäristö

Kehitysympäristönä käytettiin Windows 11-käyttöjärjestelmään, jonka WSL (Windows Subsystem for Linux) -järjestelmäarkkitehtuurin päälle asennettiin Ubuntu-20.04 Linux-jakelu. Ubuntuun asennettiin Docker-konttimoottorisovellus sekä Node-ajoympäristö ja NPM-pakettihallinta. Git -versiohallintatyökalu yhdistettiin GitLab-käyttäjätunnukseen SSH-avaimen avulla.

Ohjelmointiympäristönä käytettiin VS Codea, jonka *Remote – WSL*-laajennusosalla oli mahdollista yhdistää Windows-ympäristössä suoritettava VS Code suoraan WSL:ssä suoritettavan Ubuntun kansioihin (ks. Remote – WSL n.d). VS Codeen asennettiin myös *Docker* -laajennus, jonka avulla Ubuntussa suoritettavalle Dockerille voitiin välittää komentoja. Lisäksi VS Code-editoriin asennettiin GitLens-laajennus, jonka avulla GitLab-tunnistautuminen ja Git-toiminnot olivat käytettävissä suoraan editorista.

6.2 GitLab-ympäristö

Kehitettävää verkkopalvelua varten luotiin yksityinen GitLab-projekti, jonka alle jokaiselle mikropalvelulle luotiin Git-arkisto. Lisäksi GitLab-projektiin luotiin omat arkistot tukiohjelmille kuten dokumentaation generoinnille ja tuotantopalvelimen asennusskripteille (ks. kuvio 15). GitLabin tarjoamia CI/CD-tehtäväputkia hyödynnettiin rakentamaan mikropalveluiden Docker-levykuvat. Rakennetut Docker-levykuvat tallennettiin Gitlab-projektin levykuvarekisteriin, josta ne voitiin ladata tuotantopalvelimelle ja paikalliseen kehitysympäristöön.



Kuvio 15 Gitlab-projektinäkymä

GitLab CI/CD-tehtäväputkia varten erilliselle pilvipalvelutarjoajan Ubuntu-pohjaiselle virtuaalipalvelimelle asennettiin GitLab Runner-sovellus joka rekisteröitiin Gitlab-projektille. GitLab Runnerin avulla CI/CD-tehtäväputkia voitiin automaattisesti suorittaa Git-arkistojen lähdekoodin muuttuessa. (Ks. kuvio 16.)

```
#update repos and add docker
sudo apt update && sudo apt upgrade -y
sudo apt-get install apt-transport-https ca-certificates curl software-properties-common -y
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
sudo apt update

#install docker
sudo apt install docker-ce -y
sudo groupadd docker
sudo systemctl restart docker.socket
sudo systemctl restart docker.service

#download gitlab runner binary and setup user
sudo wget -O /usr/local/bin/gitlab-runner https://gitlab-runner-downloads.s3.amazonaws.com/latest/binaries/gitlab-runner-linux-amd64
sudo chmod +x /usr/local/bin/gitlab-runner
sudo useradd --comment 'GitLab Runner' --create-home gitlab-runner --shell /bin/bash
sudo gitlab-runner install --user=gitlab-runner --working-directory=/home/gitlab-runner

#give docker privilges to runner
sudo usermod -aG docker gitlab-runner
sudo gitlab-runner start

#test if runner has docker priviledges
sudo -u gitlab-runner -H docker info
```

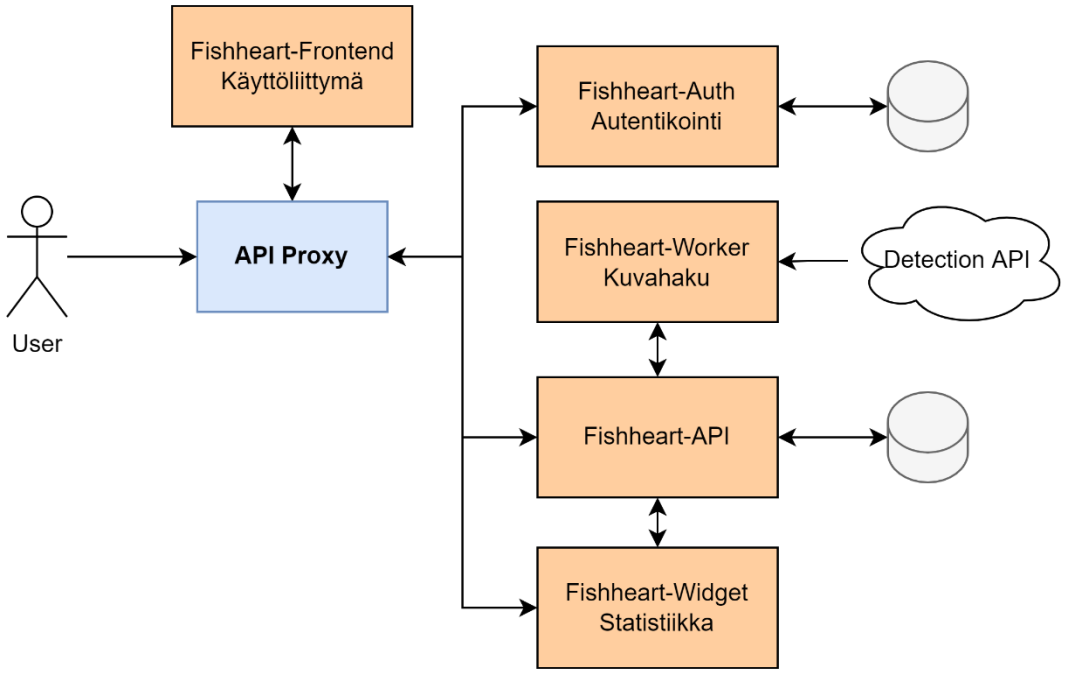
Kuvio 16. Gitlab Runner asennus

6.3 Mikropalvelut

6.3.1 Tunnistetut mikropalvelut

Palvelun toiminnallisuuksille suoritettiin tehtävärajaukset, joiden tavoitteena oli löytää selkeät yksittäiset toiminnalliset kokonaisuudet, jotka vaikuttavat palvelun skaalaamiseen. Palvelusta tunnistettiin kuusi eri toiminnallista kokonaisuutta, jotka eristettiin mikropalveluiksi. Lisäksi palvelusta tunnistettiin myös tietokannat eri mikropalveluiden tarpeisiin. Kaikki palvelussa käytetyt tietokannat suoritettiin Docker-konteissa ja ne konfiguroitiin Docker Compose -tiedostossa palvelinympäristön pystytyksen yhteydessä.

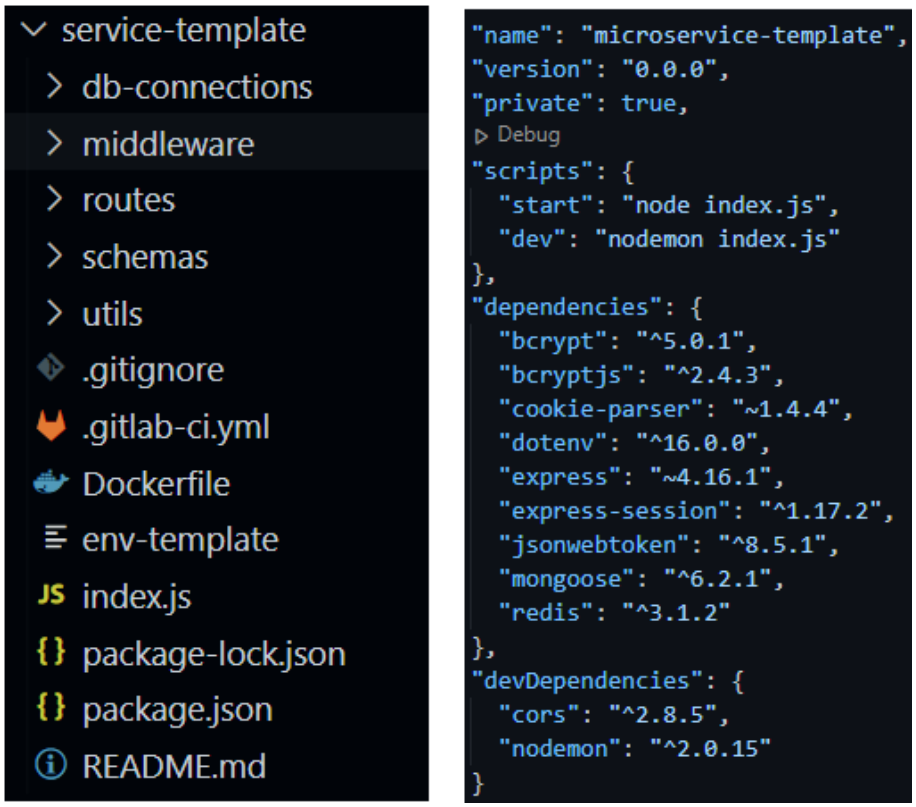
Yksi merkittävä toiminnallisuus, joka mahdollistaa mikropalveluarkkitehtuurin on välityspalvelin (Proxy), joka ohjaa käyttöliittymästä tulevat pyynnöt oikeille mikropalveluille. Käyttöliittymän tunnistaminen mikropalveluksi saattaa tuntua erikoiselta mutta sille on selkeä syy, joka esitellään luvussa 6.3.7. Mikropalveluiksi sopivia toiminnallisuuksia tunnistaessa on syytä pitää mielessä, että toiminnallisuuksia ei lähdetä pilkkomaan tarpeettoman pieniksi, jotta kokonaisuus pysyy hallinnassa. (Ks. kuvio 17.)



Kuvio 17. Tunnistetut mikropalvelut

6.3.2 REST-rajapinnat

Mikropalveluiden REST-rajapinnat toteutettiin Node.js ympäristössä hyödyntäen Express-sovelluskehystä. Mikropalveluiden pohjaksi luotiin yksinkertainen Express-sovellusrunko, johon luotiin funktiot tietokantayhteyksiä ja JWT-validointia varten sekä asennettiin valmiiksi tärkeimmät kirjastot (ks. kuvio 18). Lisäksi runkoon luotiin valmis *Dockerfile*-tiedosto joka kontittaa Express-sovelluksen ja *.gitlab-ci.yml* -konfiguraatiotiedosto joka suorittaa Gitlab integraatio tehtäväputken kun lähdekoodia muutetaan (ks. kuvio 19).



Kuvio 18. Express-sovellusrunko ja kirjastot

```
FROM node:15.8-alpine
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
CMD ["npm","start"]
```

Kuvio 19. Dockerfile

Mikäli Git-arkistossa on *.gitlab-ci.yml*-tiedosto, yrittää GitLab suorittaa kyseisessä tiedostossa määritellyt CI/CD-tehtävät aina kun arkistoon tehdään muutos. Mikäli jokin CI/CD-tehtävistä epäonnistuu, tulee siitä virheilmoitus GitLabin lokiin.

Express-rungossa käytetty *.gitlab-ci.yml*-tiedosto koostui kahdesta työvaiheesta, Docker-kuvan rakentamisesta ja GitLabin tarjoamasta SAST (Static Application Security Testing) vaiheesta. Työvaiheiden järjestys määritetään *stages*-avainsanalla. *Build*-avainsana määrittää Docker-imagen rakennusvaiheen. Tärkeimpänä huomiona on *services*-avainsana, jolla voidaan määrittää Docker-levykuva, jonka sisällä työvaihe suoritetaan. Koska rakennusvaiheen tuloksena halutaan luoda Docker-levykuva, joka voidaan tallentaa levykuvarekisteriin täytyi rakennusvaiheen pohjakuvaksi valita *docker:dind* joka tarkoittaa Docker-in-Docker.(Ks. kuvio 20.)

```
stages:
- build
- test
include:
- template: Security/SAST.gitlab-ci.yml
- template: Security/Secret-Detection.gitlab-ci.yml
- template: Security/SAST-IaC.latest.gitlab-ci.yml
build:
  tags:
  - builder
  image: docker:latest
  only:
  - tags
  - main
  stage: build
  services:
  - name: docker:dind
  variables:
    DOCKER_DRIVER: overlay2
    DOCKER_TLS_CERTDIR: ''
    IMAGE_TAG: "$CI_REGISTRY_IMAGE:$CI_COMMIT_REF_SLUG"
  before_script:
  - docker login -u "$CI_REGISTRY_USER" -p "$CI_REGISTRY_PASSWORD" "$CI_REGISTRY"
  script:
  - docker build -t "$IMAGE_TAG" .
  - docker push "$IMAGE_TAG"

phpcs-security-audit-sast:
  artifacts:
    paths:
      - gl-sast-report.json
```

Kuvio 20. Gitlab CI/CD konfiguraatio-tiedosto.

Tietokantayhteyksien luomiseen hyödynnettiin *mongoose*- ja *redis* -kirjastoja, joiden ansiosta tietokantayhteyksien avaaminen tarvittaessa oli vaivatonta. Tietokantayhteys-funktiot toteutettiin siten että tietokantojen kirjautumistiedot ja sessioavaimet luetaan ympäristömuuttujista. Tällä es-

tetään avainten, kirjautumistietojen joutuminen Git-arkistoon ja lähdekoodiin. Kontitetun sovelluksen käynnistyksen yhteydessä kirjautumistiedot ja sessioavaimet voidaan antaa ympäristömuuttujina kontille, jolloin sovellus lukee ne. (ks. kuvio 21).

```
const mongoose = require('mongoose');
const redis = require('redis');

module.exports.ConnectMongo = () => {
    mongoose.connect(
        `mongodb://${process.env.MONGO_URI}:${process.env.MONGO_PORT}/${process.env.MONGO_DB}`,
        {
            authSource: process.env.MONGO_AUTH,
            useNewUrlParser: true,
            useUnifiedTopology: true,
            user: process.env.MONGO_INITDB_ROOT_USERNAME,
            pass: process.env.MONGO_INITDB_ROOT_PASSWORD
        },
        () => {
            console.log(`connected to mongo: mongodb://${process.env.MONGO_URI}:${process.env.MONGO_PORT}/${process.env.MONGO_DB}`);
        }
    );
}

module.exports.ConnectRedis = () => {
    // Redis client connection
    return redis.createClient({
        host: process.env.REDIS_URI,
        port: process.env.REDIS_PORT,
        no_ready_check: true,
        password: process.env.REDIS_PASS
    });

}
```

Kuvio 21. Tietokantayhteys-funktiot

Mikropalveluiden rajapintapolut ja niiden funktiot määritettiin mikropalvelu kohtaisesti *routes*-kansioon, erillisiin tiedostoihin. Rajapintapolut toteutettiin Express-verkkokehyksen mukaisesti ja ne dokumentoitiin JSDoc-merkintäkielellä. Rajapintapoluissa hyödynnettiin Expressin mahdollistamia middleware-väliohjelmistoja, joilla esimerkiksi käyttäjän oikeudet tarkastettiin ennen kuin rajapintapolun funktio suoritetaan. HTTP-vastauskoodeille luotiin globaalit muuttujat, jotka nimettiin vastauskoodin mukaisesti. MongoDB-tietokantaoperaatiot toteutettiin hyödyntämällä mongoose-kaavoja, joiden avulla dokumenttikokoelmille voitiin suorittaa CRUD-operaatioita. Redis on paljon yksinkertaisempi tietokanta, joten tietokantaoperaatiot suoritettiin *redis*-kirjaston tarjoamilla funktioilla. (Ks. kuvio 22.)

Express polku

```
const validate = require('../utils/jwt-verity');
const router = require('express').Router();
const Image = require('../mongo-models/image');

/**
 * @name GET api/image/get
 * @function
 * @memberof module:image
 * @param {req.headers.authorization}  jwt token
 * @param {req.headers.id}  image_id
 * @return {object} Labels object
 * @author viertola
 * @description Returns image with image_id.
 * @implements IMAGE_GET
 */
router.get('/get', validate.User, (req, res) => {
    try {
        Image.find({ imageID: req.headers.id }).then((result) => {
            if (result) {
                res.status(OK).json(result);
            } else {
                res.sendStatus(BAD_REQUEST);
            }
        });
    } catch (error) {
        res.sendStatus(SERVER_ERROR);
    }
});
```

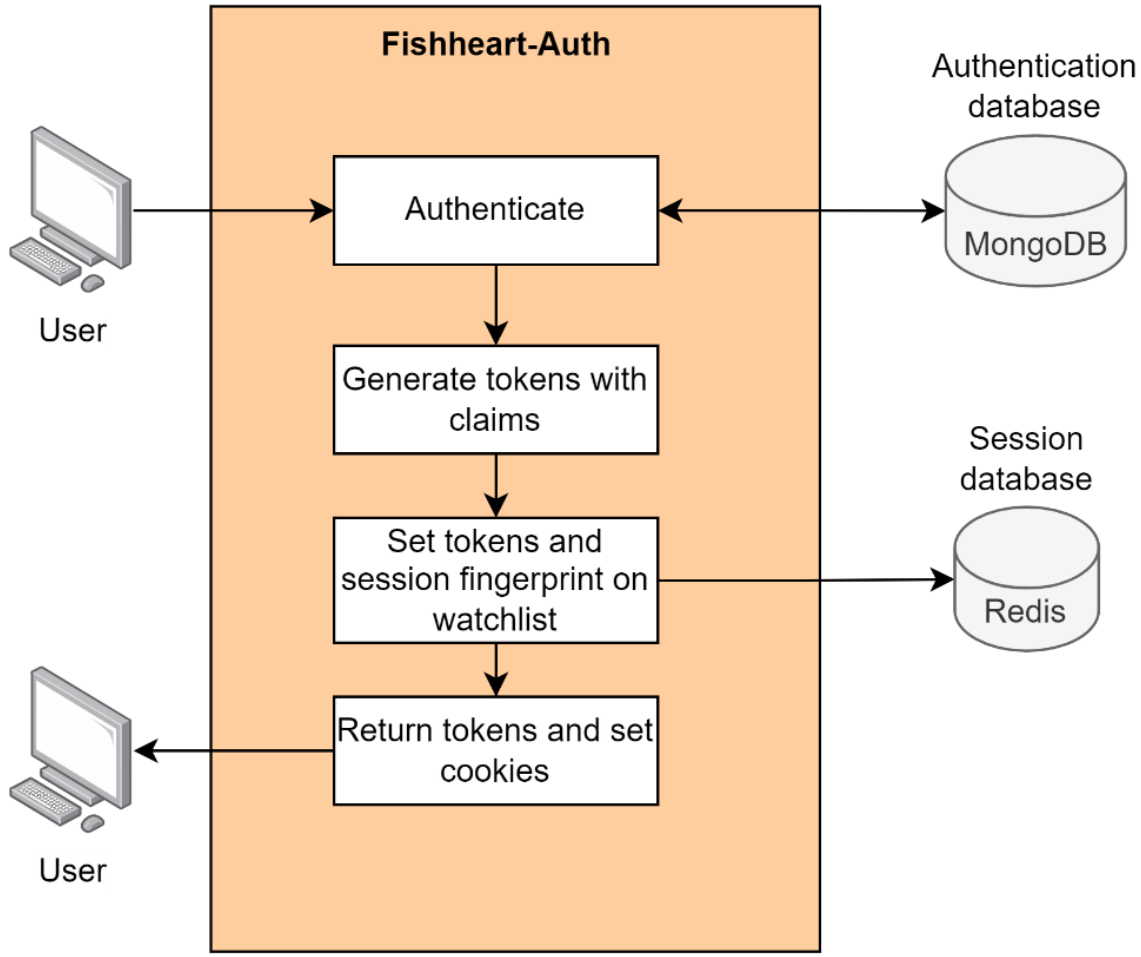
Mongoose-kaava

```
var mongoose = require('mongoose');

var imageSchema = new mongoose.Schema({
    imageID: {
        type: String,
        required: true,
        unique: true,
    },
    source: {
        type: String,
        required: true,
    },
    date: {
        type: Date,
        required: true,
    },
    utc_date: {
        type: Date,
        required: true,
    },
    reviewed: { type: String, default: null },
    thumbnail: {
        type: String,
        required: true,
    },
    filename: String,
    image: { type: String, default: null },
    downloaded: {
        type: Boolean,
        default: false,
    },
    geoloc: Array,
    result: Array,
    size: String,

    show: { type: Boolean, default: false },
    review_date: { type: String, default: null },
});

module.exports = mongoose.model('Image', imageSchema);
```

Kuvio 22. Express polku ja mongoose-kaava

6.3.3 Fishheart-Auth

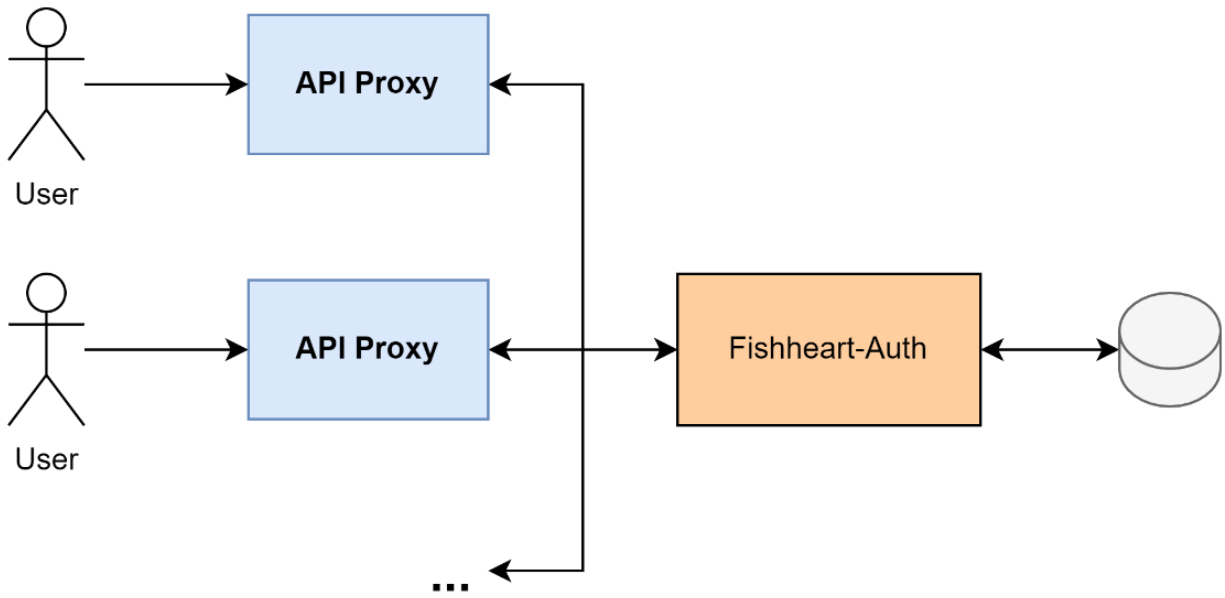
Fishheart-Auth vastaa käyttäjien autentikoinnista sekä käyttäjien valtuuttamisesta JWT-tokenien avulla (ks. Introduction to JSON Web Tokens n.d). Käyttäjien autentikointi toteutettiin hyvin perinteisellä sähköposti ja salasana periaatteella. Valtuuttaminen toteutettiin hyödyntäen kahta eri JWT-tokenia sekä istuntosormenjälkeä (session-fingerprint). (Ks. kuvio 23.)



Kuvio 23. Käyttäjän valtuuttaminen

Autentikointipalvelun tietokantojen toteutuksessa hyödynnettiin MongoDB:tä sekä Redistä. MongoDB vastaa kirjautumistietojen tallentamisesta ja Redis puolestaan vastaa JWT-tokenien ja istuntosormenjälkien seuraamisesta ja terminoinnista. Redis tarjoaa mahdollisuuden asettaa TTL (Time-To-Live) -ajan tallennetuille objekteille ja ajan umpeutuessa objektit poistetaan tietokannasta. TTL-toiminnon avulla voimassa olevia JWT-tokeneita pysyttiin seuraamaan ja varmistamaan että käyttäjän kirjautuessa ulos palvelusta, tokeneita ei ole enää mahdollista käyttää, vaikka JWT-tokeni, olisi vielä voimassa.

Autentikointipalvelu tunnistaminen mikropalveluksi mahdollistaa tilanteen, jossa palvelua monistetaan laite tai asiakas kohtaiseksi. Tässä tapauksessa Fishheart-Auth voi palvella useaa palveluinstanssia autentikoinnissa. (Ks. kuvio 24.)



Kuvio 24. Fishheart-Auth skaalautuminen

6.3.4 Fishheart-API

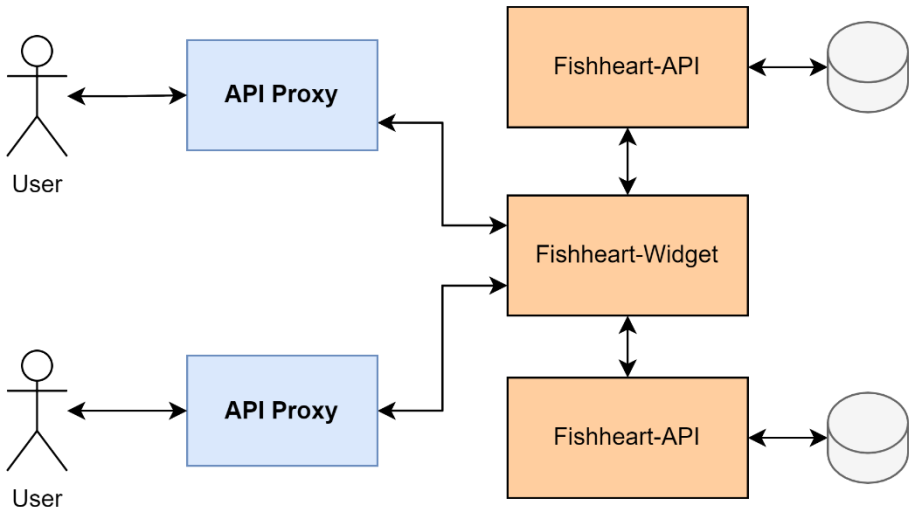
Fishheart-API on ”yleinen rajapinta”, joka vastaa verkkopalvelun toiminnallisuuksista ja tietokanta-toiminnoista, jotka ovat suoraan käyttöliittymään yhteydessä kuten esimerkiksi zip- ja excel-tiedostojen luonti, kuvien, statistiikan ja raporttitietojen hakeminen. Fishheart-API on suurin yksittäinen mikropalvelu, joka palveluun toteutettiin.

Fishheart-API yhdistettiin MongoDB-tietokantaan, joka toimi palvelun päätietokantana. Tietokantaan tallennettiin kuvantunnistusrajapinnasta saatuja kuvia, käyttöliittymän työkaluilla tuotettua dataa ja statistiikkaa sekä palvelu ja käyttäjäkohtaisia asetuksia. Tietokanta operaatioiden tunnistaminen mikropalveluksi mahdollistaa skaalautuvuutta palvelurakenteessa mutta Fishheart-API:n on tarkoitus olla hitaasti skaalautuva mikropalvelu. Fishheart-API:lle varataan muita mikropalveluita suuremmat resurssit ja muut palvelut skaalataan sen ympärille tarpeen vaatiessa.

6.3.5 Fishheart-Widget

Fishheart-Widget on statistiikkapalvelu, jonka tehtävä on ylläpitää laitekohtaista statistiikkaa ja kokoelma parhaita kuvia sekä tarjota asiakkaille rajapinta, josta nämä tiedot ovat käytettävissä. Statistiikkarajapinnan eristäminen mikropalveluksi mahdollistaa tilanteen, jossa yksi Fishheart-Widget voi palvella useaa palveluinstanssia.

Koska statistiikkapalvelun tarkoitus on tarjota yksinkertainen rajapinta, ei se hyödynnä mitään tietokantaa vaan laitekohtaiset kuvat tallennetaan staattisina *JPEG*-kuvina mikropalvelun sisälle tiedostorakenteeseen. Laitekohtainen statistiikka noudetaan tunneittain Fishheart-API:n tietokannasta ja siitä luodaan JSON-tiedosto, johon staattisten kuvien linkit lisätään. Asiakas voi implementoida statistiikkaa ja ladata kuvia omille sivuilleen rajanpinnan avulla. (Ks. kuvio 25.)

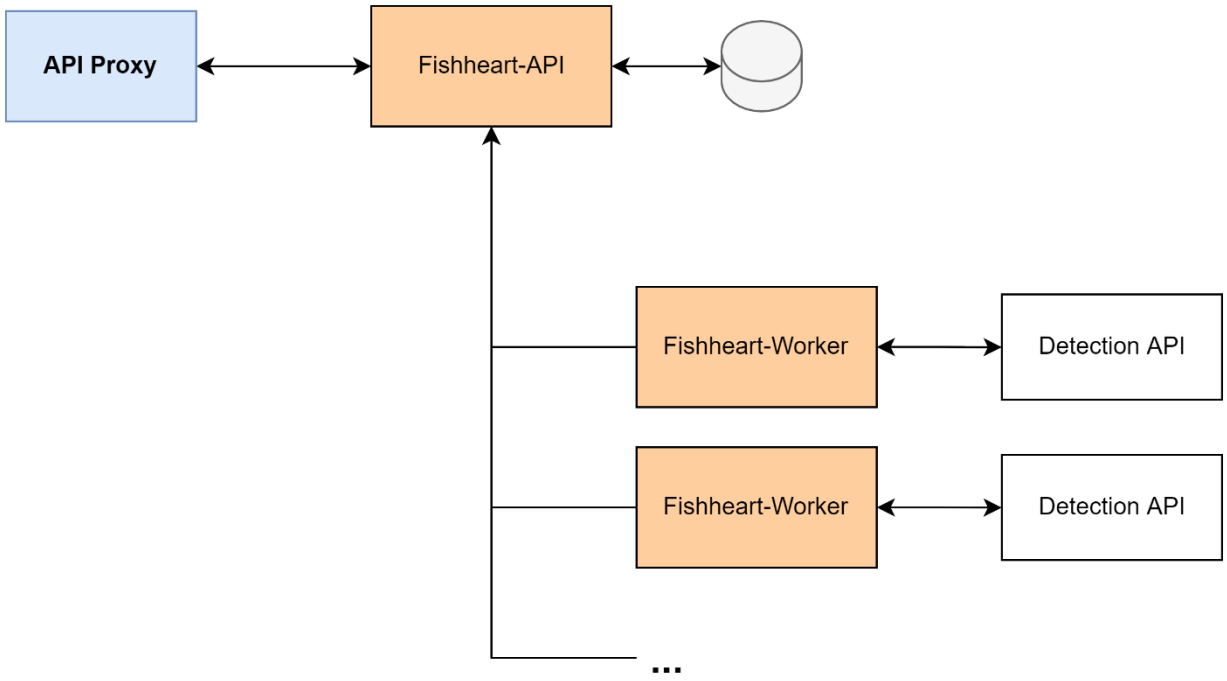


Kuvio 25. Statistiikkapalvelun skaalaus

Fishheart-Widgetin rajapintaan voidaan luoda jokaiselle käyttäjälle yksilöllinen polku, jonka kautta statistiikka on käytettävissä. Lisäksi käyttäjille voidaan määrittää laitekohtaisesti mitä statistiikkaa he saavat rajapinnasta.

6.3.6 Fishheart-Worker

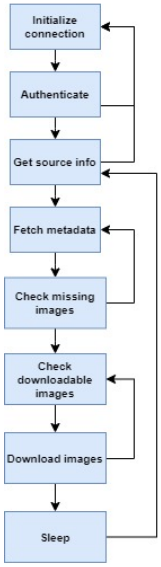
Fishheart-Worker on kuvahakupalvelun, jonka tehtävä on hakea uusia kuvia kuvatunnistusrajapinnasta ja tallentaa ne Fishheart-API:n kautta tietokantaan. Kuvahakupalvelu eristettiin mikropalveluksi koska se mahdollistaa räätälöityjen palvelukokoonpanojen tarjoamisen riippuen mihin suuntaan asiakas haluaa tulevaisuudessa kehittää järjestelmää ja kuvantunnistusrajapintoja. (Ks. kuvio 26.)



Kuvio 26. Fishheart-Worker skaalaus

Fishheart-Worker toteutettiin yksinkertaisena tilakoneena, joka ottaa yhteyden ja kirjautuu kuvantunnistusrajapintaa, jonka jälkeen rajapinnalta pyydetään uutta metadataa ladattavista kuvista. Mikäli metadataa löytyy, uudet kuvat ladataan ja tallennetaan tietokantaan. Kuvantunnistusrajapinnasta saatavat kuvat tallennetaan tekstimuodossa base64-koodattuina MongoDB-tietokantaan. Kuvantunnistusrajapinta hyödyntää metadata pyynnöissä hash-merkkijonoa, joka toimii kirjanmerkkinä haettaessa uusinta metadataa rajapinnasta. Fishheart-Worker käyttää tätä hash-merkkijonoa kutsuessaan kuvantunnistusrajapintaa.

Tilakoneeseen implementoitiin dynaaminen *SLEEP*-viive, jotta tilakoneen sykliä saatiin kontrolloitua tarpeen mukaan eikä tilakone aiheuta turhaa resurssipainetta kuvantunnistusrajapinnalle ja mikropalvelua suorittavalle palvelimelle. Tilakoneeseen toteutettiin *BUSY*, *FORCE* ja *SLEEPING*-tilat, joilla tilakoneen intervalliin pystyttiin reagoimaan dynaamisesti ja pakottamaan erilaisia tilasekvenssejä tarpeen mukaan. Tilakoneen vaiheista tallennettiin lokitietoa tietokantaan ja käyttöliittymään implementoitiin kehittäjille tarkoitettu näkymä josta tilakoneen lokia, oli mahdollista seurata. (Ks. kuvio 27.)




```
module.exports.INIT = {
    name: 'INIT',
    func: lib.initFunc,
    log: lib.writeLog,
    next: 'TOKEN',
    fallback: 'INIT',
};
module.exports.TOKEN = {
    name: 'TOKEN',
    func: lib.fetchToken,
    log: lib.writeLog,
    next: 'SOURCE',
    fallback: 'INIT',
};

module.exports.SOURCE = {
    name: 'SOURCE',
    func: lib.fetchSourceData,
    log: lib.writeLog,
    next: 'FETCH',
    fallback: 'INIT',
};
```

```
const StateMachine = async () => {
    TEST_CONNECTION();
    if (!BUSY && FORCE) await EXEC_FORCE_STATE();
    if (!BUSY && !FORCE && !SLEEPING) STATE = (await EXEC_CURRENT_STATE()) ? state[STATE.next] : state[STATE.fallback];
    if (!FORCE && SLEEPING) {
        if (SLEEPING._destroyed) {
            SLEEPING = false;
            UPDATE_INTERVAL = default_interval;
            clearInterval(Main);
            Main = setInterval(StateMachine, UPDATE_INTERVAL);
        } else {
            var timeleft = Math.ceil((SLEEPING._idleStart + SLEEPING._idleTimeout) / 1000 - process.uptime());
            console.log(timeleft);
        }
    }
};
```

Kuvio 27. Kuvahakupalvelu tilakone

6.3.7 API Proxy

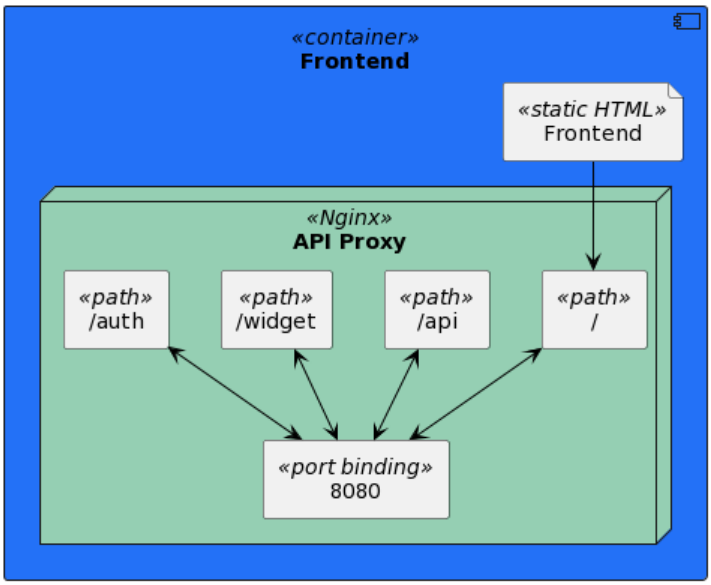
API Proxy on rajanpintakutsujen välityspalvelin, joka sitoo kaikki mikropalvelut yhteen. Se siis toimii välityspalvelimena käyttöliittymästä tuleville pyynnöille. Sitä ei ole tarkoitettu skaalautuvaksi palveluksi vaan se sidotaan palvelimen domain-osoitteeseen. API Proxy toteutettiin Nginx-välityspalvelin ohjelmistolla ja se poikkeaa teknisesti muista mikropalveluista huomattavasti luonteensa takia.

Aikaisempien lukujen kuvissa API Proxy on kuvattu erillisenä objektina, se ei kuitenkaan pidä aivan paikkaansa. Tämä johtuu siitä, että teknisesti API Proxy ja Fishheart-Frontend-käyttöliittymä elävät samassa Docker-kontissa. Kontin rakentamisessa hyödynnettiin Dockerin multi-stage rakennusta, jonka avulla Docker-kontti voidaan rakentaa vaiheittain useasta eri Docker-imagesta. Eri rakennusvaiheiden välillä voidaan valita mitä rakennusvaiheesta siirtyy seuraavaan vaiheeseen. (Ks. kuvio 28.)

```
# Ensimmäinen rakennusvaihe
# Vue-sovellus rakennetaan /app kansioon
FROM node:15.8-alpine as build
WORKDIR /app
COPY package*.json /app/
COPY . .
RUN npm install --silent
RUN npm run build

# Toinen rakennusvaihe
FROM nginx
# Kopioidaan edellisestä vaiheesta kansio /app/dist/
COPY --from=build /app/dist/ /var/www/html
COPY default.conf /etc/nginx/conf.d
CMD ["nginx","-g","daemon off;"]
```

Kuvio 28. Docker multistage rakennus

Nginx-välityspalvelimella on mahdollista jakaa staattisia HTML-sivuja, joten sitä voidaan hyödyntää myös WWW-palvelimena. Verkkopalvelun Vue-käyttöliittymä voidaan jakaa käyttäjän selaimeen suoraan API Proxyn kautta, kun käyttäjä navigoi selaimella verkkopalvelun juuriosoitteeseen. Tämän jälkeen kaikki käyttöliittymästä tulevat rajapintakutsut ohjataan API Proxyyn määritettyihin polkuihin. (Ks. kuvio 29.)



Kuvio 29. API Proxy ja frontend

Nginx:n konfiguroitiin kuuntelemaan paikallista porttia 8080, koska palvelinympäristössä on erillinen kuormanjakaja, joka hoitaa verkkoliikenteen salauksen ja terminoinnin. Kuormanjakaja ohjaa verkkoliikenteen paikalliseen porttiin 8080. (Ks. kuvio 30.)

```
server{

    listen 127.0.0.1:8080;

    location / {
        root /var/www/html;
        try_files $uri /index.html;
    }

    location /api {
        proxy_pass http://backend:3000;
    }

    location /auth {
        proxy_pass http://auth-api:3001;
    }

    location /widget {
        proxy_pass http://widget:3002/;
    }
}
```

Kuvio 30. Nginx konfiguraatio

Rajapintapolkujen määritys aloitetaan *location*-direktiivillä, jonka jälkeen voidaan määrittää haluttu polku kuten esimerkiksi ”/api”. Mikäli polun halutaan osoittavan HTML-sivulle, *root*-direktiivillä kerrotaan mistä kansiosta haluttu tiedosto löytyy ja *try_files*- direktiivillä ilmaistaan mihin HTML-tiedostoon polun halutaan osoittavan. Mikäli polku halutaan osoittaa toiseen verkko-osoitteeseen, voidaan se määrittää *proxy*_pass-direktiivillä, jota seuraa verkko-osoite. Määritetyt polut ovat käytettävissä palvelimen verkko-osoitteen perässä, esimerkiksi ”www.palvelu.com/api”. (Ks. kuvio 30.)

Mikäli Nginx suoritetaan Docker-verkkoon yhdistetystä Docker-kontista, voidaan rajapintapolun verkko-osoitteena käyttää konttien nimiä ja portteja kuten esimerkiksi ”http://auth-api:3001” (ks. kuvio 30). Näin Nginx:lle tulevia rajapintapyyntöjä voidaan ohjata toisessa Docker-kontissa sijaitsevaan rajapintaan. Palvelussa API Proxynä käytetty Nginx konfiguroitiin kuvan 30 mukaisesti ohjaamaan liikennettä mikropalveluille.

6.4 Käyttöliittymä

6.4.1 Vue-sovellus

Käyttöliittymän Vue-sovellus luotiin hyödyntämällä Vue-CLI ohjelmaa, jonka konsolikäyttöliittymän avulla voitiin valita Vue-sovelluksen versio sekä asennettavat laajennukset Vuex ja Vue Router. Vue-CLI luo kansiorakenteen Vue-sovellukselle huomioiden valitut laajennukset ja samalla luo siitä Git-arkiston ja lisäksi Vue-CLI huomioi valitut laajennukset kansiohierarkiaa luodessa. (Ks. Creating a Project 2022.)

Router-kansioon toteutettiin kaikki Vue Routerin käyttämät näkymäpolut. Mikäli näkymä vaatii tiettyjä käyttäjäoikeuksia, *beforeEnter*-navigaatiovahdilla suoritettu *isAuthorized*-funktio tarkistaa mikropalvelulta onko käyttäjällä oikeudet näkymään (ks. kuvio 31).

Vue Router-näkymäpolut

```
import store from '../store'
Vue.use(VueRouter)

import { AUTH } from '../store/constants'
function isAuthorized(to, from, next) {
  store.dispatch(AUTH, to.name).then((res) => {
    if (res) {
      next()
    } else {
      next({ name: 'Login' })
    }
  })
}

const routes = [
  {
    path: '/login',
    name: 'Login',
    component: require('@/views/login.vue'),
  },
  {
    path: '/register',
    name: 'Register',
    component: require('@/views/register.vue'),
  },
  {
    path: '/home',
    name: 'Home',
    component: require('@/views/home.vue'),
    beforeEnter: isAuthorized,
  },
```

Polkujen kutsuminen HTML-elementistä

```
b-nav-item.hover-scale(
  v-if='!isLogged',
  :to='{ path: "/login" }',
  exact,
  exact-active-class='active'
) Sign in
b-nav-item.hover-scale(
  v-if='!isLogged',
  :to='{ path: "/register" }',
  exact,
  exact-active-class='active'
) Register
```

Kuvio 31. Vue Router polut

Store-kansio pitää sisällään Vuex-tilanhallintafunktiota ja muuttujia kuten rajapintakutsut ja käyttäjäkohtaisia sovellusasetuksia. *Store*-kansioon määriteltiin Vuex-action funktioita jokaiselle rajapintapyynnölle. Funktioille määritettiin ominaisuusnimet (computed property name), jotta kaikki funktiot voitiin dokumentoida kootusti. Ominaisuusnimen perustella funktioita voitiin kutsua Vuex-moduulista käyttöliittymän Vue-komponenteissa *this.$store.dispatch* -funktiolla. Ominaisuusnimet dokumentoitiin hyödyntäen JSDoc-merkintäkieltä ja niistä generoitiin html-pohjainen dokumentti. (Ks. kuvio 32.)

Alias

```
/**
 * @description Returns messageboard messages
 * @host GET /api/message/get
 */
export const GET_MSG = 'GET_MSG'
```

Action

```
[GET_MSG]: async ({ commit }) => {
  try {
    var response = await ApiQuery('GET', '/message/get')
    if (response.ok) {
      commit(LOADING_DONE)
      return await response.json()
    } else {
      return false
    }
  } catch (error) {
    console.log(error)
    return false
  }
},
```

Funktion kutsuminen komponentissa

```
getMsg() {
  this.$store.dispatch(GET_MSG).then((result) => {
    this.allMessages = result
  })
},
```

Kuvio 32. Vuex-action funktio ja alias dokumentointi

Rajapinta-funktioiden tueksi luotiin *ApiQuery*-funktio, jonka parametreinä voitiin syöttää http-metodi, rajapintapolku, sisällön tyyppi, headereita sekä body-tietoja. *ApiQuery*-funktion tarpeellisuus tunnistettiin kehitysvaiheessa, kun huomattiin että useat rajapintakutsut käyttävät samaa tai hyvin samanlaisia kutsurakenteita. *ApiQuery*-funktio suorittaa ennen rajapintakutsua JWT-tokenien tarkistuksen ja tarvittaessa noutaa mikropalvelulta uuden tokenin. (Ks. kuvio 33.)

```
const ApiQuery = async (
  Method,
  Path,
  Headers = {},
  Body = {},
  ContentType = 'application/json'
) => {
  var default_headers = {
    'Content-Type': ContentType,
  }
  var keys = Object.keys(Headers)

  if (keys.length > 0) {
    keys.forEach((key) => {
      default_headers[key] = Headers[key]
    })
  }

  var query = {
    method: Method,
    credentials: 'include',
    headers: default_headers,
    body: JSON.stringify(Body),
  }
  if (['GET', 'HEAD'].includes(Method.toUpperCase())) delete query.body
  if (!tokenStatus()) {
    if (await fetchAccessToken()) {
      return await fetch(URI + Path, query)
    } else {
      return false
    }
  } else {
    return await fetch(URI + Path, query)
  }
}
```

Kuvio 33. ApiQuery-apufunktio

6.4.2 Visuaalinen rakenne

Käyttöliittymä toteutettiin yksinkertaisena SPA (Single Page Application) -rakenteena ilman monimutkaisia valikkoja. Asiakas yrityksen logo on näkyvissä *Bannerissa* ja se ohjaa asiakkaan julkisille verkkosivuille. *Navbar*-navigaatiopalkista käyttäjä voi navigoida eri näkymiin. Navigaatiopalkin polut riippuvat kirjautuneen käyttäjän oikeuksista ja toiminnot tulevat käytettäväksi, kun käyttäjä kirjautuu palveluun. Navigaatiopalkin osoittimen kohdalla oleviin elementteihin lisättiin interaktiivisuutta animoimalla suurentuva ja pienentyvä teksti. *Route-View* on Vue Router -lisäosan komponentti, joka vastaa navigaatiopalkista valitun näkymän näyttämisestä. (Ks. kuvio 34.)

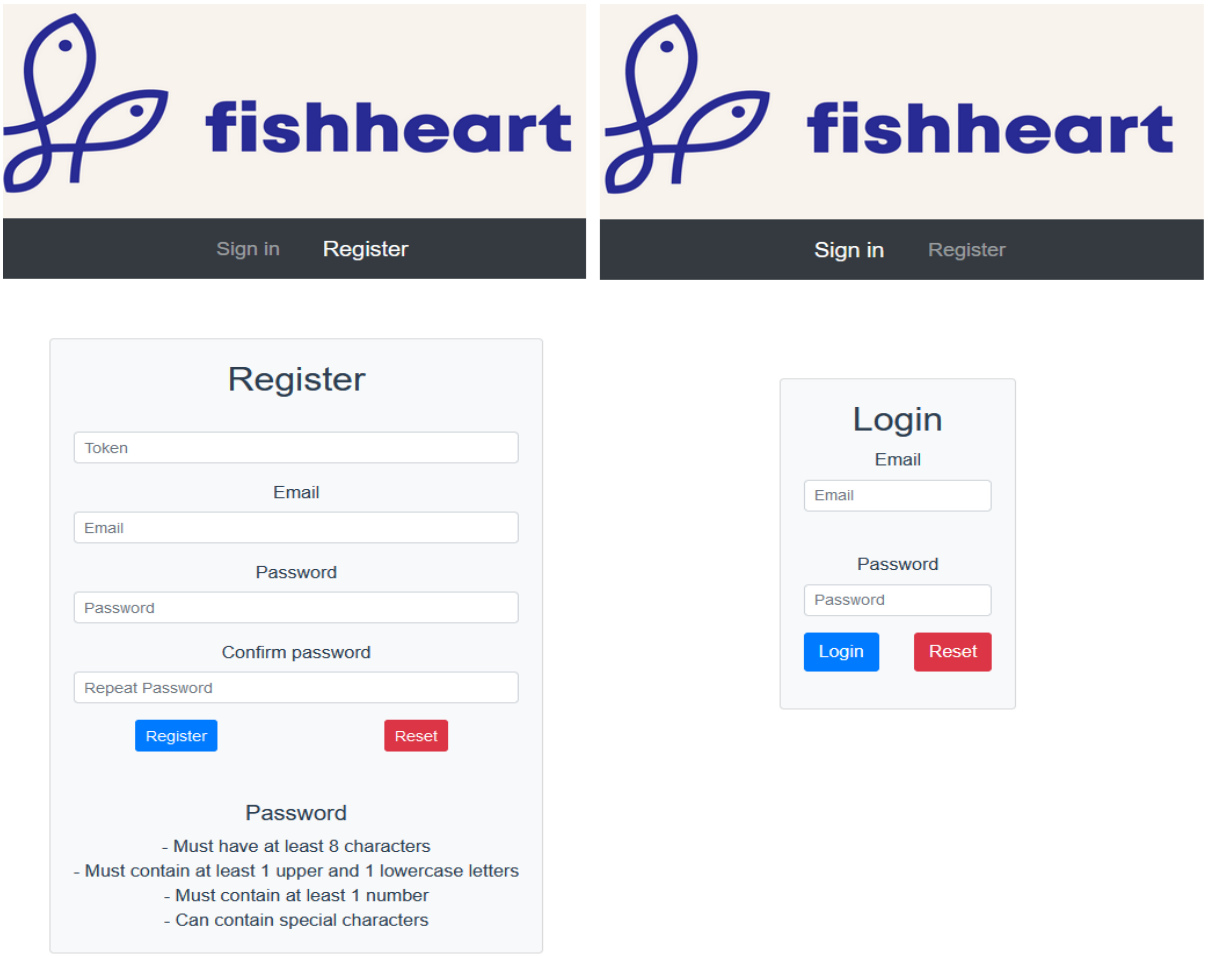
Kuvio 34. Käyttöliittymän rakenne

6.4.3 Näkymät

Kirjautumis- ja rekisteröitymisnäkymät

Koska palvelu ei ole julkisesti käytettävissä niin kirjautumisnäkymä toimii palvelun etusivuna, kun käyttäjä ei ole kirjautunut. Kirjautumislomakkeessa on syöttökentät sähköpostille ja salasanalle sekä *Login*-painike, joka lähettää kenttien tiedot pyyntönä Fishheart-Auth mikropalvelulle. (Ks. kuvio 35.)

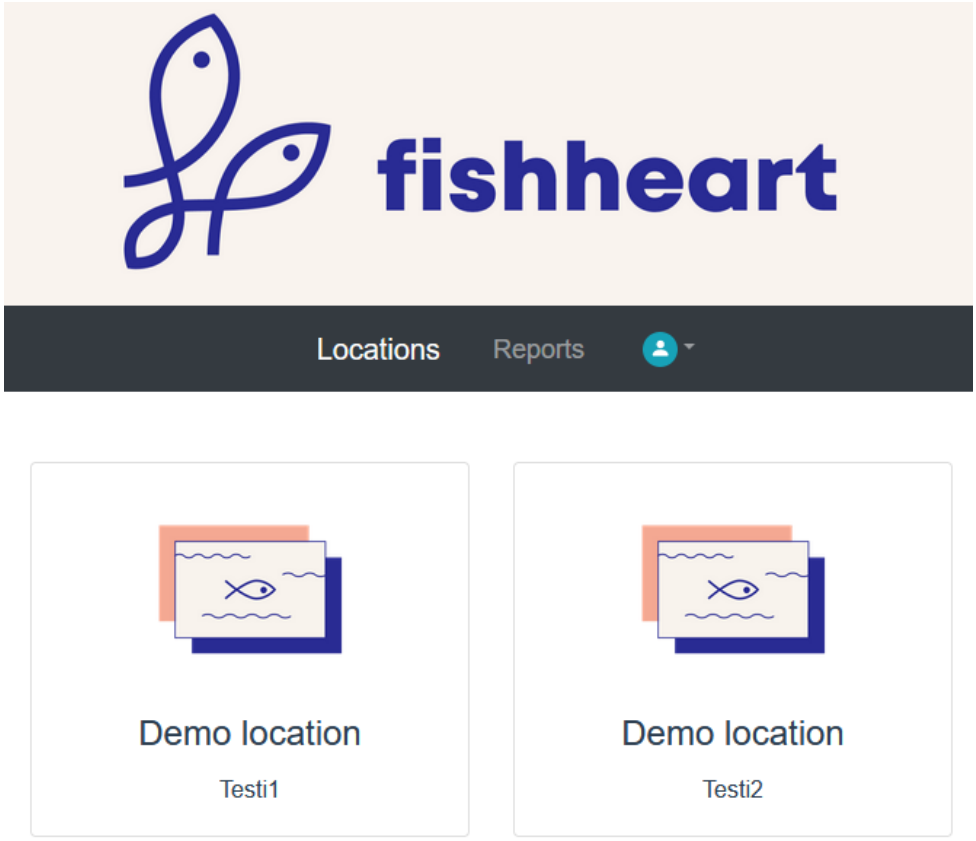
Rekisteröitymislomake vaatii rekisteröitymisavaimen, jonka vain palvelun pääkäyttäjät voivat luoda. Avain on kertakäyttöinen ja voimassa rajoitetun ajan. Käyttöliittymä suorittaa taustalla rekisteröintilomakkeen sähköpostiosoitteen ja salasanan alku varmennuksen ja antaa virhe ilmoituksen, mikäli kentät eivät ole vaaditun mukaisia. Fishheart-Auth mikropalvelu varmistaa vielä kutsun yhteydessä kenttien arvot ja virhetilanteessa palauttaa viestin takaisin käyttöliittymään. *Reset*-painikkeet kummassakin näkymässä palauttaa kyselyn kentät alkuarvoon. (Ks. kuvio 35.)



Kuvio 35. Rekisteröitymis- ja kirjautumisnäkymät

Kohdenäkymä

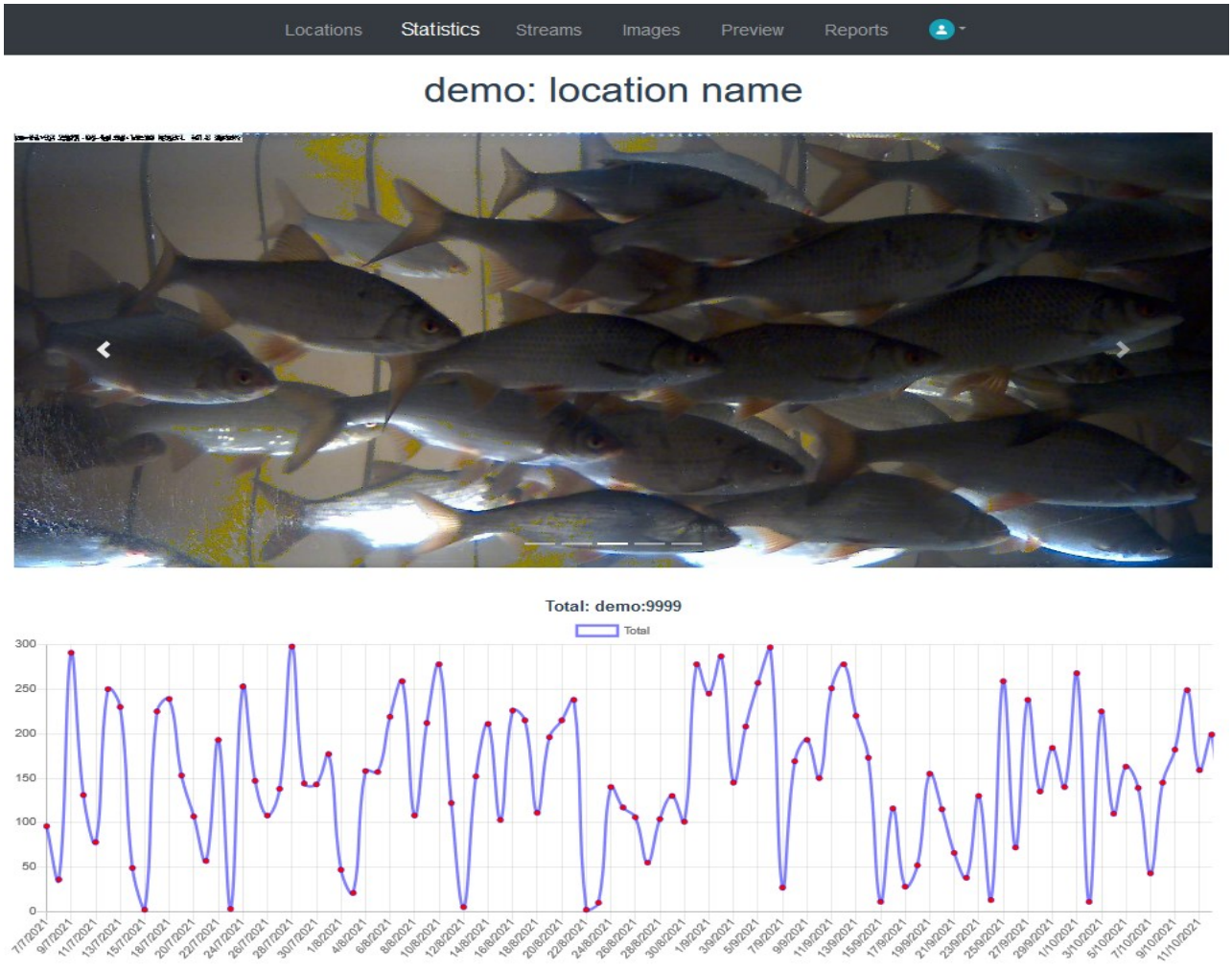
Kun käyttäjä on kirjautunut onnistuneesti palveluun, avautuu kohdenäkymä. Kohdenäkymästä käyttäjä voi nähdä kaikki kohteet, joihin käyttäjälle on myönnetty oikeudet. Kohde vastaa yhtä tai useampaa Fishheart-kalatiejärjestelmää ja käyttäjä voi valita minkä järjestelmän tunnistustietoja haluaa tarkastella. Kun käyttäjä on valinnut kohteen, käyttäjä ohjataan kohteen statistiikkanäkymään. (Ks. kuvio 36.)



Kuvio 36. Kohdenäkymä

Statistiikkanäkymä

Statistiikkanäkymässä käyttäjälle näytetään valitun kohteen lajitunnisteiden statistiikkaa viivakaaviona ja viisi viimeisintä validoitua kuvaa, jotka ovat merkitty esittelykuvaksi operaatoreiden toimesta. Operaattorit voivat palvelun asetuksista muokata mitkä tunniste-etiketit lasketaan statistiikkaan mukaan ja lisäksi statistiikalle voidaan määrittää aikaväli. Käyttäjän siirtyessä statistiikkanäkymään, käyttöliittymä automaattisesti lähettää Fishheart-API mikropalvelulle pyynnön uusimmasta kohteen statistiikasta. Mikropalvelun vastaus ohjataan taulukkokomponenttiin, joka parsii vastauksesta aikavälin statistiikan ja piirtää siitä viivakaavion. Viivakaavion toteutuksessa hyödynnettiin *vue-chartjs*-kirjastoa. (Ks. kuvio 37.)

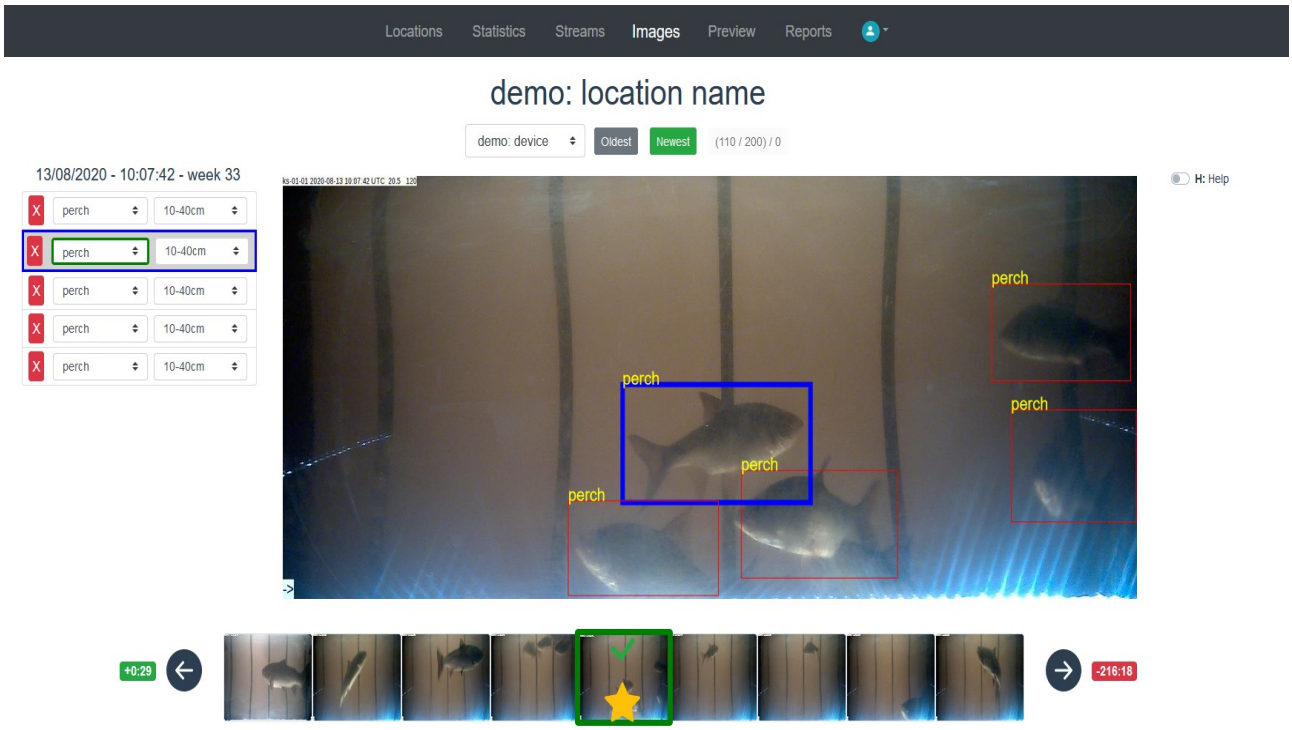


Kuvio 37. Statistiikkanäkymä

Validointinäkymä

Kuvien validointinäkymä on yksi tärkeimmistä palveluun toteutetuista työkaluista. Validointinäkymässä käyttäjät pystyvät selaamaan ja validoimaan kuvantunnistusrajapinnasta ladattuja kuvia ja niiden metadataa. Validointinäkymässä valittuna olevan kuvan metadatasta parsitaan tunnistettujen kohteiden koordinaatit, tunniste-etiketit ja kellonaika. Koordinaattien perustella kuvan päälle piirretään rajoituslaatikot (bounding-box) jotta tunnistetut kohteet voidaan varmentaa. Lisäksi metadatan sisältämät tunniste-etiketit ja kellonajat lisätään näkymän vasempaan laitaan listana. Listasta tunnisteita voidaan valita aktiiviseksi, jolloin rajoituslaatikko muuttuu väriltään siniseksi indikoiden valittua kohdetta. Lisäksi listasta voidaan poistaa tunnisteita tai muokata tunniste-etikettejä. (Ks. kuvio 38.)

Käyttäjä pystyy piirtämään hiirellä rajoituslaatikoita kuvaan, mikäli kuvantunnistus ei ole tunnistanut haluttua kohdetta. Kuvaan piirrettyjen rajoituslaatikoiden xywh-koordinaatit skaalataan suhteessa alkuperäiseen kuvakokoon, jotta koordinaatti ovat oikein riippumatta selaimen koosta. Kun käyttäjä validoi kuvan, jokaisesta rajoituslaatikosta tallennetaan metatiedot tietokantaan, joka pitää sisällään koordinaatit, tunniste-etiketti, päivämäärä, kellonaika, tiedostonimi, kohde ja laite tietoja. (Ks. kuvio 38.)



Kuvio 38. Käyttöliittymän validointinäkymä

Enter-näppäimellä kuva voidaan validoida, jolloin tunnistettujen kohteiden metadata tallennetaan ja siirrytään automaattisesti seuraavaan kuvaan. Mikäli kuva ei pidä sisällään järkeviä tunnisteita tai kuva on huonolaatuinen, *Välilyönti*-näppäimellä kuva voidaan validoida mutta sen sisältö merkitään hylättäväksi statistiikasta. Pääkäyttäjät voivat *Q*-näppäimellä asettaa kuvan esittelykuvaksi, jolloin se on nähtävissä statistiikkanäkymän kuvakarusellissa. (Ks. kuvio 38.)

Kuvia on mahdollista selata eteen- ja taaksepäin vasemmalla ja oikealla nuolinäppäimellä. Mikäli kuvassa on useampi tunnistettu kohde, ylös ja alan nuolinäppäimillä voidaan valita tunniste aktii-

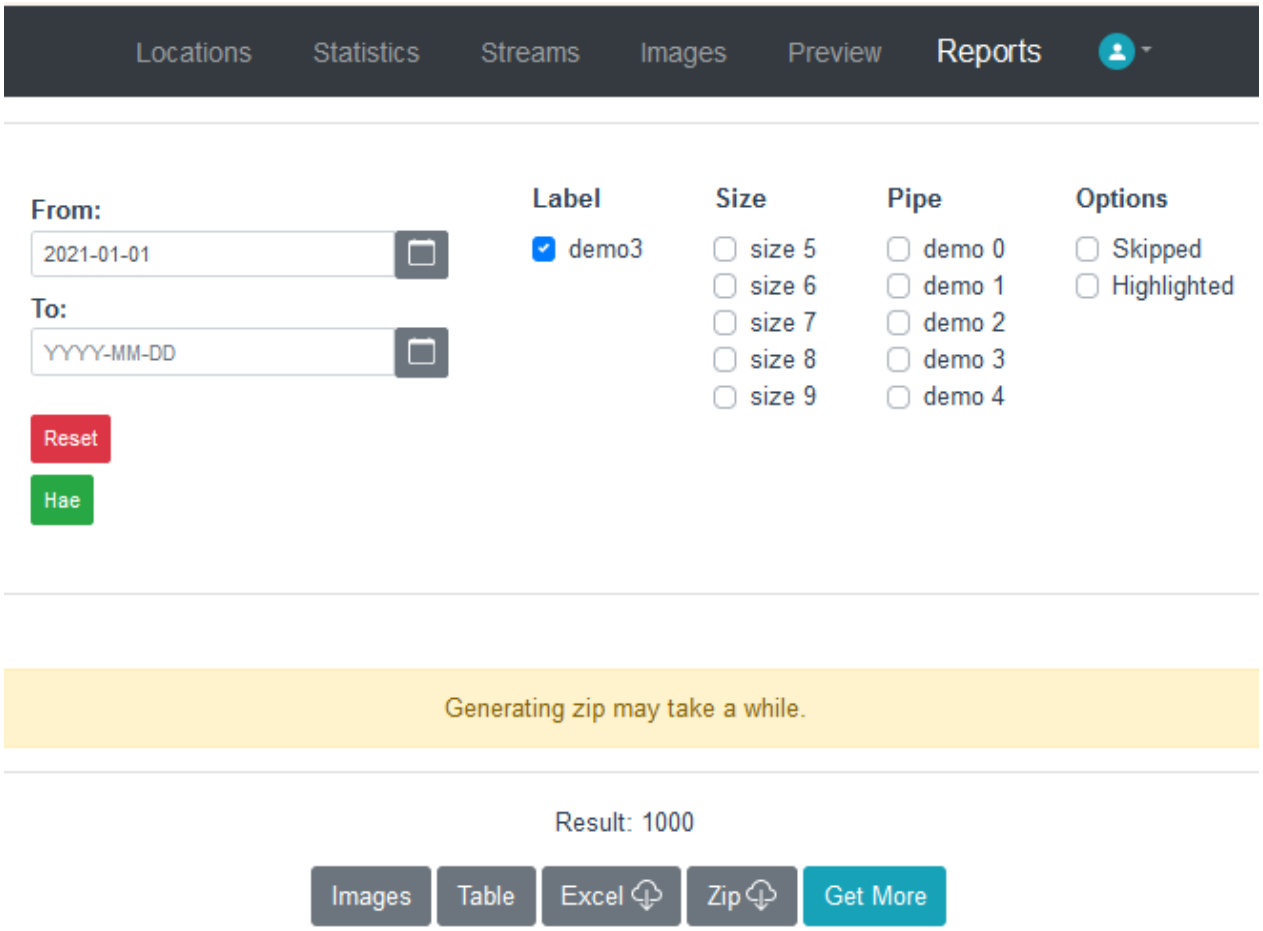
viseksi. Kohdistettuun tunnisteeseen voidaan vaihtaa tunnistusluokkia numeronäppäimillä. Näkymän alaosassa on esinäkymä, jossa nykyinen kuva on reunustettu virheällä indikoiden, että kuva on valittuna. Lisäksi esinäkymästä nähdään maksimissaan neljä edellistä ja seuraavaa kuvaa sekä nykyisen kuvan aikaero edelliseen ja seuraavaan kuvaan. Kun käyttäjä on validoinut tai asettanut kuvan esittelykuvaksi, esinäkymäkuvan päälle lisätään vihreä hyväksyttymerkki ilmoittamaan, että kyseinen kuva on validoitu. Keltainen tähti lisätään kuvan päälle, mikäli kuva on asetettu esittelykuvaksi. (Ks. kuvio 38.)

Näkymän yläosan pudotusvalikosta käyttäjä voi valita kohteessa olevan laitteen, jonka kuvia haluaa tarkastella. Lisäksi yläosassa olevista *Oldest* ja *Newest*-painikkeista käyttäjä voi halutessaan järjestää kuvia, lähtökohtaisesti kuvat järjestetään vanhimmasta uusimpaan. Validointinäkymään noudetaan rajapinnasta maksimissaan 200 kuvaa kerrallaan ja kun kuvat ovat käsitelty, noudetaan seuraava eräkuvia. (Ks. kuvio 38.)

Raporttinäkymä

Raporttinäkymässä käyttäjät voivat hakea tietokannasta tunnistettuja kohteita hakuparametreillä: päivämäärä, etiketti, koko, laite id tai onko tunnistettu kohde hylätty tai nostettu erityisen hyväksi. Mikäli hakutuloksia löytyy yli tuhat kappaletta, jaetaan hakutulokset tuhannen kappaleen eriin. *Images*-painikkeella hakutuloksia voidaan tarkastella, painike aukaisee hakutulokset listana peukalokuvakkeita sivun alaosaan. Yksittäistä peukalokuvaketta painamalla, sivulle avautuu muokkausikkuna, josta käyttäjä pystyy tarkastelemaan kuvan metatietoa ja lisäksi tunniste-etikettejä on mahdollista muokata. *Table*-painikkeella sivun alaosaan avautuu lista hakutuloksista taulukkomuodossa. (Ks. kuvio 39.)

Excel-painikkeesta hakutulokset muutetaan taulukoksi ja tallennetaan Excel-tiedostoon, jonka jälkeen käyttäjälle aukeaa selaimeen Excel-tiedoston lataus mahdollisuus. *Zip*-painikkeesta hakutuloksen kuvat muutetaan jpeg-formaattiin ja paketoidaan zip-tiedostoon, jonka jälkeen käyttäjälle aukeaa selaimeen mahdollisuus ladata zip-tiedosto. Mikäli hakutuloksia on yli tuhat, *Get More*-painike tulee näkyviin ja sitä painettaessa noudetaan seuraava erä hakutuloksia näytettäväksi sivulle. (Ks. kuvio 39.)



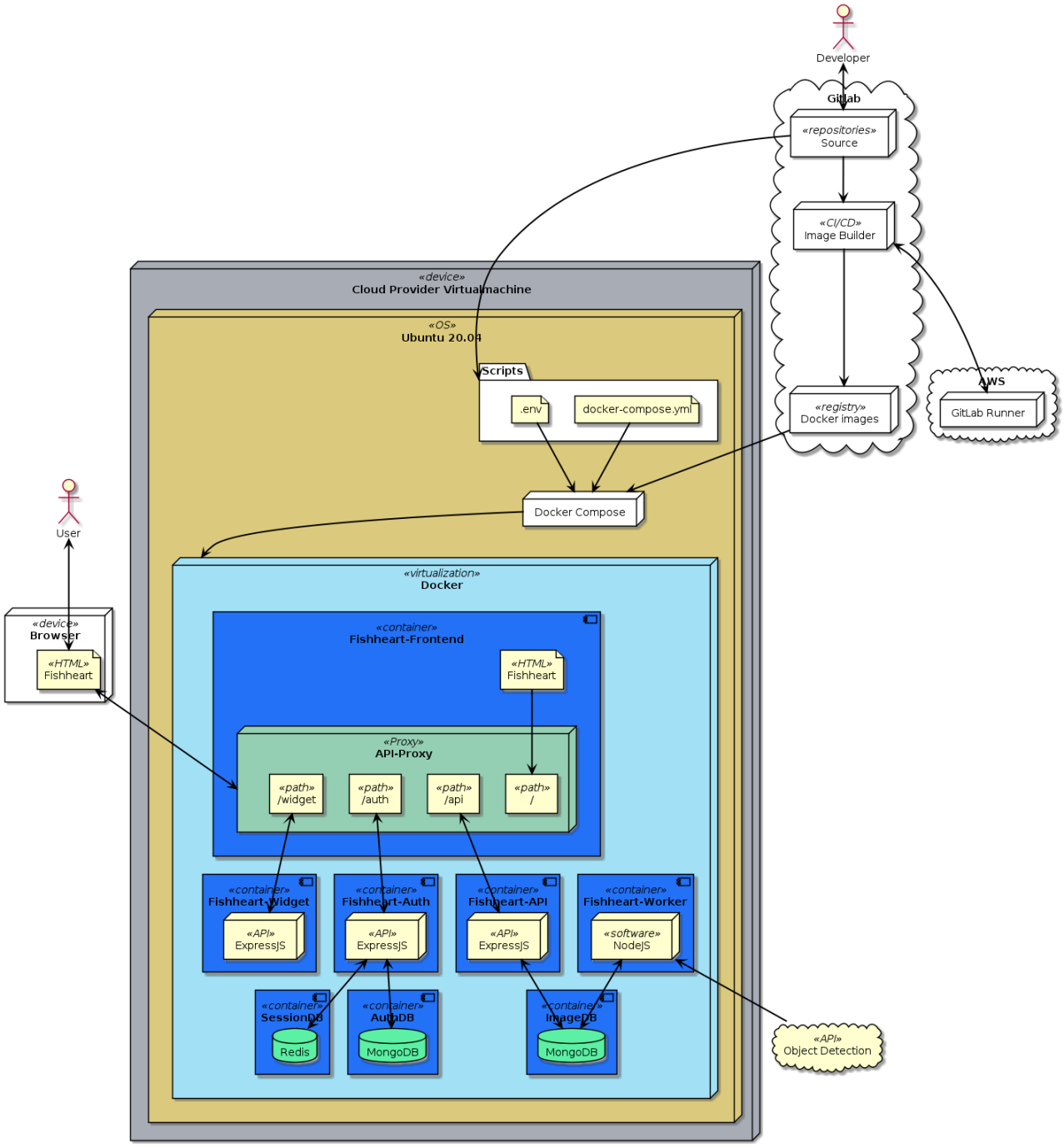
Kuvio 39. Raporttinäkymä

6.5 Julkaisu

Palvelun julkaisuympäristönä käytettiin toimeksiantajan virtuaalikonetta Ubuntu-käyttöjärjestelmällä, johon asennettiin Docker ja Docker-Compose. Käynnistettävät palvelut määriteltiin Docker-Compose konfiguraatiotiedostossa (ks. kuvio 40). Tietokantoihin tarvittavat kirjautumistiedot ja mikropalveluiden sessioavaimet luotiin palvelinympäristöön *.env*-tiedostoon, josta Docker-Compose lukee ja välittää ympäristömuuttujat käynnistettäville Docker-konteille. (Ks. kuvio 41.)

```
version: '3.7'
services:
  database:
    container_name: mongo
    restart: unless-stopped
    image: mongo
    environment:
      - MONGO_INITDB_DATABASE= ${MONGO_DB}
      - MONGO_INITDB_ROOT_PASSWORD= ${MONGO_INITDB_ROOT_PASSWORD}
      - MONGO_INITDB_ROOT_USERNAME= ${MONGO_INITDB_ROOT_USERNAME}
    volumes:
      - mongo_data:/data/db
      - mongo_config:/data/configdb
    networks:
      - app-network
    ports:
      - '127.0.0.1:27017:27017'

  server:
    container_name: api
    restart: unless-stopped
    image: registry.gitlab.com/kovakoodarit/fishhear-backend:main
    environment:
      - REDIS_URI= ${REDIS_URI}
      - SESSION_SECRET= ${SESSION_SECRET}
      - PORT= ${PORT}
      - MONGO_URI= ${MONGO_URI}
      - DEVTOKEN= ${DEVTOKEN}
    ports:
      - '127.0.0.1:3000:3000'
    networks:
      - app-network
    depends_on:
      - database

  worker:
    container_name: worker
    restart: unless-stopped
    image: registry.gitlab.com/kovakoodarit/fishheart-worker:main
    environment:
      - API_URL=${API_URL}
      - API_USR=${API_USR}
      - API_PW=${API_PW}
      - MONGO_URI=${MONGO_URI}
      - WORKER_INIT_DATE=${WORKER_INIT_DATE}
    networks:
      - app-network
    depends_on:
      - database

volumes:
  mongo_data: {}
  mongo_config: {}

networks:
  app-network:
    driver: bridge
```

Kuvio 40. Docker-compose esimerkki



Kuvio 41. Palvelun kokonaiskuva

7 Pohdinta

Opinnäytetyön kirjoitusvaiheessa haasteena oli aihealueiden ja syventymisen rajaus koska opinnäytetyön aiheeseen liittyi useita laajoja käsitteitä ja teknologioita. Tästä syystä esimerkiksi käyttöliittymän komponenttien teknologioista toteutusta ei lähdetty syvällisesti kertomaan, jotta tasapaino työssä säilyisi.

Opinnäytetyön toteutus sujui ilman suurempia ongelmia ja mikropalveluarkkitehtuuri osoittautui erittäin hyväksi valinnaksi palvelun ja kehitystyön näkökulmista. Mikropalveluissa käytetyn Express-pohjan toteutus osoittautui myös erittäin toimivaksi ja Gitlab integraatiot sujuivat ongelmitta. Suurin tekninen haaste oli validointityökalun piirto-ominaisuuden toteuttaminen, jonka avulla rajauslaatikoita voitiin piirtää kuvien päälle. Haasteellisen siitä teki canvas-elementin toteuttaminen responsiivisena sekä kuvasuhteen skaalaus alkuperäiseen kuvakokoon, jotta piirretyn tunnistuslaatikon koordinaatit ovat oikein.

Jatkokehityksen näkökulmasta mikropalveluarkkitehtuuri mahdollistaa monipuoliset vaihtoehdot kehitykselle. Mikäli palvelua tullaan laajentamaan merkittävästi usean palvelimen konfiguraatioon, on Kubernetes tai Docker Swarm -orkestrointityökalun käyttämistä syytä harkita skaalauksen maksimoimiseksi. Lisäksi palveluun on siinä tapauksessa syytä implementoida dynaaminen rajapintarekisteri ja palvelurekisteröinti, jotta skaalaus ja reititys mikropalveluille saadaan järkevästi toteutettua.

Toteutettujen mikropalveluiden osalta Fishheart-API:n olisi voitu pilkkoa pienempiin mikropalveluihin omine tietokantoineen. Tämä olisi tukenut mikropalveluarkkitehtuurin ideologiaa ja vahvistanut palvelun skaalausta hienojakoisemmaksi mutta toteutus hetkellä tätä ei nähty tarpeelliseksi tiukan projektiaikataulun puitteissa.

Käyttöliittymän osalta voisi olla hyödyllistä erottaa palvelun asiakkaille suunnattu statistiikkatoiminnallisuus erilliseksi sovellukseksi, jonka visuaalisesta ilmeestä voisi tehdä esteettisemmän. Järjestelmän operaattoreille tarkoitetut työkalut voisivat olla ilmeeltään toteutuksen mukaisia. Edellä mainitut asiat toisivat palveluun selkeän rajan henkilökunnan ja asiakkaiden käyttämien palveluiden välille.

Opinnäytetyön tuloksena saatiin nykyaikaisia menetelmiä ja teknologioita hyödyntävä verkkopalvelu, joka vastasi asiakkaan palvelulle asettamiin vaatimuksiin. Kalasydän Oy:n otti palvelun käyttöön osana Fishheart-järjestelmää jo kehitysvaiheessa, kun tärkeimmät toiminnallisuudet saatiin valmiiksi. Järjestelmä operaattoreilta saatiin hyvää palautetta työkalun toiminnasta ja sen tuomista hyödyistä. Opinnäytetyön toteutusta voidaan pitää kaikin puolin onnistuneena ja työlle asetettuihin tavoitteisiin päästiin.

Lähteet

About GitLab. N.d. Gitlab. Viitattu 3.5.2022. https://about.gitlab.com/company/

Azure DevOps, N.d. Identify microservice boundaries. Viitattu 5.4.2022. https://docs.microsoft.com/en-us/azure/architecture/microservices/model/microservice-boundaries

BootstrapVue. N.d. Buttons. Viitattu 1.3.2022. https://bootstrap-vue.org/docs/components/button

Buchanan, I. N.d. Containers vs. virtual machines. Viitattu 25.4.2022. Artikkeli konttien ja virtuaalikoneiden eroista. https://www.atlassian.com/microservices/cloud-computing/containers-vs-vms

Chacon, S. & Straub, B. 2022. Pro Git. Apress. Viitattu 10.3.2022. https://git-scm.com/book/en/v2

Choudhary, M. 2019. Introduction to Web Development Fundamentals. Blogi-kirjoitus verkkokehityksen perusteista.Viitattu 4.3.2022. https://medium.com/swlh/introduction-to-web-development-fundamentals-b89f5c91ed3f

Containerization. 2021. IBM Cloud Education. Viitattu 6.3.2022. https://www.ibm.com/cloud/learn/containerization

Creating a Project. 2022. Vue CLI. Viitattu 24.4.2022. https://cli.vuejs.org/guide/creating-a-project.html#using-the-gui

DevOps Tools. N.d. Atlassian. Viitattu 1.5.2022. https://www.atlassian.com/devops/devops-tools

Docker Architecture. N.d. Aqua. Viitattu 1.5.2022. https://www.aquasec.com/cloud-native-academy/docker-container/docker-architecture/

Docker. 2021. IBM Cloud Education. Viitattu 4.3.2022. https://www.ibm.com/cloud/learn/docker

Docker-compose cheatsheet. N.d. Devhints. Viitattu 25.4.2022. https://devhints.io/docker-compose

Dockerfile reference. N.d. Docker. Viitattu 24.4.2022. https://docs.docker.com/engine/reference/builder/

Doglio, F. 2018. REST API Development with Node.js Manage and Understand the Full Capabilities of Successful REST Development. 2.p. Apress.

Express/Node introduction. 2022. Mdn web docs. Viitattu 9.3.2022. https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction

Filipova, O & Vilãö, R. 2018. Software Development From A to Z: A Deep Dive into all the Roles Involved in the Creation of Software. Apress. Viitattu 5.4.2022.

Gitlab. N.d. Gitlab. Viitattu 13.3.2022. https://about.gitlab.com/

Gorton, I. 2011. Essential Software Architecture. 2. p. Springer.

Huffine, T. 2020. The 2021 Web Developer Roadmap. Viitattu 24.4.2022. https://levelup.gitconnected.com/the-2020-web-developer-roadmap-76503ddfb327

Introduction to JSON Web Tokens. N.d. JWT. Viitattu 28.4.2022. https://jwt.io/

Introduction to Node.js. N.d. NodeJS. Viitattu 2.3.2022. https://nodejs.dev/learn

Jarrod, 2016. Linux Web Server Performance Benchmark – 2016 Results. Viitattu 2.4.2022. https://www.rootusers.com/linux-web-server-performance-benchmark-2016-results/

Karnik, N. 2018. Introfuction to Mongoose for MongoDB. Viitattu 2.3.2022. https://www.freecodecamp.org/news/introduction-to-mongoose-for-mongodb-d2a7aa593c57/

KovaKoodarit. N.d. Kovakoodarit verkkosivu. Yrityksen verkkosivu. Viitattu 13.4.2022.

https://www.kovakoodarit.com/

Lotanna, N. 2020. How the virtual DOM works in Vue.js. Blogi-kirjoitus Vuejs DOM toiminnasta. Viitattu 9.3.2022. https://blog.logrocket.com/how-the-virtual-dom-works-in-vue-js/

Luukkainen, M. 2021. Ohjelmistotuotanto 2021. Github. Viitattu 20.3.2022. https://ohjelmistotuotanto-hy.github.io/osa2/

Microservices. 2021. IBM Cloud Education. Viitattu 12.3.2022. https://www.ibm.com/cloud/learn/microservices

Networking overview. N.d. Docker Docs. Viitattu 1.5.2022. https://docs.docker.com/network/

NGINX. N.d. What is NGINX?. Viitattu 2.4.2022. https://www.nginx.com/resources/glossary/nginx/

Overview of Docker Compose. N.d. Docker docs. Viitattu 13.3.2022. https://docs.docker.com/compose/

Peabody, B. N.d. Server-side I/O Performance: Node vs. PHP vs. Java vs. Go. Viitattu 1.5.2022. https://www.toptal.com/back-end/server-side-io-performance-node-php-java-go

Remote – WSL. N.d. Visual Studio Marketplace. Viitattu 25.4.2022. https://marketplace.visualstudio.com/items?itemName=ms-vscode-remote.remote-wsl

Richardson, C. N.d. Pattern: Monolithic Architecture. Viitattu 4.4.2022. https://microservices.io/patterns/monolithic.html

StateOfJS. 2020. Front-end Frameworks. Viitattu 28.02.2022. https://2020.stateofjs.com/en-US/technologies/front-end-frameworks/

Taylor, D. 2022. What is MongoDB?. Viitattu 2.3.2022. https://www.guru99.com/what-is-mongodb.html

Use Volumes. N.d. Docker Docs. Viitattu 1.5.2022. https://docs.docker.com/storage/volumes/

Vue Router. N.d. Vue Router. Viitattu 24.4.2022. https://router.vuejs.org/guide/

Vue.js. N.d. What is Vue?. Viitattu 28.02.2022. https://vuejs.org/guide/introduction.html

What is DevOps. N.d AWS. Viitattu 24.4.2022 https://aws.amazon.com/devops/what-is-devops/

What is Node.js?. N.d. Tutorialpoint. Viitattu 13.4.2022. https://www.tutorialspoint.com/nodejs/nodejs_introduction.htm

What is Redis. N.d. AWS. Viitattu 2.3.2022. https://aws.amazon.com/redis/

What is REST?. N.d. Codeacademy. Viitattu 12.4.2022. https://www.codecademy.com/article/what-is-rest

What is serverless computing?. N.d. Cloudflare. Viitattu 24.3.2022. https://www.cloudflare.com/learning/serverless/what-is-serverless/

What is Vuex?. 2022. Vuex. Viitattu 15.4.2022. https://vuex.vuejs.org/

Wilson, B. 2022. What is Docker? How Does it Work?. Viitattu 6.3.2022. https://devopscube.com/what-is-docker/

Wilson, C. 2020. How to Design a Web Application: Software Architecture 101. Viitattu 16.3.2022. https://www.educative.io/blog/how-to-design-a-web-application-software-architecture-101