LAB University of Applied Sciences

Relaatiotietokantapohjainen taustajärjestelmä mobiilisovellukseen

Suunnittelusta toteutukseen

LAB-ammattikorkeakoulu
Insinööri (AMK)
2025
Risto Flink

Tiivistelmä

Tekijä(t)	Julkaisun laji	Valmistumisaika
Risto Flink	Opinnäytetyö, AMK	2025
	Sivumäärä 40	

Työn nimi

Relaatiotietokantapohjainen taustajärjestelmä mobiilisovellukseen

Suunnittelusta toteutukseen

Tutkinto ja koulutusala

Insinööri (AMK), tieto- ja viestintätekniikka

Toimeksiantajaorganisaatio (jos opinnäytetyöllä on toimeksiantaja)

Tiivistelmä

Tämän opinnäytetyön tarkoituksena oli suunnitella tietoturvallinen ja suorituskykyinen taustajärjestelmä tietokantoineen mobiilisovellukselle. Opinnäytetyössä keskityttiin erityisesti taustajärjestelmän arkkitehtuuriin, tietokannan rakenteeseen, ohjelmointirajapintojen suunnitteluun sekä tietoturvallisuuden ja yksityisyydensuojan varmistamiseen. Työssä hyödynnettiin nykyaikaisia teknologioita, kuten NestJS -kehystä ja TypeORM -objekti-relaatiokartoitustyökalua.

Opinnäytetyön lopputuloksena syntyi proof of concept -tason toteutus, joka demonstroi taustajärjestelmän perustoiminallisuutta sekä kykyä kommunikoida saumattomasti erillisen frontend-sovelluksen kanssa. Työ tarjoaa perustan ymmärtää nykyaikaisen mobiilisovelluksen taustajärjestelmän suunnittelun ja toteutuksen keskeisiä periaatteita ja haasteita.

Asiasanat

taustajärjestelmä, relaatiotietokanta, TypeScript, NestJS, objekti-relaatiokartoitus, autentikointi, auktorisointi

Abstract

Author(s)	Type of Publication	Published
Risto Flink	Thesis, UAS	2025
	Number of Pages	
	40	

Title of Publication

A relational database-driven backend for a mobile application

From design to implementation

Degree, Field of Study

Engineer (UAS), Information and Communication Technology

Organisation of the client (if the thesis work is commissioned by another party)

Abstract

The purpose of this thesis was to design a secure and high-performance backend system, including its database, for a mobile application. The thesis focused particularly on the architecture of the backend system, the database structure, the design of application programming interfaces (APIs), and ensuring information security and privacy. Modern technologies were utilized in the work, such as the NestJS framework and the TypeORM object-relational mapping (ORM) tool.

The outcome of the thesis was a proof-of-concept level implementation that demonstrates the basic functionality of the backend system and its ability to communicate seamlessly with a separate frontend application. This work provides a foundation for understanding the key principles and challenges in designing and implementing the backend system of a modern mobile application.

Keywords

backend system, relational database, TypeScript, NestJS, object-relational mapping, authentication, authorization

Sisällys

1 Johdanto 1
2 Teoria 3
2.1 Järjestelmäarkkitehtuuri 3
2.2 Sovelluksen eri komponenttien vuorovaikutus 4
2.3 Taustajärjestelmän ja tietokannan kommunikointi 5
2.4 Teknologiavalinnat 6
2.4.1 Käytettävät ohjelmointikielet- ja kehykset 6
2.4.2 Muut työkalut ja kirjastot 7
2.5 Tietokanta 9
2.5.1 Tietokantataulujen ja relaatioiden kuvaus 9
2.5.2 Tietokannan normalisointi ja optimointi 10
2.5.3 Objekti-relaatiokartoitus 11
2.6 Ohjelmointirajapinnat 11
2.6.1 Ohjelmointirajanpinnan päätepisteet ja niiden toiminnallisuus 12
2.6.2 Erilaiset ohjelmointirajapintatyypit 12
2.6.3 Dokumentaatio 13
2.7 Tietoturva ja yksityisyydensuoja 14
2.7.1 Käyttäjien autentikointi ja valtuutus 14
2.7.2 OAuth2 15
2.7.3 Tietokannan ja API:n suojausmenetelmät 16
2.7.4 Yksityisyydensuojan huomioiminen 17
2.8 Tekoäly 19
2.8.1 Tekoälyn rooli sovelluksessa 19
2.8.2 Tekoälymallien vaihtoehdot 19
3 Toteutus 21
3.1 Tietokannan toteutus 21
3.1.1 Tietokannan luominen ja konfigurointi 22
3.1.2 Tietokannan testaus 22
3.2 API:n toteutus 23
3.2.1 Päätepisteet 23
3.2.2 API:n testaus 24
3.2.3 Virheiden hallinta ja käsittely 26
3.2.4 Suorituskyky 26
3.3 Koodin laatu ja ylläpito 28

3.3.1 Käytänteet 28
3.3.2 Katselmointi 30
3.3.3 Versiohallinta 30
3.4 Käyttöönotto ja julkaisu 31
3.4.1 Palvelinympäristö 31
3.4.2 Käyttöönottoprosessi 32
3.4.3 Ylläpito ja valvonta 33
3.4.4 Skaalautuvuus 33
4 Yhteenveto ja pohdinta 35
4.1 Tulokset ja arvointi 35
4.2 Jatkokehitysmahdollisuudet 35
Lähteet 38

1 Johdanto

Mobiilisovellukset ovat nykypäivänä olennainen osa digitaalista ekosysteemiä, ja niiden merkitys kasvaa jatkuvasti niin yksityiselämässä kuin liiketoiminnassakin. Käyttäjäystävällisten ja monipuolisten mobiilisovellusten suosion taustalla on kuitenkin usein näkymätön, mutta sitäkin tärkeämpi komponentti: vankka, luotettava ja tietoturvallinen taustajärjestelmä. Taustajärjestelmä, eli backend, vastaa sovelluksen datan hallinnasta, toimintalogiikasta ja kommunikoinnista käyttöliittymän (frontend) kanssa. Sen suorituskyky, luotettavuus ja tietoturva vaikuttavat suoraan käyttäjäkokemukseen, vaikka käyttäjä ei sitä suoraan näekään. Hidas tai epäluotettava taustajärjestelmä johtaa väistämättä huonoon käyttäjäkokemukseen mobiilisovelluksessa.

Tämän opinnäytetyön keskiössä onkin tietokantapohjaisen taustajärjestelmän suunnittelu ja toteutus mobiilisovelluksen tarpeisiin. Työssä tutkitaan järjestelmän arkkitehtuurin perusperiaatteita, olennaisia teknologiavalintoja, tietoturvan varmistamista sekä käyttäjien yksityisyydensuojan huomioimista. Nämä ovat kriittisiä osa-alueita, joiden onnistunut toteutus on edellytys nykyaikaisen mobiilisovelluksen toimivuudelle ja luotettavuudelle. Työssä käytettäviksi keskeisiksi teknologioiksi valittiin NestJS-sovelluskehys, TypeScript-ohjelmointikieli, TypeORM-objekti-relaatiokartoituskirjasto ja PostgreSQL-relaatiotietokanta. Näiden teknologioiden valintaa ja niiden soveltuvuutta projektiin perustellaan yksityiskohtaisesti teoriaosuudessa.

Tämä opinnäytetyö rajataan proof of concept (PoC) -tason toteutukseen. PoC-projektin ensisijaisena tavoitteena ei ole luoda täysin valmista ja kaupallistettavaa tuotetta, vaan demonstroida ja testata taustajärjestelmän ydintoiminnallisuuksien teknistä toteutettavuutta. Opinnäytetyön konkreettisena tavoitteena on siis suunnitella ja luoda toimiva prototyyppi taustajärjestelmästä. Järjestelmä hyödyntää edellä mainittuja nykyaikaisia teknologioita. Työssä paneudutaan taustajärjestelmän arkkitehtuurin suunnitteluun ja käytännön toteutukseen keskittyen erityisesti tietokannan rakenteen luomiseen, API-rajapintojen määrittelyyn ja toteuttamiseen sekä käyttäjien tietoturvan ja henkilötietojen suojan varmistamiseen.

Opinnäytetyö jakautuu teoria- ja toteutusosuuksiin. Teoriaosuudessa (luku 2) käsitellään järjestelmäarkkitehtuurin eri malleja, perustellaan tehdyt teknologiavalinnat, syvennytään tietokantojen suunnitteluun ja optimointiin, esitellään API-rajapintojen periaatteita ja tyyppejä sekä käydään läpi tietoturvan ja yksityisyydensuojan keskeisiä näkökohtia. Lisäksi pohditaan lyhyesti tekoälyn mahdollisia rooleja sovelluksessa. Toteutusosuudessa (luku 3) kuvataan, miten teoriaosuudessa esitellyt konseptit on toteutettu käytännössä PoC-projektissa, sisältäen esimerkkejä tietokannan ja API:n toteutuksesta, testauksesta, koodin laadun ylläpidosta sekä käyttöönotosta Docker-konttien avulla. Lopuksi yhteenveto ja pohdinta

(luku 4) kokoavat työn tulokset, arvioivat projektin onnistumista ja esittävät jatkokehitysideoita. Tämä opinnäytetyö tarjoaa perustan ymmärtää nykyaikaisen mobiilisovelluksen taustajärjestelmän suunnittelun ja toteutuksen perusperiaatteita ja haasteita.

2 Teoria

2.1 Järjestelmäarkkitehtuuri

Mobiilisovelluksen toiminnallinen perusta rakentuu sen järjestelmäarkkitehtuurille, jonka toteuttamiseen on useita eri lähestymistapoja. Arkkitehtuurivalinnalla on merkittävä vaikutus sovelluksen ylläpidettävyyteen, skaalautuvuuteen ja suorituskykyyn ja se edustaa kompromissia eri vaatimusten ja rajoitteiden välillä.

Richadrsin (2022, 5-6) mukaan monoliittisessa arkkitehtuurissa (monolithic architecture) sovellus muodostaa yhden, integroidun yksikön. Kaikki komponentit, kuten käyttöliittymä, logiikka ja tiedonhallinta, paketoidaan ja jaellaan yhtenä kokonaisuutena. Tämä malli voi yksinkertaistaa kehitystä ja jakelua projektin alkuvaiheessa. Sovelluksen koon ja monimutkaisuuden kasvaessa se voi kuitenkin aiheuttaa haasteita skaalautuvuudelle ja ylläpidettävyydelle, sillä muutokset yhteen osaan edellyttävät usein koko sovelluksen uudelleenrakentamista ja -jakelua. Monoliittinen arkkitehtuuri voi olla toimiva ratkaisu yksinkertaisempiin sovelluksiin, joiden vaatimukset ovat selkeästi määriteltyjä ja suhteellisen pysyviä. Tiivis kytkentä voi mahdollistaa nopeamman kehityksen ja jakelun erityisesti rajatuissa sovelluksissa, joissa muutostarpeet ovat vähäisiä.

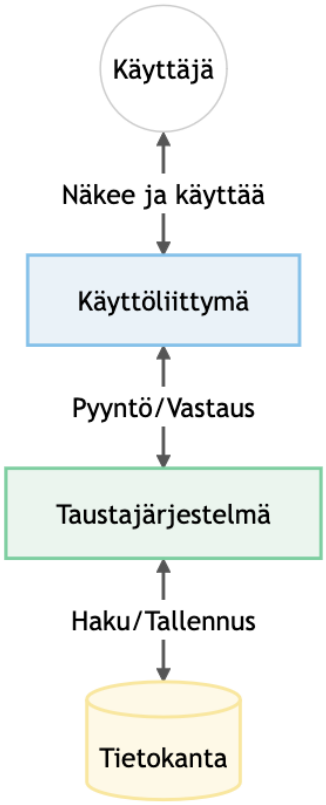
Kerrosarkkitehtuurissa (layered architecture) sovellus jaetaan hierarkkisesti kerroksiin, joilla kullakin on määritelty vastuualue sovelluksen toiminnallisuudesta. Tyypillisiä kerroksia ovat esityskerros (presentation layer), logiikkakerros (business layer) ja tiedonhallintakerros (data layer). Tämä malli edistää modulaarisuutta ja vastuualueiden eriyttämistä (separation of concerns), mikä voi parantaa sovelluksen ylläpidettävyyttä ja skaalautuvuutta. Käyttöliittymä-, logiikka- ja datakerrosten eristäminen mahdollistaa resurssien kohdentamisen tiettyihin komponentteihin ja voi nopeuttaa kehitys- ja testausprosesseja iteratiivisen ohjelmistokehityksen mukaisesti. Huolimattomasti toteutettuna kerrosten välille voi kuitenkin muodostua liian tiivis kytkentä, mikä saattaa vaikeuttaa muutosten tekemistä. Kerrosarkkitehtuuri soveltuu usein laajoihin ja monimutkaisiin sovelluksiin, joihin kohdistuu säännöllisiä päivitystarpeita. (Richards 2022, 15-24.)

Mikropalveluarkkitehtuurissa (microservice architecture) sovellus jaetaan pienempiin, itsenäisiin palveluihin, jotka vastaavat tietyistä toiminnallisuuksista tai liiketoimintakyvykkyyksistä. Palvelut kommunikoivat keskenään standardoitujen ohjelmointirajapintojen (API) kautta, usein käyttäen kevyitä protokollia kuten HTTP/REST tai gRPC, Tämä lähestymistapa lisää joustavuutta ja skaalautuvuutta. (Gavrilenko 2023.) Mikropalveluita voidaan kehittää, jaella ja skaalata itsenäisesti, mikä voi helpottaa sovelluksen päivityksiä ja ylläpitoa suurissa organisaatioissa tai monimutkaisissa järjestelmissä. Teknologiavalinnat voidaan

tehdä palvelukohtaisesti. Toisaalta useiden palveluiden hallinta lisää järjestelmän kokonaiskompleksisuutta. Palveluiden välisen kommunikaation koordinointi, hajautettujen transaktioiden hallinta ja järjestelmän monitorointi voivat aiheuttaa lisätyötä ja vaatia erikoistyökaluja. Mikropalveluarkkitehtuuri voi olla tehokas ratkaisu, kun monimutkaisia sovelluksia kehitetään ja skaalataan useiden tiimien toimesta ja päivitystarpeet ovat säännöllisiä. Sovelluksen jakaminen erillisiin palveluihin mahdollistaa yksittäisten komponenttien itsenäisen kehityksen, jakelun ja skaalauksen, mikä voi nopeuttaa kehityssyklejä laajoissa, jatkuvasti kehittyvissä sovelluksissa. (Richards 2022, 43-53.)

2.2 Sovelluksen eri komponenttien vuorovaikutus

Mobiilisovelluksen toiminta perustuu sen eri komponenttien, erityisesti käyttöliittymän (frontend), taustajärjestelmän (backend) ja tietokannan, saumattomaan vuorovaikutukseen. Tämä voidaan toteuttaa monessa eri muodossa, mutta yleistäen voidaan sanoa sen koostuvan kolmesta erillisestä kerroksesta: esityskerros (presentation layer), logiikkakerros (business layer) ja datakerros (data layer). Sovelluksen taustajärjestelmä taas koostuu kahdesta jälkimmäisestä kerroksesta. Nämä ovat vastuussa järjestelmän toiminnan logiikasta ja tietojen tallentamisesta tietokantojen avulla. (Richards 2022, 16-18.) Alla olevassa kuviossa 1 on kuvattu näiden eri kerrosten väliset toiminnot yksinkertaistettuna mallina.



Kuvio 1. Yksinkertaistettu malli eri komponenttien vuorovaikutuksesta

Esityskerros (Presentation Layer / Frontend), tai käyttöliittymäkerros, on se käyttöliittymä (UI), jonka käyttäjä näkee avatessaan sovelluksen. Se sisältää näytöt, navigointielementit, kontrollit ja visuaaliset elementit (Richards 2022, 10). Esimerkiksi WhatsAppin kaltaisessa viestisovelluksessa esityskerros koostuu keskustelunäkymistä, yhteystietoluetteloista, asetusvalikoista jne. Sen pääasiallinen funktio on mahdollistaa käyttäjän interaktiot vastaanottamalla syötteitä käyttäjiltä (kuten napautukset, pyyhkäisyt, tekstinsyöttö) ja näyttämällä alempien kerrosten tuottamaa dataa ymmärrettävässä muodossa (IBM 2025a).

Logiikkakerros (Business Layer / Backend Logic) sisältää sovelluksen ydinlogiikan, joka käsittelee tehtäviä, kuten laskutoimituksia, datan validointia, liiketoimintasääntöjen toteutusta, analytiikkaa, ilmoituksia, taustaprosesseja ja niin edelleen. Viestisovelluksessa tämä kerros hoitaa funktioita, kuten viestien lähettämisen ja vastaanottamisen, datan salauksen, roskapostin tunnistamisen, ilmoitusten hallinnan jne. Se vastaanottaa syötteitä esityskerroselta (käyttäjän toiminnot) ja datakerrokselta (haettu data), prosessoi ne liiketoimintasääntöjen mukaisesti ja valmistelee käyttöliittymässä näytettävät vasteet tai tallennettavat tiedot. (IBM 2025a.)

Datakerros (Data Layer / Backend Storage) hallitsee yhteyksiä tietokantoihin ja tallennusjärjestelmiin, mahdollistaen sovelluksen datan pysyvän tallentamisen ja noutamisen. Esimerkiksi WhatsAppissa viestidata, käyttäjäprofiilit ja asetukset on tallennettu useisiin tietokantoihin. Tämä kerros vastaa datan eheyden varmistamisesta ja tarjoaa rajapinnan logiikkakerrokselle datan käsittelyyn. (IBM 2025a.).

2.3 Taustajärjestelmän ja tietokannan kommunikointi

Taustajärjestelmän (logiikkakerros) ja tietokannan (datakerros) välinen kommunikaatio on keskeinen osa sovelluksen toimintaa. Tämä yhteys mahdollistaa datan tallentamisen, hakemisen, päivittämisen ja poistamisen. Yleensä tämä kommunikaatio tapahtuu käyttäen tietokanta-ajureita (database drivers) ja usein abstraktiotason lisääviä työkaluja, kuten ORM-kirjastoja (Object-Relational Mapper) tai kyselyrakentajia (query builders). (NestJs 2025.)

Tietokantakohtaiset ajurit ovat kirjastoja, jotka mahdollistavat sovelluksen kommunikoinnin tietokannan, kuten PostgreSQL:n, MySQL:n tai MongoDB:n, kanssa sen natiivia protokollaa käyttäen. Esimerkiksi Node.js-ympäristössä PostgreSQL-yhteys muodostetaan usein pg-ajurilla. Nämä ajurit vastaavat yhteydenmuodostuksesta, SQL-kyselyiden lähettämisestä relaatiotietokannoille ja tulosten vastaanottamisesta. Yhteydenotto tietokantapalvelimeen tapahtuu tyypillisesti TCP/IP-protokollan välityksellä tiettyyn porttiin, esimerkiksi PostgreSQL:n oletusporttiin 5432. (Carlson 2025.)

Abba (2022) kuvaa, kuinka ORM-kirjastot (Object-Relational Mappers) tarjoavat korkeamman tason abstraktion tietokantavuorovaikutukselle. Ne mahdollistavat tietokannan käsittelyn ohjelmointikielen objekteina ja luokkina suorien SQL-kyselyiden sijaan. ORM muuntaa objektitoiminnot (esimerkiksi tallennus, haku) automaattisesti tietokannan ymmärtämiksi komennoiksi. Tämä voi nopeuttaa kehitystä ja parantaa koodin luettavuutta ja ylläpidettävyyttä.

2.4 Teknologiavalinnat

Taustajärjestelmän teknologiavalinnat ovat perustavanlaatuisia päätöksiä, jotka vaikuttavat koko sovelluskokonaisuuden toimivuuteen, suorituskykyyn ja jatkokehitettävyyteen. Erityisesti kun taustajärjestelmä palvelee mobiilisovellusta, kuten tässä projektissa, on tärkeää valita teknologioita, jotka mahdollistavat tehokkaan tiedonsiirron ja skaalautuvuuden käyttäjämäärien kasvaessa. Näiden valintojen tulee myös tukea projektin yleisiä tavoitteita, kuten kehitysnopeutta ja ylläpidon helppoutta, ottaen huomioon olemassa olevat tai samanaikaisesti kehitettävät sovelluksen muut osat.

Sopivien ohjelmointikielien ja kehysten valintaprosessissa punnitaan usein monia näkökohtia, kuten tiimin osaamista, teknologian kypsyyttä, yhteisön tukea ja yhteensopivuutta projektin muiden osien kanssa. Tavoitteena on löytää tasapaino, joka palvelee parhaiten sekä välittömiä kehitystarpeita että pitkän aikavälin vision toteutumista. Tässä työssä keskeisenä ajurina tiettyihin teknologiavalintoihin on ollut pyrkimys yhtenäistää kehitysympäristöä ja hyödyntää synergiaetuja sovelluksen eri kerrosten välillä, mikä tarkentuu seuraavassa alaluvussa käsiteltäessä käytettäviä ohjelmointikieliä ja kehyksiä.

2.4.1 Käytettävät ohjelmointikielet- ja kehykset

Koska tämä työ on osa isompaa mobiilisovelluskokonaisuutta, jossa myös käyttöliittymä (frontend) toteutetaan TypeScriptillä (tarkemmin ottaen React Nativella), käytettäväksi ohjelmointikieleksi taustajärjestelmään valikoitui TypeScript. Tämä mahdollistaa yhtenäisen kielen käytön koko sovelluksen kehityksessä (full-stack TypeScript). Kun sekä frontend että backend käyttävät samaa kieltä, kehittäjien voi olla helpompi siirtyä työskentelemään eri osien välillä, ja koodin sekä tyyppimäärittelyjen uudelleenkäyttö voi olla mahdollista. TypeScriptin staattinen tyypitys, joka on JavaScriptin laajennus, auttaa havaitsemaan virheitä käännösvaiheessa, voi parantaa koodin luettavuutta ja ylläpidettävyyttä sekä mahdollistaa kehittyneemmät kehitystyökalut (kuten automaattinen täydennys ja refaktorointi) (Goldberg 2022, 6-9). Tämä voi olla hyödyllistä suurissa ja monimutkaisissa sovelluksissa.

Kehykseksi oli useita vaihtoehtoja Node.js-ekosysteemissä (kuten Express.js) ja muissa kielissä (kuten Python/Django, Java/Spring Boot, Go), mutta valinta kohdistui NestJS:ään. NestJS on modulaarinen, TypeScript-pohjainen kehys taustajärjestelmille, joka on rakennettu Node.js:n päälle ja hyödyntää oletuksena Express.js:ää (mutta tukee myös Fastifytä suorituskyvyn parantamiseksi).

Duan (2024) mukaan NestJS:n valinnalle on muun muassa seuraavia perusteluita:

1. TypeScript-tuki: NestJS on rakennettu TypeScriptillä ja tukee sitä ensisijaisesti, mikä sopii yhteen kielivalinnan kanssa.

2. Modulaarisuus ja rakenne: NestJS noudattaa modulaarista arkkitehtuuria (Modules, Controllers, Services) ja hyödyntää suunnittelumalleja kuten riippuvuuksien injektointia (Dependency Injection, DI). Tämä ohjaava (opinionated) lähestymistapa edistää koodin organisointia, uudelleenkäytettävyyttä ja ylläpidettävyyttä erityisesti suurissa projekteissa, verrattuna minimalistisempaan Express.js:ään, joka antaa enemmän vapautta mutta vaatii enemmän manuaalista organisointia. Modulaarisuus voi myös helpottaa sovelluksen skaalaamista.

3. Testattavuus: Riippuvuuksien injektointi ja modulaarinen rakenne tekevät NestJS-sovelluksista hyvin testattavia. Komponentteja on helppo testata yksikkötesteillä syöttämällä niille mock-riippuvuuksia. NestJS tarjoaa myös sisäänrakennettuja työkaluja testaamiseen (esimerkiksi @nestjs/testing).

4. Ekosysteemi ja Integraatiot: NestJS:llä on kasvava yhteisö ja ekosysteemi. Se tarjoaa valmiita integraatioita moniin yleisiin tarpeisiin, kuten tietokantojen ORM-kirjastoihin (TypeORM, Prisma), GraphQL:ään, WebSocketsiin, mikropalveluihin ja autentikointiin (Passport.js). Sillä on lisäksi myös kattava ja hyvin ylläpidetty dokumentaatio, mikä helpottaa oppimista ja ongelmanratkaisua.

Yhteenvetona, NestJS valittiin, koska se tarjoaa modernin, TypeScript-pohjaisen, modulaarisen ja skaalautuvan alustan, joka sopii yhteen projektin muiden osien kanssa ja edistää ohjelmistokehityskäytäntöjä, kuten testattavuutta ja ylläpidettävyyttä.

2.4.2 Muut työkalut ja kirjastot

Nykyaikaisen taustajärjestelmän kehittäminen vaatii usein perusohjelmointikielen ja -kehyksen lisäksi monipuolista joukkoa muita työkaluja ja kirjastoja. Nämä komponentit valitaan tyypillisesti tukemaan ja tehostamaan eri kehitysvaiheita, kuten tietokantavuorovaikutusta, autentikointia, datan validointia, konfiguraationhallintaa, suorituskyvyn optimointia, sovellusympäristön hallintaa ja versionhallintaa.

Käyttäjien tunnistamisen ja pääsynvalvonnan, eli autentikoinnin ja auktorisoinnin, toteuttamiseen hyödynnetään usein erikoistuneita kirjastoja ja standardeja. Esimerkiksi Passport.js on suosittu autentikointivälikerros Node.js-ympäristössä, joka tukee erilaisia strategioita (kuten paikallinen salasana-autentikointi tai JWT-pohjainen autentikointi) (Passport 2025). JSON Web Token (JWT) on yleinen standardi turvallisten pääsytunnisteiden (access tokens) ja päivitystunnisteiden (refresh tokens) luomiseen (Goldberg 2022, 172-175). Salasanojen turvalliseen tallentamiseen käytetään tiivistämisalgoritmeja (hashing), kuten bcrypt, jotka suojaavat salasanoja tietomurtojen varalta (Grigutytė 2023).

API-rajapintoihin saapuvan datan oikeellisuuden varmistaminen, eli datan validointi ja serialisointi, on kriittinen osa taustajärjestelmän toimintaa. Kirjastot, kuten class-validator ja class-transformer (TypeScript/JavaScript-maailmassa), mahdollistavat Data Transfer Object (DTO) -luokkien määrittelyn ja niihin liittyvien validointisääntöjen dekoraattoripohjaisen lisäämisen. Tämä auttaa estämään virheellisen datan päätymisen järjestelmään. (Ebenezer 2024.)

Sovellusten konfiguraationhallinta tähtää siihen, että konfiguraatioarvot, kuten tietokantayhteyden tiedot, API-avaimet ja JWT-salaisuudet, hallitaan turvallisesti ja joustavasti. Yleinen käytäntö on käyttää ympäristömuuttujia ja .env-tiedostoja, joita luetaan sovelluksessa erillisten konfiguraatiomoduulien (kuten @nestjs/config) avulla. Tämä mahdollistaa ympäristökohtaiset asetukset ja pitää arkaluontoiset tiedot erillään lähdekoodista. (Garfield 2025.)

Suorituskyvyn optimoinnissa yksi keskeinen keino on välimuistitus (caching). Sitä käytetään usein toistuvien ja laskennallisesti raskaiden operaatioiden tulosten väliaikaiseen tallentamiseen, mikä nopeuttaa sovelluksen vasteaikoja ja vähentää taustajärjestelmien, kuten tietokantojen, kuormitusta. Yleisiin välimuistiratkaisuihin lukeutuu esimerkiksi Redis, ja monet sovelluskehykset, kuten NestJS @nestjs/cache-manager-moduulillaan, tarjoavat työkaluja välimuistituksen helppoon integrointiin. (IBM 2025b.)

Kontitus ja ympäristönhallinta ovat myös tärkeitä moderning sovelluskehityksen osa-alueita. Docker-konttiteknologia on vakiintunut tapa paketoida sovellus ja sen riippuvuudet yhtenäiseksi, siirrettäväksi kokonaisuudeksi. Dockerfile määrittelee sovellusimagen rakennusohjeet, ja Docker Compose -työkalulla voidaan hallita monikonttisia sovellusympäristöjä paikallisessa kehityksessä. Kontitus takaa ympäristöjen yhdenmukaisuuden ja helpottaa käyttöönottoa eri alustoille. (Kane & Matthias 2018, 1-7.)

Sovellusten käyttöönottoon ja hostaukseen on saatavilla useita erilaisia PaaS (Platform as a Service) -käyttöönottoalustoja. Esimerkkejä tällaisista alustoista ovat Railway, Heroku, Google Cloud Run ja AWS Elastic Beanstalk. Nämä palvelut yksinkertaistavat merkittävästi

infrastruktuurin hallintaa ja tarjoavat usein valmiita integraatioita esimerkiksi tietokantoihin ja muihin tarvittaviin oheispalveluihin.

Nämä työkalut ja kirjastot muodostavat esimerkinomaisen kokonaisuuden, jota modernissa taustajärjestelmäkehityksessä tyypillisesti hyödynnetään. Kunkin työkalun valinta ja käyttöönoton syvyys riippuvat projektin vaatimuksista ja laajuudesta.

2.5 Tietokanta

Sovelluksen taustajärjestelmän keskeinen komponentti on tietokanta, joka toimii kaiken sovelluksen tarvitseman ja tuottaman tiedon tallennuspaikkana sekä hallintajärjestelmänä. Sen ensisijaisena tehtävänä on varmistaa datan pysyvyys, eheys ja tehokas saatavuus, mikä on elintärkeää sovelluksen toimivuuden ja luotettavuuden kannalta. Hyvin suunniteltu ja oikein valittu tietokantaratkaisu on perusta vakaalle ja suorituskykyiselle järjestelmälle, joka pystyy käsittelemään tietoa tehokkaasti ja palvelemaan käyttäjien tarpeita.

Tietokantateknologian valinta on merkittävä päätös, joka vaikuttaa laajasti sovelluskehitykseen, ylläpitoon ja järjestelmän skaalautuvuuteen tulevaisuudessa. Erilaisia tietokantajärjestelmiä on saatavilla runsaasti, ja kukin niistä tarjoaa erilaisia ominaisuuksia ja etuja riippuen tallennettavan datan luonteesta, transaktioiden hallinnan tarpeista ja odotetusta kuormituksesta.

2.5.1 Tietokantataulujen ja relaatioiden kuvaus

Tietokannat voidaan karkeasti jakaa kahteen eri tyyppiin niiden sisältämien tietojen rakenteen perusteella. Relaatiotietokannoissa (SQL) tieto säilötään taulukoihin ja näiden välisillä suhteilla on tärkeä rooli tiedon mallintamisessa. Tietokannoissa missä tätä relaatiomallia ei noudateta, kutsutaan ei-relaatiotietokannoiksi (NoSQL), ja niissä tieto voidaan säilöä esimerkki dokumenttimuodossa. (MongoDB 2025.)

SQL-tietokantaparadigmat, jotka perustuvat relaatiomalliin ja kiinteisiin skeemoihin, on optimoitu jäsennellylle datalle ja ACID-yhteensopiville transaktioille, skaalautuen tyypillisesti vertikaalisesti. Ne soveltuvat erityisesti korkeaa dataintegrititeettiä vaativiin sovelluksiin, kuten talousjärjestelmiin ja toiminnanohjausjärjestelmiin, joissa datan yhdenmukaisuus ja transaktioiden luotettavuus ovat tärkeitä. (Amazon Web Services 2025.)

Vastakohtana NoSQL-järjestelmät tarjoavat joustavia skeemoja ja moninaisia datamalleja (esimerkiksi dokumentti-, avain-arvo-, sarakeperhe- ja graafimallit) jäsentymättömän tai osittain jäsennellyn datan tehokkaaseen hallintaan. Ne mahdollistavat laajan horisontaalisen skaalautuvuuden, mikä tekee niistä sopivia suurten, heterogeenisten ja nopeasti

muuttuvien tietoaineistojen käsittelyyn esimerkiksi esineiden internetin (IoT), suurdatan analytiikan ja sisällönhallintajärjestelmien kontekstissa. (MongoDB 2025.)

Valinta relaatiopohjaisten SQL-tietokantojen ja ei-relaatiopohjaisten NoSQL-tietokantojen välillä on perustavanlaatuinen päätös tietojärjestelmien suunnittelussa, ja tietokantateknologian valinta edellyttää huolellista analyysiä sovelluskohtaisista vaatimuksista, erityisesti datan rakenteen, skeeman joustavuuden tarpeen, transaktioiden hallinnan vaatimusten ja odotetun skaalautuvuuden osalta.

2.5.2 Tietokannan normalisointi ja optimointi

Tietokannan normalisoinnilla tarkoitetaan menetelmää, jolla tietoaineisto järjestetään tietokannassa rakenteelliseksi ja eheäksi kokonaisuudeksi. Normalisoinnin ytimessä on taulukoiden luominen ja niiden välisten suhteiden määrittäminen tiukkojen sääntöjen mukaisesti. Näiden sääntöjen ensisijaisena tavoitteena on suojata tietoja ja parantaa tietokannan joustavuutta. Tämä saavutetaan poistamalla turhaa tiedon toistoa (redundanssia) ja epäloogisia riippuvuussuhteita. (Microsoft 2025.)

Tiedon toistaminen tietokannassa on ongelmallista, koska se vie tarpeettomasti tallennustilaa ja aiheuttaa ylläpitoon liittyviä haasteita. Jos samaa tietoa on päivitettävä useassa eri paikassa, on varmistettava, että päivitykset tehdään johdonmukaisesti ja täsmälleen samalla tavalla kaikkialla. Esimerkiksi asiakkaan osoitteen muuttaminen on huomattavasti yksinkertaisempaa, jos osoite on tallennettu vain yhteen paikkaan, esimerkiksi 'Asiakkaat'-taulukkoon, eikä moniin eri tauluihin ympäri tietokantaa. (Microsoft 2025.)

Epäjohdonmukainen riippuvuus puolestaan tarkoittaa tilannetta, jossa tietty tieto on tallennettu taulukkoon, johon se ei loogisesti kuulu. Vaikka on luontevaa etsiä asiakkaan osoitetta 'Asiakkaat'-taulukosta, työntekijän palkan etsiminen samasta taulukosta olisi epäloogista. Palkkatieto riippuu työntekijästä, joten loogisempi paikka palkan tallentamiselle olisi 'Työntekijät'-taulukko. Epäjohdonmukaiset riippuvuudet vaikeuttavat tiedon löytämistä tietokannasta, sillä ne voivat tehdä tiedonhakupoluista epäselviä tai katkonaisia. (Microsoft 2025.)

Tietokannan normalisointia ohjaavat normaalimuodot, jotka ovat asteittaisia sääntöjä. Ensimmäisen normaalimuodon (1NF) noudattaminen tarkoittaa tietokannan olevan "ensimmäisessä normaalimuodossa". Tämä tarkoittaa ettei taulukossa ole toistuvia ryhmiä. Toisessa normaalimuodossa (2NF) poistetaan tarpeettomasti toistuvat tiedot. Kolmannessa normaalimuodossa (3NF) poistetaan avaimesta riippumattomat tiedot. Kun kolme ensimmäistä normaalimuotoa (1NF, 2NF ja 3NF) on toteutettu, tietokannan katsotaan olevan "kolmannessa normaalimuodossa". Vaikka on olemassa korkeampiakin normalisoinnin tasoja,

kolmatta normaalimuotoa pidetään useimmissa sovelluksissa riittävänä ja jopa suositeltavana. (Microsoft 2025.)

Kuten monet teoreettiset mallit, täydellinen normalisointi ei aina ole käytännössä mahdollista tai tarkoituksenmukaista. Normalisointi lisää usein taulukoiden määrää, mikä voi joissakin tapauksissa tuntua työläältä. Jos kuitenkin päätetään joustaa kolmesta ensimmäisestä normalisointisäännöstä, on tärkeää ymmärtää, että tämä voi johtaa tiedon toistoon ja epäjohdonmukaisiin riippuvuussuhteisiin. Tällöin sovelluksen suunnittelussa on otettava huomioon näistä poikkeamista mahdollisesti aiheutuvat ongelmat ja pyrittävä minimoimaan ne. (Microsoft 2025.)

2.5.3 Objekti-relaatiokartoitus

Objekti-relaatiokartoitus on ohjelmistotekniikassa käytetty menetelmä, joka Amblerin (2022a) mukaan pyrkii ratkaisemaan olio-orientoituneen ja relaatiotietokantamallien välisen perustavanlaatuisen eroavuuden, eli niin kutsutun impedanssieron. Tämän menetelmän avulla sovelluskehittäjät voivat, kuten Ambler (2022b) kuvaa, käsitellä relaatiotietokantoja oliokeskeisesti abstrahoiden taulurakenteet ja tietueet olioiksi ja oliokokoelmiksi.

Keskeistä O/R-muunnoksessa on olion attribuuttien kuvaaminen tietokannan sarakkeisiin, minkä lisäksi olioihin liitetään usein pysyvyyden hallintaan tarvittavia "varjotietoja", kuten avaimia ja rinnakkaisuuden hallintamekanismeja. Näiden kuvausten määrittely tapahtuu metadatan avulla, joka ohjaa pysyvyyskehysten toimintaa. Erityisenä haasteena on olioiden periytymisrakenteiden sovittaminen relaatiomaailmaan, mihin on kehitetty useita strategioita, kuten koko hierarkian kuvaaminen yhteen tauluun, jokaisen konkreettisen luokan kuvaaminen omaan tauluunsa, tai jokaisen luokan kuvaaminen erilliseksi tauluksi. Myös olioiden väliset suhteet (yksi-yhteen, yksi-moneen, moni-moneen) ja niiden suuntautuvuus kuvataan relaatiotietokantaan tyypillisesti viiteavaimien ja assosiatiivisten taulujen avulla. (Ambler 2022b.)

Järjestelmän suorituskyvyn varmistaminen on olennainen osa O/R-muunnosta, ja optimointi voi kohdistua niin tietokantarakenteeseen, tiedonhakulogiikkaan kuin itse kuvausstrategioihinkin. Tavoitteena on tarjota tehokas abstraktiokerros, joka yksinkertaistaa tietokantavuorovaikutusta ja mahdollistaa keskittymisen sovelluksen ydinlogiikkaan. (Ambler 2022b).

2.6 Ohjelmointirajapinnat

Nykyaikaisessa ohjelmistokehityksessä ohjelmointirajapinnat (Application Programming Interface, API) ovat keskeisessä asemassa, mahdollistaen eri ohjelmistokomponenttien, sovellusten ja palveluiden välisen järjestelmällisen vuorovaikutuksen ja tiedonvaihdon. Ne

toimivat sopimuspohjaisina liityntöinä, jotka määrittelevät, miten eri osapuolet voivat kommunikoida keskenään, pyytää palveluita tai jakaa dataa. API:t ovat perustavanlaatuisia elementtejä esimerkiksi hajautetuissa järjestelmissä, mikropalveluarkkitehtuureissa ja mobiilisovellusten taustajärjestelmäintegraatioissa, edistäen modulaarisuutta ja uudelleenkäytettävyyttä.

Ohjelmointirajapintojen tehokas suunnittelu ja hyödyntäminen edellyttävät ymmärrystä niiden perusrakenteista, kuten päätepisteistä, sekä erilaisista arkkitehtuurityyleistä ja niiden soveltuvuudesta eri käyttötarkoituksiin. Lisäksi selkeä ja kattava dokumentaatio on välttämätöntä API:en helppokäyttöisyyden ja integroitavuuden varmistamiseksi. Tässä luvussa syvennytään näihin ohjelmointirajapintojen olennaisiin näkökohtiin: aluksi tarkastellaan päätepisteiden roolia ja toiminnallisuutta, minkä jälkeen esitellään yleisimpiä rajapintatyyppejä ja lopuksi käsitellään dokumentoinnin merkitystä ja käytäntöjä.

2.6.1 Ohjelmointirajanpinnan päätepisteet ja niiden toiminnallisuus

Tässä opinnäytetyössä API on keskeisessä roolissa mahdollistaen mobiilisovelluksen (frontend) ja taustajärjestelmän (backend) välisen vuorovaikutuksen, API-rajapinta koostuu joukosta päätepisteitä (endpoints). Päätepiste on käytännössä URL-osoite, johon asiakassovellus (esimerkiksi mobiilisovellus) voi lähettää pyyntöjä suorittaakseen tiettyjä operaatioita tai hakeakseen dataa taustajärjestelmästä. (SAP 2025.) Esimerkiksi päätepiste /users voisi palauttaa listan käyttäjistä, kun taas /users/{id} voisi palauttaa tietyn käyttäjän tiedot ID:n perusteella.

Tässä työssä keskitytään RESTful API -suunnittelumalliin, joka on yleisin tapa toteuttaa web-pohjaisia API-rajapintoja. REST (Representational State Transfer) ei ole tiukka protokolla, vaan joukko periaatteita ja rajoitteita, jotka ohjaavat hajautettujen järjestelmien suunnittelua (SAP 2025).

2.6.2 Erilaiset ohjelmointirajapintatyypit

RESTful- eli REST-arkkitehtuuriin pohjautuvat API-rajapinnat ovat suosittu valinta verkkopalveluiden kehityksessä. SAP (2025) kuvaa, kuinka ne hyödyntävät HTTP-protokollaa ja sen vakiintuneita metodeja, kuten GET, POST, PUT ja DELETE. RESTful-rajapinnat ovat tilattomia, mikä tarkoittaa, että jokainen asiakaspyyntö sisältää kaiken tarvittavan tiedon onnistuneen käsittelyn varmistamiseksi. Tämä ominaisuus parantaa järjestelmän suorituskykyä ja skaalautuvuutta. Tiedonsiirrossa käytetään yleensä JSON- tai XML-formaatteja. RESTful-rajapintojen yksinkertaisuus, helppo omaksuttavuus ja tehokkuus tekevät niistä

erinomaisen valinnan esimerkiksi pilvipohjaisiin palveluihin, mobiilisovelluksiin ja IoT-laitteisiin.

OData, lyhenne sanoista Open Data Protocol, on standardi, jonka tavoitteena on helpottaa tiedon jakamista ja järjestelmien välistä integraatiota. Se tarjoaa yhdenmukaisen lähestymistavan strukturoidun datan esille tuomiseksi ja hyödyntämiseksi. OData-rajapinnat noudattavat tiettyjä vakiintuneita käytäntöjä, joiden avulla asiakkaat voivat käsitellä tietoresursseja HTTP-protokollan perusmetodeilla (GET, POST, PUT, DELETE). OData-rajapinnat tukevat monipuolista kyselykieltä, mikä mahdollistaa tietovasteiden suodattamisen, järjestämisen ja muokkaamisen, tehostaen näin tiedonhakua. OData korostaa yhteentoimivuutta, helppokäyttöisyyttä ja standardisointia eri palveluiden ja alustojen välillä, minkä ansiosta se on arvokas työkalu organisaatioille, jotka pyrkivät virtaviivaistamaan tiedonsaantia ja parantamaan järjestelmiensä yhteispeliä. (SAP 2025.)

SOAP eli Simple Object Access Protocol -API-rajapinnat ovat, kuten SAP (2025) niitä luonnehtii, erittäin rakenteellisia ja protokolliltaan tiukkoja. Ne on suunniteltu erityisesti tapahtumienhallintaan ja korkean tietoturvatason tarjoamiseen, minkä vuoksi ne soveltuvat hyvin yritystason sovelluksiin, kuten rahoitusjärjestelmiin ja CRM-järjestelmiin. SOAP-rajapinnat kommunikoivat XML-pohjaisten viestien välityksellä ja ne tunnetaan luotettavuudestaan ja laajennettavuudestaan. Toisaalta ne voivat olla monimutkaisempia ja raskaampia kuin RESTful-rajapinnat, mikä saattaa johtaa hitaampaan suorituskykyyn tietyissä tilanteissa.

GraphQL on kyselykieli API-rajapinnoille sekä runtime-ympäristö kyselyiden toteuttamiseen, ja se hyödyntää tyyppijärjestelmää, jonka käyttäjä itse määrittelee datalleen. Toisin kuin RESTful-rajapinnat, jotka tyypillisesti tarjoavat useita eri päätepisteitä, GraphQL-rajapinnoilla on yleensä vain yksi päätepiste. Tämä mahdollistaa sen, että asiakas voi pyytää ainoastaan juuri ne tiedot, joita se tarvitsee. Tämän ansiosta GraphQL-rajapinnoista tulee joustavia ja tehokkaita, erityisesti monimutkaisissa järjestelmissä, joissa on suuria määriä ja monentyyppistä dataa. GraphQL onkin kasvattanut suosiotaan tehokkaan tiedonhaun ja kykynsä räätälöidä kyselyitä juuri tiettyihin tarpeisiin sopiviksi ansiosta. (SAP 2025.)

2.6.3 Dokumentaatio

Ohjelmistodokumentaatio on olennainen osa laadukasta järjestelmäkehitystä, ja se toimii keskeisenä apuvälineenä sovelluksen tai järjestelmän rakenteen, toiminnallisuuksien ja käytön ymmärtämisessä. Hyvin laadittu dokumentaatio helpottaa tiedonkulkua tiimin sisällä, nopeuttaa uusien kehittäjien perehtymistä projektiin sekä tukee ohjelmiston pitkäjänteistä ylläpitoa ja jatkokehitystä. Ohjelmointirajapinnan dokumentointi auttaa muita kehittäjiä ymmärtämään sen logiikan ja ominaisuudet.

NestJS tarjoaa integroidun moduulin (@nestjs/swagger) API-rajapintojen automaattiseen dokumentointiin OpenAPI-määritelmää hyödyntäen (NestJS 2025b). Näin mahdollistetaan dokumentaation luominen suoraan lähdekoodista, mikä vähentää manuaalisen dokumentaation tarvetta. Näin dokumentaatio pysyy myös paremmin ajan tasalla, vaikka koodi muuttuisikin. (Swagger 2025.)

2.7 Tietoturva ja yksityisyydensuoja

Nykyaikaisessa digitaalisessa maailmassa tietoturva ja yksityisyydensuoja ovat ensisijaisen tärkeitä näkökohtia minkä tahansa sovelluksen tai tietojärjestelmän suunnittelussa ja toteutuksessa. Ne eivät ainoastaan suojaa käyttäjien henkilökohtaisia ja arkaluontoisia tietoja luvattomalta käytöltä, muokkaukselta tai tuhoamiselta, vaan myös turvaavat organisaation maineen ja liiketoiminnan jatkuvuuden erilaisten uhkien, kuten tietomurtojen ja palvelunestohyökkäysten, varalta. Huolellinen tietoturva- ja yksityisyydensuojakäytäntöjen integrointi sovelluskehityksen jokaiseen vaiheeseen on välttämätöntä luottamuksen rakentamiseksi käyttäjien kanssa ja lakisääteisten vaatimusten, kuten yleisen tietosuoja-asetuksen (GDPR), noudattamiseksi.

Kattava tietoturva muodostuu useista kerroksista ja osa-alueista, joista yksi keskeisimmistä on sen varmistaminen, että järjestelmään ja sen sisältämään dataan pääsevät käsiksi ainoastaan oikeutetut henkilöt ja prosessit. Tämä edellyttää luotettavia mekanismeja käyttäjien tunnistamiseen sekä heidän pääsyoikeuksiensa tarkkaan hallintaan ja rajaamiseen. Seuraavassa alaluvussa syvennytäänkin tarkemmin peruspilareihin, jotka ovat kriittisiä komponentteja turvallisen ja luotettavan järjestelmän toteuttamisessa.

2.7.1 Käyttäjien autentikointi ja valtuutus

Autentikointi tarkoittaa prosessia, jolla varmistetaan käyttäjän tai järjestelmän identiteetti, eli vastataan kysymykseen "kuka sinä olet?". Käyttäjien kohdalla perinteisin tapa on esittää tunnistautumistiedot, yleensä käyttäjätunnuksen ja salasanan muodossa. Koska salasanat ovat alttiita heikkouksille, kuten arvaukselle, uudelleenkäytölle ja tietomurroissa paljastumiselle, pelkkään salasanaan perustuvaa tunnistautumista ei pidetä enää riittävän turvallisena monissa tapauksissa. Käyttäjien kohdalla perinteisin tapa on esittää tunnistautumistiedot, yleensä käyttäjätunnuksen ja salasanan muodossa. (Gough ym. 2022, 167)

Gough ym. (2022, 167-168) huomauttavat, että nykyään on yhä tavallisempaa, että monivaiheinen tunnistautuminen (MFA) on osa normaalia kirjautumisprosessia. MFA on keino lisätä varmuutta siitä, että käyttäjä todella on se, kuka hän väittää olevansa. MFA yhdistää vähintään kaksi eri todennustapaa eri kategorioista:

- Jotain, mitä tiedät: Salasana, PIN-koodi, turvakysymysten vastaukset.
- Jotain, mitä omistat: Fyysinen turva-avain (esimerkiksi FIDO U2F-avain), kertakäyttösalasanoja generoiva laite tai sovellus (OTP), älykortti, matkapuhelin (SMS-koodit tai puhelut).
- Jotain, mitä olet: Biometriset tunnisteet, kuten sormenjälki, kasvojen- tai iiriksentunnistus.
- Jossain, missä olet: Sijaintitieto (esimerkiksi IP-osoite, GPS-koordinaatit).
- Jotain, mitä teet: Käyttäytymiseen perustuvat tunnisteet, kuten kirjoitusrytmi tai hiiren liikkeet.

MFA lisää merkittävästi varmuutta siitä, että käyttäjä todella on se, kuka hän väittää olevansa, vähentäen tilien kaappausriskiä huomattavasti verrattuna pelkkään salasanaan. (OneLogin 2025.)

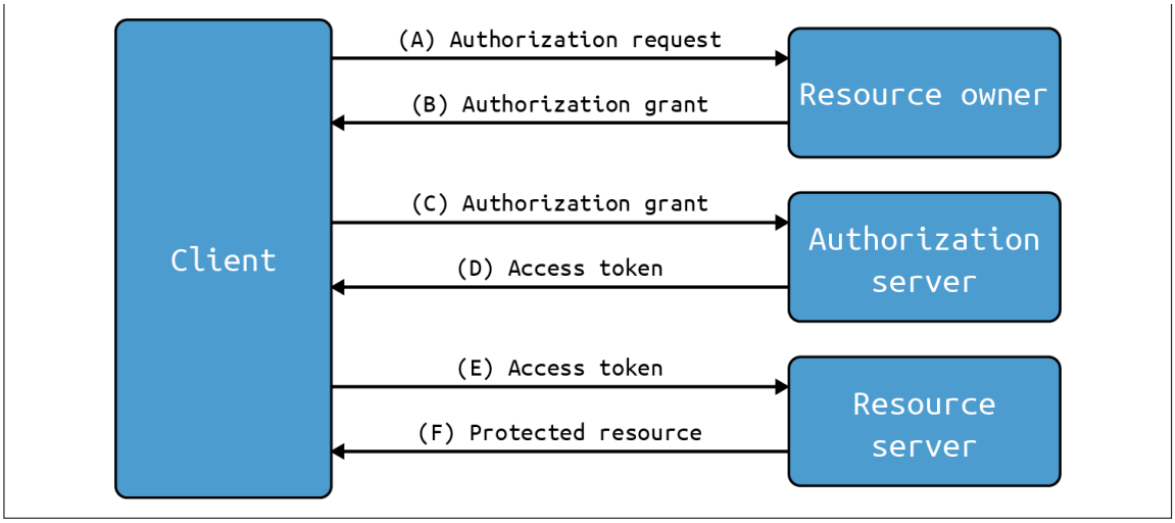
2.7.2 OAuth2

OAuth2 on tunnuspohjainen valtuutuskehys, joka mahdollistaa sen, että käyttäjä voi antaa suostumuksensa kolmannen osapuolen sovellukselle päästä käyttämään heidän tietojaan heidän puolestaan. Käyttäjän antama suostumus on valtuutus: hän joko sallii tai estää pääsyn tietoihinsa. (Gough ym. 2023, 171) OAuth2:n toiminnassa on oleellisesta siihen sisältyvät roolit. Sen virallisessa määritelmässä (Internet Engineering Task Force 2025) ne ovat:

- Resurssin omistaja (Resource Owner): Entiteetti, jolla on oikeus myöntää pääsy suojattuun resurssiin. Kun resurssin omistaja on henkilö, häntä kutsutaan loppukäyttäjäksi.
- Valtuutuspalvelin (Authorization Server): Palvelin, joka myöntää käyttöoikeustunnisteita (access token) asiakkaalle, kun resurssin omistaja on tunnistautunut onnistuneesti ja valtuutus on saatu. Useimmat identiteetintarjoajat, kuten Google tai Auth0, toimivat OAuth2-valtuutuspalvelimina.
- Asiakas (Client): Sovellus, joka tekee suojattuja resurssipyyntöjä resurssin omistajan puolesta ja tämän valtuutuksella.
- Resurssipalvelin (Resource Server): Palvelin, joka ylläpitää suojattuja resursseja ja pystyy hyväksymään ja käsittelemään suojattuja resurssipyyntöjä käyttöoikeustunnisteiden avulla.

Valtuutusprosessi on esitetty kuvassa 2 ja sen mukaiset vaiheet ovat seuraavat:

- Vaihe (A): Asiakas käynnistää valtuutusprosessin lähettämällä valtuutuspyynnön resurssin omistajalle. Tämä ohjaa resurssin omistajan tyypillisesti valtuutuspalvelimen tarjoamaan käyttöliittymään (esimerkiksi verkkosivulle).
- Vaihe (B): Resurssin omistaja suorittaa valtuutuksen myöntämisen. Tämä vaihe voi sisältää tunnistautumisen valtuutuspalvelimessa ja nimenomaisen suostumuksen antamisen asiakkaan pääsylle resursseihin.
- Vaihe (C): Valtuutuksen myöntämisen jälkeen asiakas välittää valtuutusmyönnyksen valtuutuspalvelimelle.
- Vaihe (D): Valtuutuspalvelin vastaa asiakkaalle ja luovuttaa käyttöoikeustunnisteen. Tämä tunniste toimii digitaalisena valtuutuksena myöhempiä resurssipyyntöjä varten.
- Vaihe (E): Asiakas esittää käyttöoikeustunnisteen resurssipalvelimelle jokaisen resurssipyynnön yhteydessä.
- Vaihe (F): Resurssipalvelin verifioi käyttöoikeustunnisteen validiteetin. Mikäli tämä tunnistetaan voimassaolevaksi, resurssipalvelin palauttaa pyydetyn suojatun resurssin asiakkaalle.



Kuvio 2. Valtuutusprosessi (Gough ym. 2023, 176)

2.7.3 Tietokannan ja API:n suojausmenetelmät

Tietoturva laajana käsitteenä kattaa strategiat, käytännöt ja teknologiat, joiden avulla suojataan kaikenlaista tietoa ja tietojärjestelmiä moninaisilta uhilta, kuten luvattomalta käytöltä, paljastumiselta tai häirinnältä. Kiihtyvän digitalisaation ja järjestelmien verkkoutumisen aikakaudella vahvat tietoturvakäytännöt ovat perustavanlaatuisia organisaatioiden toiminnan

suojaamiselle, sääntelyn noudattamiselle sekä luottamuksen rakentamiselle käyttäjien ja sidosryhmien keskuudessa.

Kumar (2024) huomauttaa taustajärjestelmän tietoturvan olevan kriittisen tärkeää, erityisesti kun käsitellään potentiaalisesti arkaluonteista käyttäjädataa API-rajapintojen ja tietokantojen kautta. Puutteellinen tietoturva voi johtaa luvattomaan pääsyyn, tietomurtoihin, maineen menetykseen ja lakisääteisten vaatimusten rikkomiseen. Tietoturva ei ole kertaluonteinen tehtävä, vaan jatkuva prosessi, joka vaatii säännöllistä ylläpitoa ja mukautumista kehittyviin uhkiin. Tehokas suojaus edellyttää huolellista lähestymistapaa, jossa huomioidaan sovelluksen kaikkien tasojen turvallisuus.

2.7.4 Yksityisyydensuojan huomioiminen

Henkilötietojen käsittelyssä on noudatettava tarkasti Euroopan parlamentin ja neuvoston asetusta (EU) 2016/679, joka tunnetaan yleisesti nimillä yleinen tietosuoja-asetus tai General Data Protection Regulation (GDPR). Tämä asetus määrittelee kattavat vaatimukset sille, miten henkilötietoja tulee kerätä, käsitellä, säilyttää ja suojata. Sovelluskehittäjän näkökulmasta erityisen keskeinen on GDPR:n artiklassa 25 esitetty sisäänrakennetun ja oletusarvoisen tietosuojan periaate. Kuten myös Euroopan komissio (2025a) korostaa, tämä periaate velvoittaa huomioimaan tietosuojan vaatimukset sovelluksen suunnittelussa heti alusta alkaen ja varmistamaan, että oletusasetukset ovat yksityisyyttä mahdollisimman hyvin suojaavia. Yleisen tietosuoja-asetuksen soveltamisalaa ja keskeisiä määritelmiä, kuten henkilötiedon ja käsittelyn käsitteitä, tarkennetaan sen artikloissa 2 ja 4.

Kaikkea henkilötietojen käsittelyä ohjaavat yleisen tietosuoja-asetuksen (GDPR) keskeiset perusperiaatteet, jotka on kattavasti esitelty sen 5 artiklassa. Yksi näistä on lainmukaisuuden, kohtuullisuuden ja läpinäkyvyyden periaate. Tämä edellyttää, että henkilötietojen käsittelylle on aina olemassa laillinen käsittelyperuste, jotka on lueteltu GDPR:n 6 artiklassa. Lisäksi rekisteröityjä on informoitava selkeästi heidän tietojensa käsittelystä, esimerkiksi tietosuojaselosteen avulla, noudattaen tiedottamista koskevia vaatimuksia, jotka löytyvät asetuksen artikloista 12, 13 ja 14.

Toinen olennainen periaate on käyttötarkoitussidonnaisuus (GDPR, artikla 5(1)(b)), jonka mukaan henkilötietoja saa kerätä vain tiettyä, nimenomaista ja laillista tarkoitusta varten, eikä niitä saa myöhemmin käsitellä näiden tarkoitusten kanssa yhteensopimattomalla tavalla. Tähän liittyy läheisesti tietojen minimoinnin periaate (GDPR, artikla 5(1)(c)), joka velvoittaa keräämään vain sellaisia henkilötietoja, jotka ovat tarpeellisia ilmoitetun käyttötarkoituksen kannalta.

Henkilötietojen on myös oltava täsmällisiä ja tarvittaessa päivitettyjä (GDPR, artikla 5(1)(d)). Lisäksi niiden säilytystä on rajoitettava siten, että tietoja säilytetään vain niin kauan kuin se on tarpeellista niitä käyttötarkoituksia varten, joita varten ne on kerätty (GDPR, artikla 5(1)(e)). Tietojen eheys ja luottamuksellisuus on varmistettava asianmukaisin teknisin ja organisatorisin toimenpitein (GDPR, artikla 5(1)(f)). Lopuksi keskeinen vaatimus on osoitusvelvollisuus (GDPR, artikla 5(2)), mikä tarkoittaa, että rekisterinpitäjän on kyettävä osoittamaan noudattavansa kaikkia edellä mainittuja tietosuojaperiaatteita.

Käytännössä yleisen tietosuoja-asetuksen (GDPR) periaatteiden toteuttaminen vaatii useita teknisiä ja organisatorisia toimenpiteitä. Tietoturvan varmistaminen, esimerkiksi datan salauksella ja pääsynvalvonnalla, on keskeistä (GDPR, artikla 32). GDPR myös rohkaisee pseudonymisoinnin kaltaisten menetelmien käyttöön (artikla 25), joissa suorat tunnisteet korvataan esimerkiksi koodeilla, mikä pienentää henkilötietoihin liittyviä riskejä, vaikka data pysyykin henkilötietona. Anonymisointi puolestaan tarkoittaa henkilötietojen muuttamista sellaiseen muotoon, että yksittäistä henkilöä ei voida enää tunnistaa. Tällöin, kuten GDPR:n johdanto-osan kappaleessa 26 todetaan, data ei enää kuulu asetuksen soveltamisalaan. Kaikkien tällaisten teknisten ja organisatoristen toimien on tuettava asetuksen yleisiä periaatteita (GDPR, artikla 45).

Taustajärjestelmän tulee myös mahdollistaa käyttäjien, eli rekisteröityjen, yleisessä tietosuoja-asetuksessa määriteltyjen oikeuksien tehokas toteuttaminen. Nämä oikeudet, jotka on kirjattu yksityiskohtaisesti GDPR:n III lukuun (artiklat 12-23), sisältävät muun muassa oikeuden saada pääsy omiin tietoihin, oikeuden tietojen oikaisuun, oikeuden tietojen poistamiseen (niin sanottu "oikeus tulla unohdetuksi"), oikeuden käsittelyn rajoittamiseen sekä oikeuden siirtää tiedot järjestelmästä toiseen. Näiden oikeuksien käytännön toteuttaminen sovelluksessa vaatii asianmukaisia API-päätepisteitä ja taustalogiikkaa pyyntöjen käsittelyyn sekä käyttäjän luotettavaan tunnistamiseen.

Yleinen tietosuoja-asetus, erityisesti sen artiklat 33 ja 34, asettaa myös velvollisuuden ilmoittaa henkilötietojen tietoturvaloukkauksista valvontaviranomaiselle 72 tunnin kuluessa siitä, kun loukkaus on tullut rekisterinpitäjän tietoon, mikäli loukkaus todennäköisesti aiheuttaa riskin yksilöiden oikeuksille ja vapauksille. Tämä korostaa tehokkaan lokituksen ja valvonnan merkitystä loukkausten nopeassa havaitsemisessa ja niihin reagoimisessa. Lisäksi datan fyysisellä säilytyspaikalla on huomattava merkitys. GDPR edellyttää lähtökohtaisesti henkilötietojen säilyttämistä Euroopan unionin tai Euroopan talousalueen (EU/ETA) sisällä, tai sellaisissa kolmansissa maissa, joiden osalta Euroopan komissio on tehnyt päätöksen tietosuojan tason riittävyydestä. Mikäli tietoja siirretään näiden alueiden ulkopuolelle, on yleisen tietosuoja-asetuksen V luvun (artiklat 44-50) mukaisesti varmistettava riittävät

suojatoimet, esimerkiksi käyttämällä komission hyväksymiä vakiosopimuslausekkeita (Standard Contractual Clauses, SCC). Tämä seikka on otettava huomioon hosting-palveluita ja muita alihankkijoita valittaessa.

2.8 Tekoäly

Tekoäly (engl. Artificial Intelligence, AI) on tietojenkäsittelytieteen monitieteinen ala, joka keskittyy sellaisten laskennallisten järjestelmien suunnitteluun ja toteutukseen, jotka kykenevät suorittamaan tehtäviä, joita perinteisesti pidetään ihmisen älykkyyttä vaativina. Näihin lukeutuvat muun muassa oppiminen, päättely, ongelmanratkaisu, hahmontunnistus ja luonnollisen kielen käsittely. Tekoälyn, ja erityisesti sen osakentän koneoppimisen, menetelmällinen kehitys ja laskentakapasiteetin kasvu ovat mahdollistaneet sen yhä laajemman soveltamisen eri tieteenaloilla ja teollisissa sovelluksissa.

Ohjelmistotekniikassa tekoälymenetelmien integrointi voi tarjota välineitä esimerkiksi suurten tietomäärien analyysiin, monimutkaisten prosessien mallintamiseen ja automatisointiin sekä adaptiivisten käyttöliittymien kehittämiseen. Tekoälypohjaisten komponenttien lisääminen järjestelmään edellyttää kuitenkin sovelluskohteen vaatimusten, käytettävissä olevan datan ominaisuuksien ja valittujen algoritmien soveltuvuuden huolellista tarkastelua. Seuraavassa alaluvussa käsitellään niitä spesifisiä rooleja ja toiminnallisuuksia, joissa tekoälyä voitaisiin hyödyntää tämän projektin puitteissa, sekä arvioidaan sen mahdollista vaikutusta sovelluksen ominaisuuksiin.

2.8.1 Tekoälyn rooli sovelluksessa

Tekoälyn (artificial intelligence, AI) ja koneoppimisen (machine learning, ML) integrointi taustajärjestelmään voi avata uusia mahdollisuuksia sovelluksen toiminnallisuuden ja käyttäjäkokemuksen parantamiseen. Vaikka tämän opinnäytetyön proof-of-concept ei sisällä laajamittaista tekoälytoteutusta, sen potentiaalisia rooleja voidaan hahmotella tulevaa kehitystä varten.

Tähän sovellukseen on tarkoitus lisätä tekoäly tunnistamaan käyttäjien ottamia kuvia ja luokittelemaan niitä eri kategorioihin. Myöhempänä kehityskohteena olisi myös lisätä mahdollisuus esimerkiksi eri sienilajikkeiden tunnistamiseen, mutta tämä vaatii merkittävää lisäkehitystyötä sekä varotoimia myrkyllisten lajikkeiden takia.

2.8.2 Tekoälymallien vaihtoehdot

Tekoälytoiminnallisuuksien lisäämiseen ohjelmistoihin on olemassa erilaisia teknisiä ratkaisumalleja, jotka vaihtelevat toteutustavan, vaadittavan osaamisen ja resurssien suhteen:

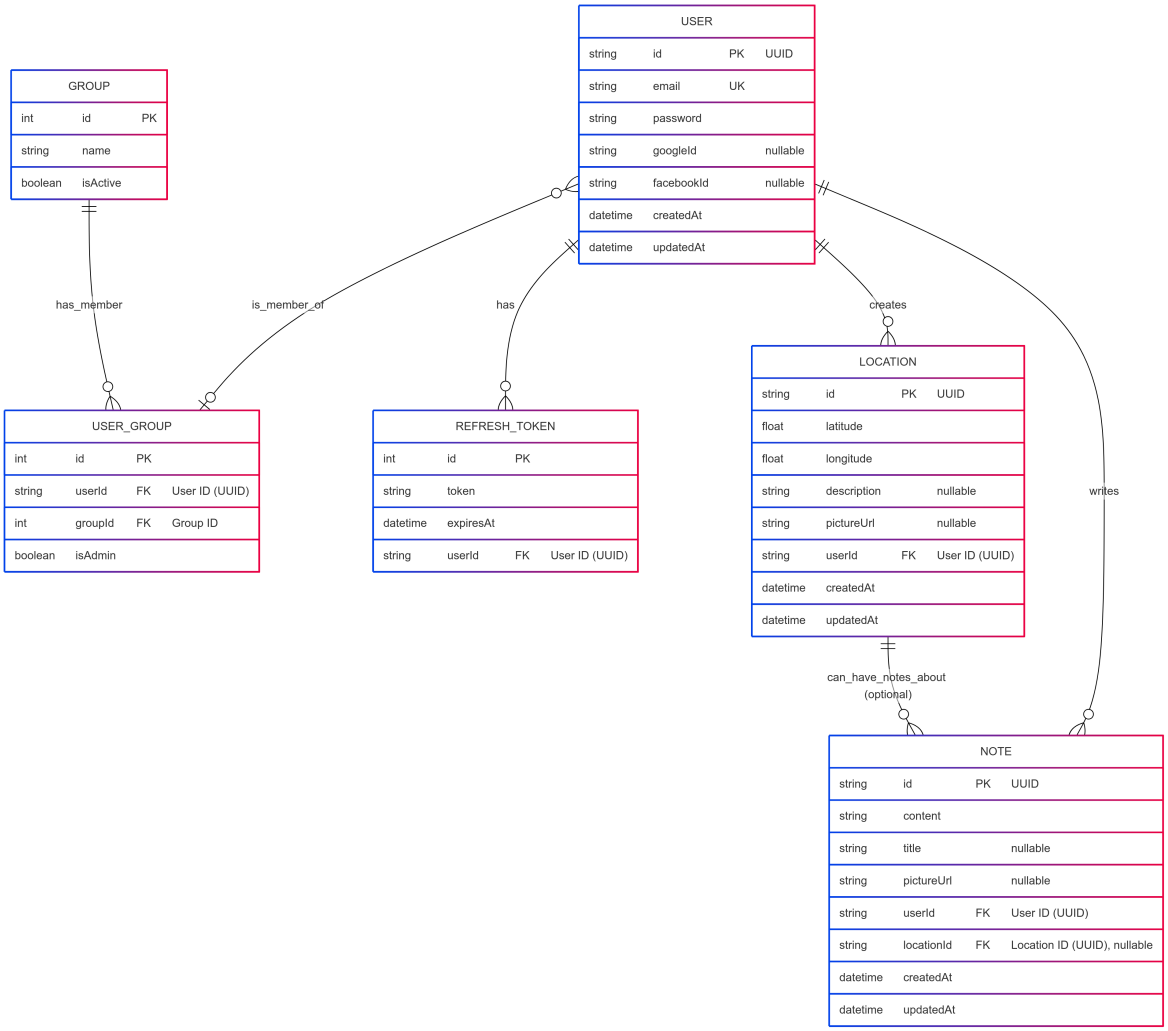
- Ulkoiset tekoälypalvelut: Sovellus voisi hyödyntää valmiita tekoälypalveluita (esimerkiksi suurten pilvitoimittajien tai erikoistuneiden yritysten tarjoamia API-rajapintoja). Tämä on mahdollisesti nopea tapa päästä alkuun ja käyttää kehittyneitä malleja ilman syvää omaa tekoälyosaamista. Haasteina voivat olla kustannukset, mahdolliset viiveet ja tietosuojakysymykset, kun dataa siirretään ulkopuoliselle palvelulle.
- Koneoppimiskirjastojen suora käyttö sovelluksessa: Tekoälymalleja voidaan ajaa suoraan sovelluksen omassa ympäristössä käyttäen soveltuvia ohjelmakirjastoja. Tämä pitää datan sovelluksen sisällä, mikä on hyvä tietosuojan kannalta, mutta voi rasittaa sovelluksen suorituskykyä ja käytettävissä olevien työkalujen valikoima voi olla rajallisempi kuin erikoistuneissa tekoäly-ympäristöissä.
- Erillinen tekoäly-mikropalvelu: Voitaisiin rakentaa erillinen, tekoälyyn keskittyvä palvelu (käyttäen siihen parhaiten soveltuvia teknologioita, kuten Pythonia). Pääsovellus kommunikoisi tämän erillisen palvelun kanssa. Tämä malli mahdollistaa tehokkaan erikoistumisen ja skaalautuvuuden tekoälyn osalta, mutta lisää järjestelmän monimutkaisuutta.

Valinta näiden vaihtoehtojen välillä riippuu projektin spesifisistä tarpeista, tiimin osaamisesta, budjetista, suorituskykyvaatimuksista ja tietosuojastrategiasta. Proof-of-concept-vaiheessa ulkoisten API-palveluiden kokeilu voi olla matalan kynnyksen tapa validoida ideoita, kun taas pitkäjänteisemmässä kehityksessä sisäisten kirjastojen tai mikropalveluiden käyttö voi tarjota enemmän kontrollia ja joustavuutta.

3 Toteutus

3.1 Tietokannan toteutus

Projektin taustajärjestelmän perustaksi valittiin PostgreSQL-relaatiotietokanta (versio 14), koska sovelluksen ydintoiminnallisuudet, kuten käyttäjien, ryhmien, sijaintien ja muistiinpanojen hallinta, sekä näiden väliset monimutkaiset suhteet, mallintuivat luontevasti ja tehokkaasti relaatiomalliin. Sovelluksen tietokannan looginen rakenne, sisältäen keskeiset entiteetit (kuten User, Group, Location ja Note) ja niiden väliset relaatiot, on esitetty yksityiskohtaisemmin Kuviossa 3.



Kuvio 3. Relaatiotietokannan rakenne

Tietokantavuorovaikutuksen hallintaan sovelluskoodissa valittiin TypeORM-kirjasto sen tarjoamien olio-relaatiokartoitusominaisuuksien (ORM) ja vahvan TypeScript-integraation vuoksi. Tämä lähestymistapa abstraktoi SQL-kyselyiden kirjoittamista ja parantaa kehittäjän tuottavuutta sekä koodin ylläpidettävyyttä. Tietokannan looginen rakenne määriteltiin

TypeORM:n entiteettiluokkien avulla, jotka kuvaavat sovelluksen datamallia ja vastaavat suoraan tietokannan tauluja ja niiden välisiä relaatioita.

3.1.1 Tietokannan luominen ja konfigurointi

Kehitysympäristön yhtenäisyyden ja siirrettävyyden varmistamiseksi PostgreSQL-palvelin ajettiin omassa Docker-kontissaan. Tämän kontin asetukset määriteltiin docker-compose-tiedostossa. Sovelluksen ja tietokannan välisessä yhteydenpidossa käytettiin TypeORM-kirjastoa tietokantaoperaatioiden suorittamisen mahdollistamiseksi TypeScript-olioden kautta.

Tietokannan taulut määriteltiin TypeORM-entiteetteinä (User, Group, RefreshToken, Location, Note, UserGroup). Entiteettien ominaisuudet määriteltiin dekoraattoreilla, kuten @Column({ type: 'text', nullable: true, unique: true }), @PrimaryGeneratedColumn('uuid'), @CreateDateColumn() ja @UpdateDateColumn(). Näillä voitiin tarkasti määritellä sarakkeen tietotyyppi, rajoitteet (pakollisuus, uniikkius, oletusarvo) ja erikoistoiminnot (kuten UUID-generointi tai automaattiset aikaleimat).

Entiteettien väliset suhteet (1:N, M:N) implementoitiin @ManyToOne, @OneToMany ja @ManyToMany-dekoraattoreilla. User-Group-suhteen toteutus UserGroup-liitosentiteetillä mahdollisti isAdmin-lisätiedon tallentamisen suoraan relaatioon liittyen, tarjoten joustavan tavan hallita ryhmäkohtaisia oikeuksia. Tietokannan viite-eheyden säilyminen poisto-operaatioiden yhteydessä varmistettiin käyttämällä TypeORM:n relaatioiden onDelete-optioita. Esimerkiksi CASCADE-asetus asetettiin siten, että käyttäjää poistettaessa myös hänen luomansa päivitystunnisteet (refresh tokens) ja muistiinpanot poistuvat automaattisesti. Vastaavasti SET NULL -optiota hyödynnettiin esimerkiksi tilanteessa, jossa sijainnin poistaminen aiheuttaa kyseiseen sijaintiin viittaavien muistiinpanojen sijaintiviittauksen asettumisen NULL-arvoon, säilyttäen kuitenkin itse muistiinpanot.

Kehitysprosessin nopeuttamiseksi käytettiin TypeORM:n synchronize: true -asetusta, joka pitää tietokantaskeeman automaattisesti synkronissa entiteettimääritysten kanssa. Tämä asetus on tarkoitettu vain kehityskäyttöön eikä sitä tulisi käyttää tuotannossa, koska se voi aiheuttaa datan menetystä skeemamuutosten yhteydessä. Tuotantokelpoinen ratkaisu vaatii siirtymisen TypeORM CLI -työkalulla generoitaviin ja hallittaviin migraatiotiedostoihin, jotka tallennetaan versionhallintaan ja ajetaan kontrolloidusti osana käyttöönottoa.

3.1.2 Tietokannan testaus

Tietokannan toiminnallisuus validoitiin pääosin epäsuorasti API-rajapinnan kautta Postman-työkalulla. Testit kattoivat systemaattisesti kaikki CRUD-operaatiot toteutetuille entiteeteille

(User, Group, Location, Note) ja varmistivat relaatioiden oikeellisen toiminnan, kuten Note-entiteetin onnistuneen liittämisen Location- ja User-entiteetteihin tai UserGroup-liitostaulun päivittymisen käyttäjää lisättäessä/poistettaessa ryhmästä. Soft delete -logiikka (Group-entiteetin isActive-lippu) varmennettiin testaamalla ryhmän poisto ja sen jälkeinen näkymättömyys listoilla sekä käyttäjätiedoissa.

Manuaalista tarkistusta DBeaver-työkalulla hyödynnettiin erityisesti varmistamaan tietokannan tila operaatioiden jälkeen. Tällä vahvistettiin esimerkiksi, että isActive-lipun arvo oli false soft deleten jälkeen, isAdmin-lippu asetettiin oikein UserGroup-tauluun, RefreshToken-tietueet tallentuivat/poistuivat odotetusti, ja että valinnaisen Note-Location-suhteen locationId-vierasavain asetettiin NULL:ksi TypeORM:n toimesta oikein. Vaikkakin manuaalinen, tämä tarjosi tärkeän varmistuksen datan eheydestä ja ORM-kirjaston toiminnasta kehitysvaiheessa.

3.2 API:n toteutus

Kuten tämän työn teoriaosiossa (luku 2.6) on kuvattu, ohjelmointirajapinta (API) on keskeinen elementti nykyaikaisissa ohjelmistoarkkitehtuureissa. Tämän projektin taustajärjestelmässä API toimii ensisijaisena kanavana, jonka kautta asiakassovellukset hyödyntävät palvelinpuolen logiikkaa, käsittelevät dataa ja suorittavat toimintoja. Sen rooli on kriittinen järjestelmän eri osien välisen hallitun ja tehokkaan vuorovaikutuksen varmistamisessa sekä kokonaisuuden modulaarisuuden ja ylläpidettävyyden tukemisessa.

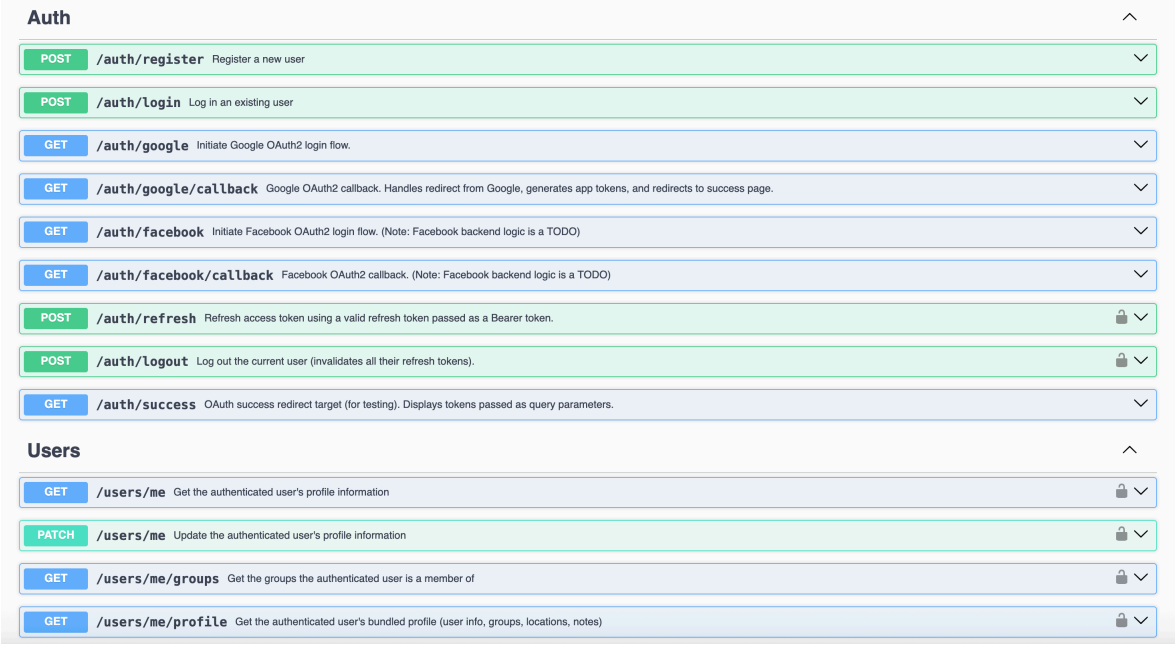
API:n käytännön toteutuksessa tehdään useita konkreettisia teknologia- ja suunnitteluvalintoja, jotka määrittävät sen toiminnalliset ominaisuudet, suorituskyvyn sekä tietoturvan tason. Näihin valintoihin kuuluvat esimerkiksi käytettävät ohjelmointikielet, sovelluskehykset, arkkitehtuurimallit ja noudatettavat rajapintastandardit, kuten REST. Seuraavissa alaluvuissa pureudutaan yksityiskohtaisesti tämän sovelluksen API:n toteutuksessa tehtyihin ratkaisuihin, käsitellen muun muassa sen rakennetta, teknologiavalintoja ja keskeisiä päätepisteitä.

3.2.1 Päätepisteet

Ohjelmointirajapinta toteutettiin NestJS-sovelluskehyksellä TypeScript-kieltä käyttäen. Tämä mahdollisti vahvan tyypityksen ja modulaarisen rakenteen, jota on helppo laajentaa tarvittaessa. Sovellus noudattaa modulaarista suunnittelumallia, jossa toiminallisuus on jaettu loogisiin kokonaisuuksiin (AuthModule, UsersModule, GroupsModule, LocationsModule). Jokainen moduuli sisältää tyypillisesti Controller-, Service- ja Entity/DTO-komponentit noudattaen vastuunjaon periaatetta (Separation of Concerns). Controllerit vastaavat http-

pyyntöjen vastaanottamisesta ja vastausten lähettämisestä, Servicet sisältävät sovelluksen sisäisen logiikan ja Repositoryt (TypeORM:n kautta) hoitavat tietokantavuorovaikutuksen.

Rajapinnan suunnittelussa pyrittiin noudattamaan RESTful-periaatteita (esimerkiksi resurssipohjaiset URL-polut ja standardien http-metodien käyttö: GET, POST, PATCH, DELETE). Kuva 1 antaa yleiskuvan keskeisimmistä toteutetuista päätepisteistä Auth- ja Users-moduuleissa, sellaisina kuin ne näkyvät generoidussa Swagger/OpenAPI-dokumentaatiossa. Auth-moduuli sisältää päätepisteet käyttäjän rekisteröinnille (/auth/register), sisäänkirjautumiselle (/auth/login), OAuth2-kirjautumisille Googlella ja Facebookilla (/auth/google, /auth/facebook ja niiden callbackit), JWT-päivitystunnisteen käytölle (/auth/refresh) sekä uloskirjautumiselle (/auth/logout). Users-moduuli tarjoaa päätepisteet autentikoidun käyttäjän tietojen hakuun ja päivitykseen (/users/me), käyttäjän ryhmien listaukseen (/users/me/groups) sekä yhdistetyn profiilidatan noutoon (/users/me/profile).



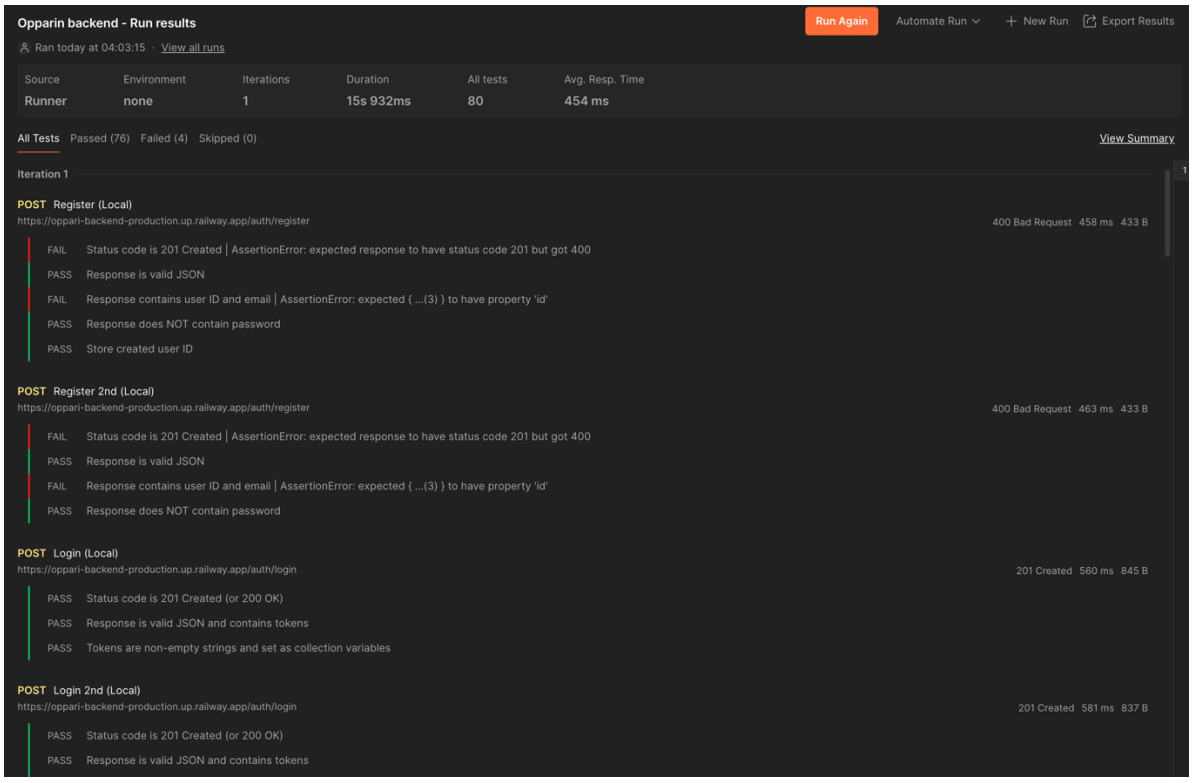
Kuva 1. Näkymä Swagger-dokumentaatiosta selaimessa

Pyyntöjen validoinnissa hyödynnettiin Data Transfer Object (DTO) -luokkia yhdessä class-validator ja class-transformer -kirjastojen kanssa. Näin rajapinnalle lähetetyn datan varmistetaan olevan oikeassa muodossa. Päätepisteet on suojattu siten, että tiettyihin päätepisteisiin on pääsy vain autentikoineilla käyttäjillä.

3.2.2 API:n testaus

Ohjelmointirajapinnan toimivuutta testattiin kehitysvaiheessa systemaattisesti Postman-työkalulla. Projektille luotiin ja ylläpidettiin Postman-kokoelmaa, joka sisältää pyynnöt kaikille toteutetuille päätepisteille. Kuva 2 esittää näkymän Postmanin Collection Runnerista,

jossa koko Opparin backend -testikokoelma on ajettu. Yhteenvedosta nähdään ajettujen testien kokonaismäärä (80), läpäisseiden (76) ja epäonnistuneiden (4) testien lukumäärät sekä keskimääräinen vastausaika. Tarkempi erittely näyttää esimerkkejä yksittäisten pyyntöjen, kuten POST /auth/register ja POST /auth/login, testien tuloksista, joissa tarkistetaan muun muassa vastausten status-koodia, JSON-muodon oikeellisuutta ja tiettyjen kenttien olemassaoloa tai puuttumista. Kuvassa näkyvät epäonnistuneet POST /auth/register -testit johtuvat odotetusti siitä, että testidata yrittää rekisteröidä käyttäjiä sähköpostiosoitteilla, jotka ovat jo olemassa tietokannassa, ja järjestelmä palauttaa oikeaoppisesti 400 Bad Request -virheen status-koodin 201 Created sijaan, mikäli sähköpostin uniikkiusrajoite on määritelty. Tämä osoittaa virheenkäsittelyn toimivuuden kyseisissä tilanteissa.



Kuva 2. Postmanissa ajetun testikokoelman tulokset

Testaus kattoi onnistuneet operaatiot oikeilla syötteillä, virheellisten syötteiden käsittelyn, autentikointi- ja auktorisointitilanteet, resurssien omistajien tarkastukset, sekä olemattomien resurssien käsittelyn. Näiden lisäksi testauksen kuului myös JWT refresh tokenien uusinta, ryhmien soft deleten vaikutus näkyvyyteen ja välimuistituksen toiminta. Testauksessa hyödynnettiin Postmanin ominaisuuksia, kuten ympäristömuuttujia ({{baseURL}}) ja tokenien manuaalista syöttämistä Authorization-otsakkeeseen. Iteratiivinen testaus suoritettiin aina merkittävien koodimuutosten tai uusien ominaisuuksien lisäämisen jälkeen. Docker-konttien ja Railwayn lokien seuranta oli olennainen osa virheiden diagnosointia testauksen aikana.

3.2.3 Virheiden hallinta ja käsittely

Projektissa hyödynnettiin NestJS:n sisäänrakennettua virheidenkäsittelymekanismia (Exception Filters). Virheidenhallinta kattaa tyyppilliset tilanteet, kuten olemattomien resurssien haun (NotFoundException), autentikointivirheet kuten väärä salasana (UnauthorizedException) ja auktorisointivirheet (ForbiddenException). Virhevasteet noudattavat pääosin NestJS:n oletusmuotoa, joka sisältää statusCode, message ja error -kentät, antaen selkeän tiedon virheen syystä. TypeScriptin käyttäminen ohjelmointikielen auttaa myös jo itsessään poistamaan muuttujien tyypityksiin liittyviä virheitä, joita kehitystyössä mahdollisesti olisi esiintynyt enemmän, jos toteutus olisi tehty JavaScriptillä.

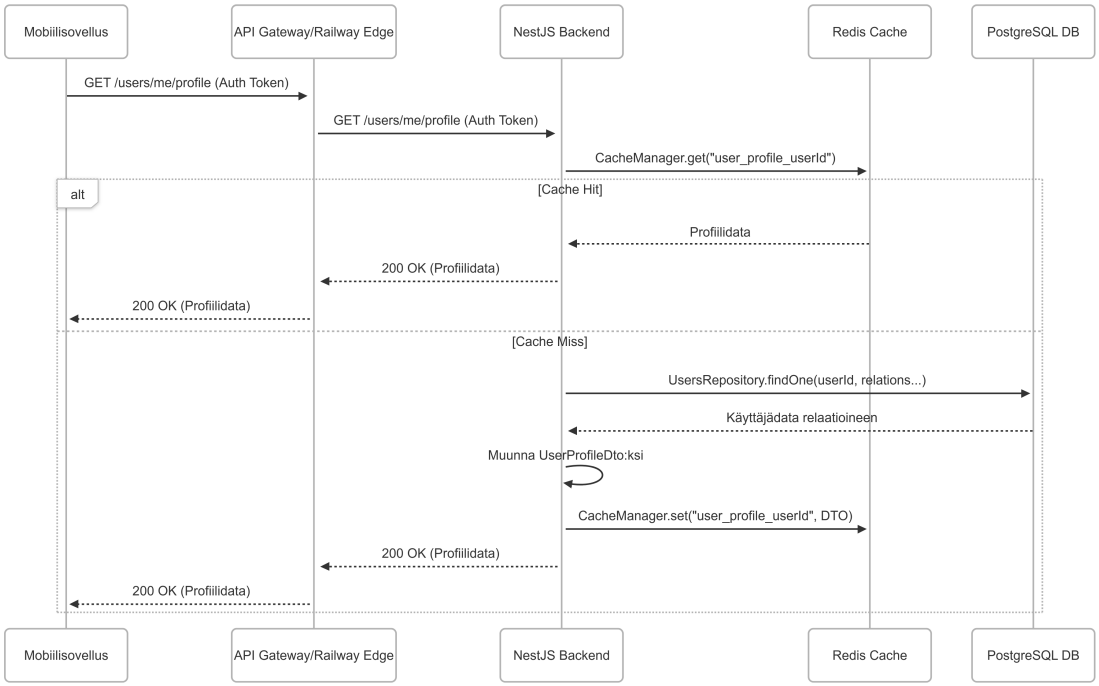
Kehityksen aikana ilmeni ja korjattiin myös useita TypeScript-tyypitysjärjestelmään liittyviä virheitä, jotka olisivat voineet johtaa ajonaikaisiin ongelmiin. Esimerkiksi TypeORM:n create- ja preload-metodien odottama DeepPartial<Entity>-tyyppi osoittautui tiukaksi null-arvon käsittelyn suhteen valinnaisissa relaatioissa (location?: Location), mikä vaati logiikan muokkaamista niin, että null-arvo asetettiin relaatiolle vasta preload-kutsun jälkeen ennen save-kutsua (update-metodissa) tai että null-arvoa sisältävää ominaisuutta ei välitetty lainkaan create-metodille. Myös ESLint-työkalun raportoimat varoitukset, kuten käyttämättömät muuttujat (no-unused-vars) ja epäturvalliset tyyppimääritykset (no-unsafe-assignment, no-unsafe-call, no-unsafe-member-access) liittyen erityisesti any-tyypin käyttöön välimuisti-instanssin (CacheManager) tai Passport.js:n req.user-olion käsittelyssä, korjattiin tarkentamalla tyyppejä (esimerkiksi verifyAsync<JwtPayload>) tai muuttamalla ESLint-sääntöjä (varsIgnorePattern: "^_"). Nämä korjaukset paransivat koodin tyyppiturvallisuutta ja vähensivät potentiaalisten ajonaikaisten virheiden määrää.

3.2.4 Suorituskyky

Koko sovelluksen läpi, erityisesti Service-kerroksessa tapahtuvissa tietokanta- ja välimuistioperaatioissa, hyödynnettiin async/await-syntaksia. Tämä varmistaa, että Node.js:n yksisäikeinen tapahtumasilmukka ei jää jumiin odottamaan hitaita I/O-operaatioita, mikä on kriittistä sovelluksen reagointikyvyn ja samanaikaisten pyyntöjen käsittelykyvyn kannalta.

Konkreettisena optimointitoimenpiteenä toteutettiin Redis-välimuistitus (@nestjs/cache-manager ja cache-manager-redis-yet) /users/me/profile-päätepisteelle. UsersService.findUserProfile-metodi toteuttaa "cache-aside"-mallin, jossa ensin tarkistetaan, löytyykö data käyttäjäkohtaisella avaimella (user_profile_{userId}) välimuistista. Jos löytyy (cache hit), data palautetaan suoraan sieltä. Jos ei (cache miss), data haetaan tietokannasta (joka sisältää käyttäjän perustiedot, ryhmät ja sijainnit yhdellä optimoidulla TypeORM-kyselyllä hyödyntäen relations-optiota), muunnetaan UserProfileDto-muotoon, tallennetaan Redis-

välimuistiin määritellyllä elinajalla (TTL), ja palautetaan asiakkaalle. Alla oleva kuvio 4 esittää tätä prosessia sekvenssikaavion avulla.



Kuvio 4. Esimerkki profiilidatan hakuprosessista

Tämä vähentää merkittävästi tietokantakuormaa toistuvissa profiilihauissa. Välimuistin invalidointi (cacheManager.del) on implementoitu käyttäjätietojen päivityksen (UsersService.update) yhteyteen, ja vastaava logiikka tulisi lisätä myös ryhmien, sijaintien ja muistiinpanojen muokkausoperaatioihin, jos niiden muutosten halutaan heijastuvan välittömästi profiilidataan.

TypeORM:n relaatioiden latauksessa (relations-optio find/findOne-metodeissa) pyrittiin lataamaan vain kussakin operaatiossa tarvittavat datat (eager loading by demand) N+1-kyselyongelman välttämiseksi. Esimerkiksi /users/me/profile lataa kerralla käyttäjän, tämän userGroups (ja niiden group-tiedot) sekä locations ja notes (ja niiden location-tiedot). Vaikka indeksejä ei luotu tässä POC-vaiheessa TypeORM-entiteettien kautta (pl. pääavaimet ja @Column({unique:true}) -määritetyt sarakkeet), tuotantoympäristössä tietokannan suorituskyvyn kannalta olisi kriittistä lisätä indeksit usein haettaviin sarakkeisiin (esimerkiksi User.email, vierasavaimet). Listoille, jotka voivat kasvaa suuriksi (esimerkiksi käyttäjän muistiinpanot tai sijainnit), tulisi jatkokehityksessä toteuttaa paginointi, jotta kaikkea dataa ei ladata kerralla.

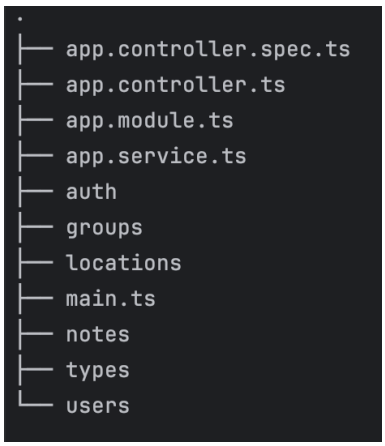
3.3 Koodin laatu ja ylläpito

Ohjelmiston toiminnallisten vaatimusten täyttämisen ohella lähdekoodin laatu ja sen pitkän aikavälin ylläpidettävyys ovat keskeisiä tekijöitä projektin onnistumisen ja elinkaaren kannalta. Laadukas koodi on ymmärrettävää, testattavaa ja muokattavaa, mikä vähentää virheiden syntymisen todennäköisyyttä ja helpottaa järjestelmän jatkokehitystä sekä uusien ominaisuuksien lisäämistä. Hyvä ylläpidettävyys puolestaan takaa, että ohjelmistoa voidaan tehokkaasti mukauttaa muuttuviin tarpeisiin ja korjata mahdolliset ongelmat ilman kohtuutonta työmäärää tai riskiä uusien virheiden syntymisestä.

Koodin laadun ja ylläpidettävyyden varmistaminen ei tapahdu itsestään, vaan se edellyttää tietoista panostusta ja sovittujen käytäntöjen noudattamista läpi koko kehitysprosessin. Tässä luvussa käsitellään niitä keskeisiä periaatteita, menetelmiä ja työkaluja, joita tässä projektissa on hyödynnetty korkean laadun tavoittelemiseksi ja ylläpidon helpottamiseksi. Tarkastelu kattaa niin konkreettiset koodauskäytänteet ja -konventiot, laadunvarmistukseen liittyvät katselmointi- ja testausprosessit kuin systemaattisen versionhallinnankin.

3.3.1 Käytänteet

Projekti noudattaa NestJS-kehyksen asettamia konventioita ja parhaita käytäntöjä. Keskeistä on modulaarinen arkkitehtuuri, jossa sovelluksen eri toiminnallisuudet on eristetty omiin loogisiin kokonaisuuksiinsa. Kuva 3 esittää projektin hakemistorakenteen, joka ilmentää tätä modulaarisuutta. Sovelluksen päämoduuli (app.module.ts) kokoaa yhteen ominaisuusmoduulit, kuten autentikaation (auth), käyttäjienhallinnan (users), ryhmien (groups), sijaintien (locations) ja muistiinpanojen (notes) käsittelyn. Tämä rakenne edistää koodin organisointia, uudelleenkäytettävyyttä ja ylläpidettävyyttä, kuten NestJS:n periaatteisiin kuuluu.



Kuva 3. Projektin modulaarinen rakenne

Modulaarisen rakenteen lisäksi projektissa hyödynnettiin johdonmukaisesti NestJS:n sisäänrakennettua Dependency Injection (DI) -mekanismia. Tämä tarkoittaa, että esimerkiksi Service-luokat injektoitiin Controllereihin ja TypeORM Repositoryt sekä CacheManager injektoitiin Serviceihin konstruktorin kautta (@Injectable, @InjectRepository, @Inject(CACHE_MANAGER)). Tämä kääntää riippuvuuksien hallinnan vastuun kehykselle (Inversion of Control), mikä vähentää komponenttien välistä kytkentää (loose coupling) ja parantaa merkittävästi koodin testattavuutta (mahdollistaa mock-objektien käytön testeissä) ja ylläpidettävyyttä.

Staattisen tyypityksen hyödyt konkretisoituivat useassa vaiheessa. Esimerkiksi siirtyminen numeerisista ID:istä UUID-merkkijonoihin käyttäjätunnisteissa paljasti välittömästi tyyppivirheet eri puolilla koodikantaa, joita olisi ollut vaikeampi havaita dynaamisesti tyypitetyllä kielellä. strictNullChecks: true -asetus pakotti käsittelemään mahdollisesti puuttuvat arvot (esimerkiksi valinnainen relaatio Note-entiteetin ja Location-entiteetin välillä) eksplisiittisesti, estäen potentiaalisia undefined is not a function -tyyppisiä ajonaikaisia virheitä. Myös geneeristen tyyppien käyttö (esimerkiksi Repository<User>, cacheManager.get<UserProfileDto>) paransi koodin luettavuutta ja turvallisuutta.

TypeORM:n dekoraattoripohjainen entiteettien määrittely mahdollisti tietokantaskeeman pitämisen synkronissa TypeScript-luokkien kanssa. UserGroup-liitosentiteetin käyttö ManyToMany-suhteessa User:n ja Group:n välillä tarjosi joustavan tavan lisätä relaatioon liittyvää metadataa (isAdmin-lippu). UUID-pääavaimien valinta (@PrimaryGeneratedColumn('uuid')) tehtiin tulevaisuuden skaalautuvuutta ja hajautettuja järjestelmiä silmällä pitäen, vaikka se vaatikin tarkempaa tyypityksen hallintaa (ParseUUIDPipe). onDelete-asetusten (CASCADE/SET NULL) harkittu käyttö relaatioissa on tärkeä osa tietokannan integriteetin ylläpitoa. Soft delete -strategia (isActive-lippu Group-entiteetissä) valittiin mahdollistamaan datan palauttaminen ja historiatiedon säilyttäminen verrattuna fyysiseen poistoon.

JWT-strategiassa käytettiin lyhytikäistä (60s) accessToken-tunnistetta varsinaiseen API-kutsujen autentikointiin ja pidempään voimassa olevaa (7d) refreshToken-tunnistetta uuden pääsytunnisteen hankkimiseen. Refresh tokenien validointi perustui niiden löytymiseen tietokannasta ja vanhenemisajan tarkistukseen, ja käytetyt refresh tokenit poistettiin kierron yhteydessä turvallisuuden parantamiseksi (token rotation). bcryptjs-kirjaston hash- ja compare-metodeja käytettiin asianmukaisesti. Auktorisointi resurssien omistajuuden perusteella (esimerkiksi käyttäjä voi muokata vain omia muistiinpanojaan) implementoitiin Service-kerroksessa lisäämällä userId osaksi tietokantakyselyiden WHERE-ehtoja. Ryhmäkohtainen admin-rooli tarkistettiin lukemalla isAdmin-lippu UserGroup-liitosentiteetistä ennen hallinnollisten toimintojen sallimista. ValidationPipe:n whitelist: true ja forbidNonWhitelisted: true

-asetukset lisäsivät turvallisuutta estämällä ylimääräisten, ei-toivottujen kenttien lähettämisen API-pyyntöjen mukana. Arkaluontoisten tietojen (kuten salasanatiivisteen) poistaminen vastauksista toteutettiin systemaattisesti Service-kerroksessa ennen datan palauttamista Controllerille.

Kaikki Service-metodit, jotka suorittivat tietokanta- (findOne, find, save, delete, preload) tai välimuistioperaatioita (cacheManager.get, set, del), määriteltiin async-avainsanalla ja niiden sisällä käytettiin await-avainsanaa odottamaan Promise-olioiden ratkeamista. Tämä on kriittistä Node.js-ympäristössä, jotta sovelluksen pääsäie ei jää jumiin odottamaan hitaita I/O-operaatioita.

3.3.2 Katselmointi

Merkittävä osa laadunvarmistuksesta tapahtui iteratiivisen virheenkorjauksen kautta. Esimerkiksi käyttöönotto Railway-alustalle paljasti konfiguraatio-ongelmia (kuten PostgreSQL:n initdb-virheen ja Redis-yhteysmuuttujien virheellisen asettamisen), jotka diagnosoitiin ja korjattiin järjestelmällisesti alustan lokitietoja ja dokumentaatiota hyödyntäen.

Vaikka muodollista vertaiskatselmointia ei suoritettu, projektin laadunvarmistus oli jatkuva prosessi. TypeScript-kääntäjä ja ESLint toimivat automaattisina "katselmoijina", jotka pakottivat noudattamaan kielen sääntöjä ja projektin koodausstandardeja. Esimerkiksi noUnusedLocals- tai noUnusedParameters-tyyppiset auttoivat pitämään koodin siistinä. Postman-testaus toimi keskeisenä manuaalisen validoinnin muotona: jokaisen uuden päätepisteen tai muutoksen jälkeen ajettiin läpi relevantit testitapaukset varmistaen sekä onnistuneet ("happy path") että virhetilanteiden ("sad path") toimivuus. Kun testaus paljasti virheen (esimerkiksi 4xx/5xx HTTP-statuskoodi, väärä vastausdata), lokien analysointi (Docker/Railway) oli ensisijainen tapa jäljittää ongelman juurisyy Service- tai jopa tietokantakerrokseen asti. Refaktorointia tehtiin tarpeen mukaan; esimerkiksi kun huomattiin, että käyttäjän salasana palautui useassa päätepisteessä, logiikka keskitettiin UsersService:n metodeihin käyttämään Omit<User, 'password'> -tyyppiä. Samoin AuthService:n validateUser-metodi refaktoroitiin käyttämään erillistä UsersService.findForAuth-metodia, kun huomattiin konflikti salasanan palauttamisen ja yleiskäyttöisen findByEmail-metodin välillä.

3.3.3 Versiohallinta

Git ja GitHub valittiin työkaluiksi projektin versiohallintaan. Gitin käyttö mahdollisti projektin kehityksen dokumentoinnin commit-kohtaisesti. Pyrkimyksenä oli tehdä atomisia committeja, joissa jokainen commit sisälsi yhden loogisen muutoksen (esimerkiksi uuden ominaisuuden lisäys, virheenkorjaus, refaktorointi) ja kuvaavan commit-viestin. Tämä helpottaa

muutoshistorian ymmärtämistä ja mahdollisten regressioiden jäljittämistä (git blame, git log). Feature branch -työnkulun avulla uudet ominaisuudet (locations, notes) tai kokeilut voitiin kehittää erillään vakaasta main-haarasta. Muutokset yhdistettiin main-haaraan vasta testauksen jälkeen. Versionhallintaan sisällytettiin kaikki projektin rakentamiseen ja ajamiseen tarvittava konfiguraatio, mukaan lukien Dockerfile, docker-compose.yml kehitysympäristölle, package.json ja package-lock.json riippuvuuksien hallintaan, tsconfig.json TypeScript-asetuksille, eslint.config.mjs linttausasetuksille sekä Makefile Docker-komentojen helpottamiseksi. .gitignore-tiedosto varmisti, että paikalliset riippuvuudet (node_modules), käännetyt tiedostot (dist), ja ympäristökohtaiset salaisuudet (.env) pysyivät versionhallinnan ulkopuolella.

3.4 Käyttöönotto ja julkaisu

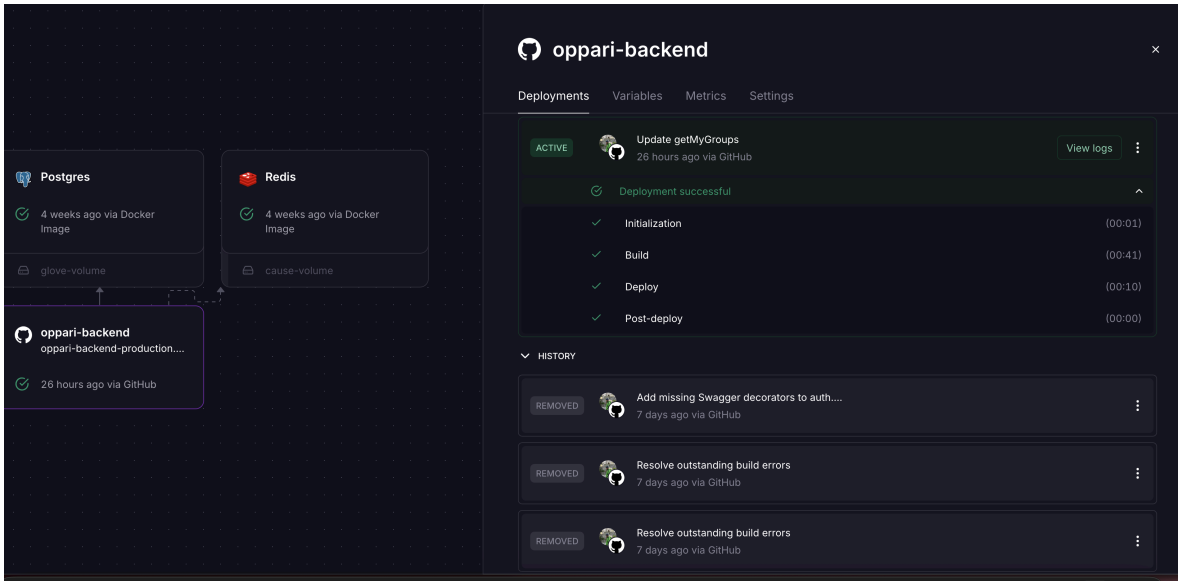
Ohjelmistokehitysprojektin kulminaatiopiste on sovelluksen käyttöönotto ja julkaisu, jolloin valmis tuote saatetaan loppukäyttäjien saataville ja sen operatiivinen toiminta tuotantoympäristössä alkaa. Tämä vaihe on kriittinen, sillä se edellyttää paitsi teknistä suorituskykyä myös huolellista suunnittelua ja prosessien hallintaa varmistaakseen sovelluksen vakaan, turvallisen ja tehokkaan toiminnan. Onnistunut käyttöönotto ja selkeä julkaisustrategia ovat avainasemassa sovelluksen menestyksekkään elinkaaren ja käyttäjätyytyväisyyden kannalta.

Tässä luvussa tarkastellaan niitä menetelmiä, teknologioita ja prosesseja, jotka liittyvät tämän projektin taustajärjestelmän käyttöönottoon ja julkaisuun. Käsiteltävät aiheet kattavat palvelinympäristön valinnan ja konfiguraation, itse käyttöönottoprosessin vaiheet ja automaation, sovelluksen ylläpitoon ja valvontaan liittyvät käytännöt sekä järjestelmän skaalautuvuuteen varautumisen. Näiden osa-alueiden huolellinen suunnittelu ja toteutus ovat välttämättömiä toimivan ja luotettavan palvelun tarjoamiseksi.

3.4.1 Palvelinympäristö

Projektin palvelinympäristö hyödynsi Docker-konttiteknologiaa johdonmukaisesti sekä paikallisessa kehityksessä että lopullisessa Railway.app-pilvipalveluun (PaaS) toteutetussa käyttöönottoympäristössä. Paikallisessa kehityksessä docker-compose.yml-tiedoston avulla määriteltiin paikallinen monikonttinen ympäristö, joka sisälsi NestJS-sovelluksen (nestjs-app), PostgreSQL-tietokannan (postgres-db, postgres:14-imagesta) ja Redis-välimuistin (redis-cache, redis:alpine-imagesta). Palvelut kommunikoivat Docker Composen luomassa bridge-verkossa käyttäen palvelunimiä hostnameina, ja data persistentoitiin nimetyillä volyymeilla (postgres-data, redis-data). Kuva 4 näyttää Railway-alustan käyttöliittymän, jossa nähdään projektin keskeiset palvelut: oppari-backend-sovellus, PostgreSQL-

tietokanta (Postgres) ja Redis-välimuisti (Redis). Sovellus on otettu käyttöön GitHubista, kun taas tietokanta ja välimuisti hyödyntävät Railwayn hallinnoituja palveluita. Kuvan oikea laita näyttää myös oppari-backend-palvelun käyttöönottohistorian ja viimeisimmän onnistuneen käyttöönoton vaiheet.



Kuva 4. Näkymä Railwayn hallintapaneelista

Railway-alustalla hyödynnettiin sen tarjoamia hallinnoituja palveluita (managed services) PostgreSQL:lle ja Redisille. Tämä strategia valittiin ylläpidon, varmuuskopioinnin ja skaalautuvuuden helpottamiseksi. NestJS-sovellus ajettiin Docker-kontissa, jonka Railway rakensi automaattisesti projektin Dockerfile:n pohjalta. Yhteydet sovelluksen ja hallinnoitujen palveluiden välillä konfiguroitiin Railwayn ympäristömuuttujilla käyttäen alustan dynaamisia palveluviittauksia (${{ Postgres.DATABASE_URL }}, ${{ Redis.REDIS_URL }}), jotka sovellus luki ConfigService:n avulla. PostgreSQL:n initdb-yhteensopivuusongelma Railwayn volyymien kanssa ratkaistiin PGDATA-ympäristömuuttujalla. Sovelluksen Dockerfile toteutti monivaiheisen buildin (multi-stage build): builder-vaihe (node:18) asensi riippuvuudet ja käänsi koodin, kun taas lopullinen tuotantovaihe (node:18-slim) sisälsi vain ajoympäristön ja tarvittavat artefaktit, minimoiden imagen koon ja parantaen tietoturvaa.

3.4.2 Käyttöönottoprosessi

Käyttöönotto Railway-alustalle perustuu Git-pohjaiseen työnkulkuun. Kun muutokset työnnetään projektin GitHub-repositoryyn (main-haaraan), Railway havaitsee muutokset automaattisesti ja käynnistää uuden build- ja deploy-prosessin. Build-prosessi hyödyntää projektin juuressa olevaa Dockerfile:a rakentaakseen NestJS-sovelluksen tuotantoimagen (multi-stage build). Onnistuneen buildin jälkeen Railway korvasi ajossa olevan

sovelluskontin uudella versiolla ja injektoi siihen määritellyt ympäristömuuttujat ennen sovelluksen käynnistämistä (npm run start:prod).

Projektissa käytetty TypeORM:n synchronize: true -asetus ei sovellu tuotantoon. Tuotantokelpoinen prosessi vaatisi TypeORM CLI -migraatioiden käyttöönoton tietokantaskeeman hallittuun päivittämiseen osana deploy-prosessia. Vaikka tämä prosessi oli automatisoitu Git-pushista, täydellinen CI/CD-putki vaatisi lisäksi automaattisten testien (yksikkö-, integraatio-, ja API-) ajamisen ennen build- ja deploy-vaiheita.

3.4.3 Ylläpito ja valvonta

Sovelluksen toimintaa ja mahdollisia virheitä seurattiin käyttöönoton ja testauksen aikana hyödyntämällä Railwayn tarjoamaa reaaliaikaista lokivirtaa suoraan alustan käyttöliittymästä. Sovelluksen sisäinen lokitus (@nestjs/common Logger) tarjosi lisätietoa esimerkiksi välimuistioperaatioiden onnistumisesta. Tuotantoympäristössä valvontaa tulisi laajentaa kattavammilla työkaluilla ja hälytyksillä. Railway tarjoaa myös sähköposti-ilmoituksia mahdollisista virhetilanteista ja yllättävistä kuluista.

Kehityksen ja käyttöönoton aikana valvonta perustui vahvasti Railwayn reaaliaikaisen lokivirran manuaaliseen seurantaan. Tämän avulla diagnosoitiin onnistuneesti mm. tietokanta- ja Redis-yhteysvirheet (ECONNREFUSED, Redis host or port not configured), PostgreSQL:n initdb-ongelma sekä varmistettiin sovelluksen moduulien oikea alustus. NestJS:n sisäistä Logger-palvelua käytettiin lisäämään sovelluskohtaisia logeja esimerkiksi välimuistin toiminnan (Serving... from cache / Workspaceing... from DB) seuraamiseksi. Vaikka tämä oli riittävää proof-of-concept-vaiheessa, tunnistettiin tarve systemaattisemmalle tuotantoympäristön valvonnalle, johon sisältyy seuraavat toiminnot:

- Lokien keskittäminen: Otetaan käyttöön keskitetty lokienhallintajärjestelmä.
- Metriikoiden seuranta: Lisätään työkalu seuraamaan sovelluksen suorituskykyä.
- Terveystarkistukset: Toteutetaan /health-päätepiste.
- Hälytykset: Määritellään automaattiset hälytykset kriittisille virheille ja poikkeamille.
- Päivitykset: Luodaan prosessi riippuvuuksien säännölliseen tarkistamiseen ja päivittämiseen.

3.4.4 Skaalautuvuus

Nykyisessä tilassa on yksi sovellusinstanssi ja tietokantainstanssi Docker Composen kautta. NestJS-sovellus itsessään on tilaton (stateless), mikä mahdollistaa helpon

horisontaalisen skaalautumisen. Sovelluksesta voidaan useita identtisiä instansseja kuormantasajaan (load balancer) takana. Jos tarve on skaalata PostgreSQL-tietokanta, tämä voidaan toteuttaa esimerkiksi lukureplikoilla (read replicas) tai jakamalla dataa useammalle palvelimelle (sharding).

Redis-välimuistin käyttöönotto on konkreettinen toimenpide, joka parantaa sovelluksen skaalautuvuutta vähentämällä suoraa tietokantakuormitusta. Railway-alusta tarjoaa myös mekanismeja sekä sovelluspalvelun instanssien määrän horisontaaliseen skaalaukseen (useampi kontti) että tietokanta- ja Redis-palveluiden resurssien (muisti, CPU) vertikaaliseen skaalaukseen tarpeen niin vaatiessa.

4 Yhteenveto ja pohdinta

4.1 Tulokset ja arvointi

Tämän opinnäytetyön ensisijaisena tavoitteena oli suunnitella ja toteuttaa toimiva proof-of-concept (POC) -tason taustajärjestelmä mobiilisovellukselle, ja tämä tavoite saavutettiin onnistuneesti. Projektin tuloksena syntyi modulaarinen ja laajennettavissa oleva NestJS/TypeScript-sovellus, joka sisältää keskeiset backend-toiminnallisuudet ja kykenee kommunikoimaan ulkoisten asiakasohjelmien kanssa REST API -rajapinnan kautta.

Teknologiavalinnat osoittautuivat pääosin onnistuneiksi POC-toteutukseen. NestJS:n vahvat konventiot ja TypeScriptin tyyppiturvallisuus nopeuttivat kehitystä ja vähensivät virheitä. TypeORM helpotti tietokantakäsittelyä, vaikka sen tyyppijärjestelmä ja relaatioiden konfigurointi vaativatkin perehtymistä. Suurimmat haasteet liittyivät ympäristöjen konfigurointiin (Docker, Railwayn hallinnoidut palvelut, PGDATA-ongelma, ympäristömuuttujat), TypeORM:n ja TypeScriptin yhteispelin yksityiskohtiin (DeepPartial, null/undefined-käsittely relaatioissa) sekä autentikointilogiikan (JWT payload, Guardien käyttö) virheenjäljitykseen. Näiden haasteiden ratkaiseminen oli merkittävä osa oppimisprosessia, syventäen ymmärrystä backend-kehityksen tyypillisistä ongelmakohdista ja niiden ratkaisutavoista. Vaikka virallista katselmointia ei ollut, jatkuva testaus ja iteratiivinen virheenkorjaus toimivat tehokkaana laadunvarmistusmenetelmänä. POC-tavoitteet saavutettiin, ja luotiin vankka perusta jatkokehitykselle.

4.2 Jatkokehitysmahdollisuudet

Vaikka opinnäytetyössä toteutettu proof-of-concept (POC) -taustajärjestelmä kattaa keskeiset perustoiminnallisuudet, on useita osa-alueita, joita voitaisiin jatkokehittää sovelluksen viemiseksi kohti tuotantovalmiutta ja laajempaa toiminnallisuutta:

- OAuth2-integraation viimeistely: Nykyiset Google- ja Facebook-autentikoinnin päätepisteet ja service-metodit ovat pääosin aihioita. Täysi implementaatio vaatii passport-google-oauth20- ja passport-facebook-strategioiden käyttöönoton, käyttäjien luonti- ja linkityslogiikan (AuthService.validateOAuthUser) viimeistelyn sekä erityisesti sellaisten tilanteiden käsittelyn, joissa OAuth-tarjoaja ei palauta sähköpostiosoitetta.

- Kuvien käsittely: Sekä Location- että Note-entiteeteissä on kenttä (pictureUrl) kuvalle, mutta varsinainen kuvien latausmekanismi puuttuu. Jatkokehityksessä tulisi implementoida tiedostojen vastaanotto backendissä (esimerkiksi @nestjs/platform-express ja multer-kirjasto), kuvien prosessointi (tarvittaessa koon muuttaminen,

optimointi) ja tallennus joko paikalliseen tiedostojärjestelmään tai pilvitallennuspalveluun (kuten AWS S3, Google Cloud Storage, Azure Blob Storage). API palauttaisi tallennetun kuvan URL-osoitteen.

- Kattavampi testaus: Projektista puuttuvat tällä hetkellä formaalit yksikkö- ja integraatiotestit. Jatkossa tulisi kirjoittaa testejä @nestjs/testing-kirjastolla Service-luokkien liiketoimintalogiikalle (yksikkötestit, käyttäen mock-repositoryja) ja Controller-päätepisteiden toiminnallisuudelle (integraatio-/e2e-testit, mahdollisesti erillisellä testitietokannalla tai testcontainers-työkalulla). Myös Postman-kokoelman testit voitaisiin automatisoida Newmanilla.

- CI/CD-putken rakentaminen: Täysin automatisoitu CI/CD-putki (esimerkiksi GitHub Actions) nopeuttaisi ja vakauttaisi kehitystä ja käyttöönottoa. Putki voisi sisältää vaiheet koodin linttaukselle, testien ajamiselle, Docker-imagen rakentamiselle ja pushaamiselle konttirekisteriin sekä lopulta käyttöönotolle (deploy) kohdeympäristöön (esimerkiksi Railway).

- Tietokantamigraatiot: Siirtyminen TypeORM:n synchronize: true -asetuksesta hallittuihin tietokantamigraatioihin (TypeORM CLI) on välttämätöntä tuotantoympäristössä tietokantaskeeman turvalliseksi ja hallituksi päivittämiseksi versioiden välillä. Migraatiotiedostot tallennettaisiin versionhallintaan.

- Ylläpito ja valvonta: Tuotantoympäristöä varten tulee implementoida kattava lokitus (keskitetty, strukturoitu), monitorointi (APM, metriikat) ja hälytysjärjestelmä sovelluksen tilan ja suorituskyvyn seuraamiseksi ja ongelmatilanteisiin reagoimiseksi. /health-päätepisteen lisääminen on myös suositeltavaa.

- Välimuistituksen laajentaminen: Nyt implementoitua Redis-välimuistitusta voitaisiin laajentaa koskemaan muitakin usein luettavia GET-päätepisteitä (esimerkiksi GET /locations, GET /groups) ja varmistaa kattava invalidointilogiikka kaikille välimuistitettua dataa muokkaaville operaatioille (POST, PATCH, DELETE).

- Edistyneempi auktorisointi: Mikäli sovelluksen vaatimukset kasvavat, nykyistä yksinkertaista (isAdmin lipun tarkistus) auktorisointia voitaisiin laajentaa monipuolisempaan rooli- tai oikeuspohjaiseen malliin (RBAC/ABAC) esimerkiksi NestJS:n Guardien avulla.

- Reaaliaikaisuus (WebSockets): Jos sovellukseen halutaan reaaliaikaisia ominaisuuksia, kuten pikaviestejä tai notifikaatioita, WebSockets-tuki voitaisiin implementoida NestJS:n Gateway-abstraktion avulla.

- Sovelluskohtaiset ominaisuudet: Luonnollisesti keskeisin jatkokehitysalue on varsinaisten, sovelluksen ydinideaan liittyvien toiminnallisuuksien lisääminen ja syventäminen.

Kuten edellä kuvatut jatkokehitysmahdollisuudet osoittavat, tämän opinnäytetyön tuloksena syntynyt vankka perusta tarjoaa erinomaisen lähtökohdan sovelluksen potentiaalin täysimittaiseen hyödyntämiseen ja sen kehittämiseen entistä monipuolisemmaksi ja käyttäjäystävällisemmäksi kokonaisuudeksi.

Lähteet

Abba, I. 2022. What is an ORM – The Meaning of Object Relational Mapping Database Tools. Viitattu 24.4.2025. Saatavissa https://www.freecodecamp.org/news/what-is-an-orm-the-meaning-of-object-relational-mapping-database-tools/

Amazon Web Services. 2025. What is SQL (Structured Query Language)? Viitattu 10.5.2025. Saatavissa https://aws.amazon.com/what-is/sql/

Ambler, S.W. 2022a. Overcoming the Object-Relational Impedence Mismatch. Viitattu 10.4.2025. Saatavissa https://agiledata.org/essays/impedanceMismatch.html

Ambler, S.W. 2022b. Mapping Objects to Relational Databases: O/R Mapping In Detail. Viitattu 10.4.2025. Saatavissa https://agiledata.org/essays/mappingObjects.html

Carlson, B. 2025. Node-postgres. Viitattu 5.5.2025. Saatavissa https://node-postgres.com/

Dua, A. 2024. What Is Nest.JS? Why Should You Use It? Viitattu 1.5.2025. Saatavissa https://www.turing.com/blog/what-is-nest-js-why-use-it#_why_use__nest._j_s?

Ebenezer, A. 2024. Mastering Data Validation in NestJS: A Complete Guide with Class-Validator and Class-Transformer. Viitattu 8.5.2025. Saatavissa https://medium.com/@ahureinebenezer/mastering-data-validation-in-nestjs-a-complete-guide-with-class-validator-and-class-transformer-02a029db6ecf

Euroopan parlamentin ja neuvoston asetus (EU) 2016/679, annettu 27 päivänä huhtikuuta 2016, luonnollisten henkilöiden suojelusta henkilötietojen käsittelyssä sekä näiden tietojen vapaasta liikkuvuudesta ja direktiivin 95/46/EY kumoamisesta (yleinen tietosuoja-asetus)

European Commission. 2025a. What does data protection ‘by design’ and ‘by default’ mean? Viitattu 13.5.2025. Saatavissa https://commission.europa.eu/law/law-topic/data-protection/rules-business-and-organisations/obligations/what-does-data-protection-design-and-default-mean_en

Garfield, L. 2025. What is .env? Viitattu 8.5.2025. Saatavissa https://upsun.com/blog/what-is-env-file/

Gavrilenko, A. 2023. The System Design Cheat Sheet - API Styles (REST, GraphQL, WebSocket, WebHook, RPC/gRPC, SOAP). Viitattu 15.4.2025. Saatavissa https://hackernoon.com/the-system-design-cheat-sheet-api-styles-rest-graphql-websocket-webhook-rpcgrpc-soap

Goldberg, J. 2022. Learning TypeScript. Sebastopol: O’Reilly Media, Inc.

Gough, J., Bryant, D., & Auburn, M. 2023. Mastering API Architecture. Sebastopol: O'Reilly Media, Inc.

Grigutytė, M. 2023. What is bcrypt and how does it work? Viitattu 1.5.2025. Saatavissa https://nordvpn.com/blog/what-is-bcrypt/

IBM. 2025a. What is three-tier architecture? Viitattu 1.5.2025. Saatavissa https://www.ibm.com/think/topics/three-tier-architecture

IBM. 2025b. What is Redis? Viitattu 10.5.2025. Saatavissa https://www.ibm.com/think/topics/redis

Internet Engineering Task Force. 2025. The OAuth 2.0 Authorization Framework. Viitattu 12.4.2025. Saatavissa https://datatracker.ietf.org/doc/html/rfc6749#section-1.1

Kumar, S. 2024. Securing Your Backend API: A Comprehensive Guide. Viitattu 13.5.2025. Saatavissa https://medium.com/codex/securing-your-backend-api-a-comprehensive-guide-9bf5e0166fd6

Microsoft. 2025. Tietokannan normalisoinnin kuvauksen perusteet. Viitattu 12.4.2025. Saatavissa https://learn.microsoft.com/fi-fi/office/troubleshoot/access/database-normalization-description

MongoDB. 2025. Understanding SQL vs NoSQL Databases. Viitattu 10.5.2025. Saatavissa https://www.mongodb.com/resources/basics/databases/nosql-explained/nosql-vs-sql

NestJS. 2025a. Database. Viitattu 10.4.2025. Saatavissa https://docs.nestjs.com/techniques/database

NestJS. 2025b. OpenAPI Introduction. Viitattu 10.4.2025. Saatavissa https://docs.nestjs.com/openapi/introduction

OneLogin. 2025. What is Multi-Factor Authentication (MFA) and How Does it Work? Viitattu 13.5.2025. Saatavissa https://www.onelogin.com/learn/what-is-mfa

Passport. 2025. Features. Viitattu 12.5.2025. Saatavissa https://www.passportjs.org/

Richards, M. 2022. Software Architecture Patterns – Second Edition. Sebastopol: O'Reilly Media, Inc.

SAP. 2025. Mikä on ohjelmointirajapinta (API)? Viitattu 18.4.2025. Saatavissa https://www.sap.com/finland/products/technology-platform/integration-suite/what-is-api.html

Swagger. 2025. API Documentation. Viitattu 20.4.2025. Saatavissa https://swagger.io/solutions/api-documentation/