Jere Janhunen & Mikael Thure

VIRRAKKEEN TAUSTAJÄRJESTEL-MÄN REFAKTOROINTI

Opinnäytetyö

Tekniikan ammattikorkeakoulututkinto

Ohjelmistotekniikan koulutus

2025

Kaakkois-Suomen
ammattikorkeakoulu

Kaakkois-Suomen ammattikorkeakoulu

Tutkintonimike	Insinööri (AMK)
Tekijä/Tekijät	Jere Janhunen ja Mikael Thure
Työn nimi	Virrakkeen taustajärjestelmän refaktorointi
Toimeksiantaja	Kaakkois-Suomen ammattikorkeakoulu
Vuosi	2025
Sivut	78
Työn ohjaaja(t)	Juha Ojala

TIIVISTELMÄ

Tässä opinnäytetyössä toteutettiin Virrake-sovelluksen taustajärjestelmän modernisointi refaktoroimalla olemassa oleva JavaScript-koodikanta. Projektin tavoitteena oli parantaa sovelluksen ylläpidettävyyttä, skaalautuvuutta, tietoturvaa ja jatkokehityksen tehokkuutta hyödyntämällä moderneja ohjelmistokehitysteknologioita ja -menetelmiä.

Refaktoroinnin yhteydessä sovelluksen ohjelmointikieli päivitettiin JavaScriptistä TypeScriptiin. TypeScriptin tuoma staattinen tyypitys paransi virheidenhallintaa ja selkeytti sovelluksen rakennetta. Aiemmin suoraan SQL-kyselyillä hoidettu tietokantayhteys korvattiin Sequelize-ORM-kirjastolla, mikä mahdollisti paremman abstrahoinnin, selkeämmän tietokantakerroksen hallinnan sekä helpotti muutosten tekemistä tietokantarakenteisiin.

Testauksen merkitys korostui projektissa, ja sovellukselle rakennettiin testikanta hyödyntäen Jest-testikirjastoa. Yksikkötestauksen avulla varmistettiin, että refaktorointi ei heikentänyt olemassa olevaa toiminnallisuutta ja että sovellus säilyi luotettavana koko kehitysprosessin ajan.

Modernisointityön tuloksena Virrake-sovelluksen taustajärjestelmästä muodostui selkeästi jäsennelty, turvallinen ja skaalautuva kokonaisuus, joka tukee sovelluksen tulevaa kehitystä ja ylläpitoa.

Asiasanat: JavaScript, TypeScript, ohjelmistokehitys, ohjelmistokirjastot, ohjelmistotestaus, tietotyypit

South-Eastern Finland University of Applied Sciences

Degree title	Bachelor of Engineering
Author (authors)	Jere Janhunen and Mikael Thure
Thesis title	Refactoring the backend of Virrake
Commissioned by	South-Eastern Finland University of Applied Sciences
Time	2025
Pages	78
Supervisor	Juha Ojala

ABSTRACT

In this thesis, the backend system of the Virrake application was modernized through the refactoring of the existing JavaScript codebase. The primary objective was to enhance the maintainability, scalability, security, and future development efficiency of the application by adopting modern software development technologies and methodologies.

As part of the refactoring process, the programming language was migrated from JavaScript to TypeScript. The introduction of static typing through TypeScript improved error handling, increased code clarity, and enhanced the overall structure of the system. The direct SQL queries previously used for database communication were replaced with the Sequelize ORM library, enabling greater abstraction, more structured database layer management, and facilitating changes to database schemas.

Testing was emphasized throughout the project, with a test suite established using the Jest testing library. Unit testing ensured that the refactoring efforts did not compromise existing functionalities, and that the application maintained its reliability during the development process.

As a result of the modernization, the backend system of the Virrake application was transformed into a well-structured, secure, and scalable architecture that better supports the application's future development and maintenance.

Keywords: data types, JavaScript, TypeScript, software development, software libraries, software testing

SISÄLLYS

1 JOHDANTO

Tässä opinnäytetyössä käsitellään Virrake-sovelluksen taustajärjestelmän koodin refaktoroinnin suunnittelua, kehittämistä ja toteuttamista pohjustamalla työssä käytettyjä tekniikoita ja teknologioita käymällä ensin läpi teoriaa näistä aiheista, jonka jälkeen käydään läpi toteutus. Opinnäytetyön tarkoituksena on esitellä, miten nykyaikaiset teknologiat ja modernit ohjelmistokehityksen menetelmät voidaan ottaa käyttöön koodipohjan refaktoroinnissa. Nykyisen koodipohjan osalta on havaittu useita kehityskohteita, kuten epäyhtenäiset funktiot, tietoturvahaavoittuvuudet sekä haastava tietokantaintegraatio (erityisesti PostgreSQL:n ja SQL-kyselyiden osalta).

Luvuissa 3–12 käsittelemme teoriaa kaikista työssä käyttämistämme tekniikoista ja teknologioista. Teoriaosiossa käymme läpi teoriaa muun muassa refaktoroinnista, Virrakkeesta, taustajärjestelmistä, tietokannoista ja versionhallinnasta. Toteutusosiossa käymme vaiheittain läpi, mitä työssä on saatu aikaan. Luvussa 13.1 kerromme taustajärjestelmän logiikan uudistamisesta ja/tai muokkaamisesta ja luvussa 13.2 käymme läpi eri vaiheet, joita kävimme läpi koodia refaktoroidessa. Luvussa 13.3 käsittelemme tietokantaintegraatioongelmien ratkomista, jonka jälkeen luvussa 13.4 kerromme, kuinka toteutimme testejä varmistaaksemme sovelluksen toimivuuden.

2 OPINNÄYTETYÖN TAVOITE

Virrakkeen backendin refaktoroinnin tavoitteena on parantaa ja yhtenäistää lähdekoodia sekä sen kehitystyötä hyödyntämällä moderneja ohjelmistokehityksen työkaluja ja kirjastoja. Nykyisessä versiossa merkittävimpiä havaittuja ongelmia ovat olleet yhteensopivuus tiettyjen tietokantatyyppien kanssa sekä tiettyjen tietoturvaperiaatteiden, kuten API-avainten tallennuksen, puutteellinen noudattaminen. Esimerkiksi SQL-kyselyt ovat usein kovakoodattuja, mikä vaikeuttaa tietokannan hallintaa ja tekee koodista vaikealukuisemman. Refaktoroinnilla pyritään erityisesti parantamaan yhteensopivuutta, suorituskykyä, tietoturvaa, ylläpidettävyyttä ja skaalautuvuutta.

Projektissa pyritään siirtymään pois suoraan koodissa olevista SQL-lauseista ja hyödyntämään ORM-ratkaisuja sekä modernisoimaan toteutusta esimerkiksi TypeScriptin avulla. Tavoitteena on saavuttaa uusi, aiempaa modulaarisempi ja kehittäjäystävällisempi koodipohja, joka on helpommin luettava ja ylläpidettävä. Tällä tavoin pyritään ratkaisemaan nykyisiä puutteita, kuten epäyhtenäiset funktiot ja tietoturvahaavoittuvuudet, samalla kun parannetaan suorituskykyä ja varmistetaan tulevien integraatioiden sujuvuus muiden järjestelmien kanssa.

Parannusten toteutumista arvioidaan testausmenetelmien avulla sekä refaktoroinnin aikana tehtyjen havaintojen perusteella. Mittareina käytetään muun muassa koodin luettavuuden, suorituskyvyn ja tietoturvallisuuden parantumista. Lisäksi arvioidaan, kuinka hyvin uudet ratkaisut tukevat sovelluksen laajennettavuutta ja mahdollistavat integraation muiden yritysten järjestelmien kanssa. Tärkeimpinä hyötyjinä ovat tulevat kehittäjät, joille jää aiempaa selkeämpi, helpommin ylläpidettävä ja laajennettava koodikanta, joka helpottaa uusien ominaisuuksien kehitystä ja sovelluksen jatkokehitystä.

Refaktorointi kohdistuu ensisijaisesti Virrakkeen backendin toiminnallisuuksiin. Vaikka Virrakkeen ohjauspaneeli ja itse sovellus eivät kuulu refaktoroinnin suoraan piiriin, niitä käytetään refaktoroinnin tehokkuuden ja toimivuuden testaamiseen. Näin varmistetaan, että tehdyt parannukset vaikuttavat positiivisesti koko sovelluksen käyttökokemukseen ja sen kykyyn vastata muuttuvan teknologiaympäristön haasteisiin.

3 REFAKTOROINTI

Koodin refaktorointi on prosessi, jossa ohjelmiston koodipohjan sisäistä rakennetta, luettavuutta ja ylläpidettävyyttä parannetaan muuttamatta sen ulkoista käyttäytymistä tai toiminnallisuutta. Tämän käytännön tavoitteena on parantaa koodin laatua ja vähentää teknistä velkaa järjestelemällä, yksinkertaistamalla tai optimoimalla koodia, jolloin siitä tulee tehokkaampi, modulaarisempi ja helpommin ymmärrettävä kehittäjille. (CodeSee s.a.)

Miksi refaktorointia tarvitaan?

Refaktorointi vähentää parannusten kustannuksia. Kun ohjelmistojärjestelmä menestyy, sitä on jatkuvasti kehitettävä, virheitä korjattava ja uusia ominaisuuksia lisättävä. Koodipohjan luonne vaikuttaa kuitenkin suuresti siihen, kuinka helppoa muutosten tekeminen on. Usein parannuksia lisätään päällekkäin tavalla, joka tekee jatkokehityksestä yhä vaikeampaa. Ajan myötä uusien muutosten tekeminen hidastuu huomattavasti. Tämän estämiseksi on tärkeää refaktoroida koodia, jotta lisäykset eivät johda tarpeettomaan monimutkaisuuteen. (Fowler s.a.)

Milloin kannattaa refaktoroida?

Refaktorointi ei ole lisäosa jokapäiväiseen ohjelmointirutiiniin. Sen tulee kuitenkin kohtaamaan jossain vaiheessa työn aikana. Sanotaan, että paras hetki refaktoroinnille on silloin, kun haluaa korvata tai lisätä koodia. Toinen hyvä ajankohta on loppuvaiheen tarkistuksen yhteydessä, koska se on yleensä suoraviivaista. (Sokolov 2021.)

4 VIRRAKE-SOVELLUS

Virrake alkoi alun perin vuonna 2018 Virtuaalinen rakentaminen -hankenimen alla. Tavoitteena oli etsiä uudenlaisia tapoja alueiden ja rakennusten visualisoimiseen ja tutkimiseen. Sovelluksen tarkoitus oli luoda yleinen alusta alueen tai rakennuksen nopealle pelillistämiselle. (Ojala 2025.)

Virrakkeen tarkoitus on siis ”pelillistää” rakennukset ja muut sijainnit ja luoda niistä digitaalinen versio, jota kutsutaan digitaaliseksi kaksoseksi (digital twin). Pelillistämisellä tarkoitetaan pelimekaniikoiden, kuten lentämisen ja ympäristön manipuloinnin lisäämistä kohteeseen, joka ei normaalisti ole peliympäristössä. Tällaisia ympäristöjä voivat esimerkiksi olla sairaala- tai koulurakennukset. (Ojala 2025.)

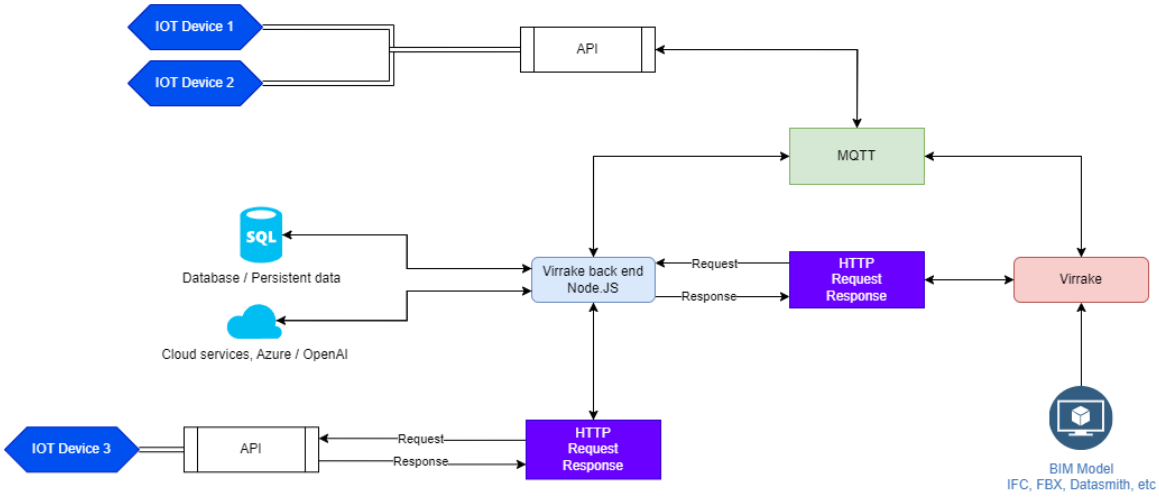
Virrake on rakennettu Unreal Engine 5 -pelimoottorin päälle, ja siihen voi ladata tuetussa tiedostomuodossa olevan 3D-mallin esimerkiksi rakennuksesta.

Virrake-ohjelmisto jatkaa pelimoottorin oletustuontiprosessia muun muassa automatisoimalla törmäystunnisteiden generoinnin asioihin, joihin sellaista tarvitsee, kuten seinät, lattia, katto, ja jättää ne pois asioista, joissa niitä ei tulisi olla, kuten ovet. Tuettuja tiedostomuotoja ovat muun muassa FBX, OBJ, STL ja IFC. Rakennusteknisissä tiedostomuodoissa, kuten IFC:ssä, Virrake-ohjelmistoon tuodaan 3D-mallin lisäksi myös rakennukseen liittyvä metadata. (Ojala 2025.)

Virrakkeen perustoimintoihin kuuluvat muun muassa toiminnot ovien ja kaappien avaamiseen, vipujen vetämiseen, nappien painamiseen, liukusäätimiin, kävelemiseen, juoksemiseen ja lentämiseen. Lisäksi mikä tahansa geometria voidaan muuntaa niin sanotuksi Virrake-objektiksi. Virrake-objekteja voidaan manipuloida sisäänrakennetuilla työkaluilla ja niiden sijaintia voidaan esimerkiksi vaihtaa reaaliajassa sovelluksen ollessa käynnissä. (Ojala 2025.)

Virrake tukee myös moninpeluuta ja virtuaalitodellisuutta. Näihin kuuluvat mukaan myös pelaajien välinen tekstipalsta ja voice chat, jotta eri asiakkaat ja eri alojen kehittäjät voivat jutella keskenään. Moninpelejä varten voi luoda kustomoitavan palvelimen, joka ei ole riippuvainen kolmannen osapuolen palveluista. (Ojala 2025.)

Virrake-sovellus on yhteydessä Virrakkeen omaan taustajärjestelmään (kuva 1), joka tarvittaessa kommunikoi kolmansien osapuolien kanssa, esimerkiksi erinäisten IOT-laitteiden kanssa. Taustajärjestelmä myös ylläpitää pysyvää dataa, kuten pelimaailmaan jätettyjä viestejä, palautteita sekä äänestyksiä. Taustajärjestelmä on olennainen osa Virrake-sovellusta. (Ojala 2025.)



Kuva 1. Virrakkeen pipeline-rakennekaavio (Ojala 2025)

5 BACKEND-OHJELMISTOKEHITYS

Codecademyn (Back-End Web Architecture s.a.) mukaan taustajärjestelmä eli backend on se osa koodista, joka suoritetaan palvelimella ja joka vastaanottaa pyyntöjä asiakkailta (clients). Se sisältää myös kaiken tarvittavan logiikan pyyntöjen vastaanottamiseen sekä vastausten tuottamiseen ja lähettämiseen takaisin asiakkaalle. Useimmiten backend sisältää nämä kolme asiaa:

- **Palvelin**: Vastaanottaa asiakkaiden pyyntöjä.
- **Sovellus**: Ohjelma, jota palvelimella suoritetaan. Kuuntelee pyyntöjä, hakee tietoja tietokannasta ja lähettää vastauksia.
- **Tietokanta**: Paikka, jossa dataa säilytetään ja käsitellään.

Palvelin

Yksinkertaistettuna palvelin on tietokone, joka on valmiina vastaanottamaan pyyntöjä. Tällainen tietokone voi olla juuri tähän tarkoitukseen tehty ja optimoitu tai aivan tavallinen päivittäisessä käytössä oleva tietokone, kunhan se on yhteydessä verkkoon. (Back-End Web Architecture s.a.)

Sovelluksen ydintoiminnot

Sovelluksen päätarkoitus on käsitellä HTTP-pyyntöjä (Hypertext Transfer Protocol) ja palauttaa niihin vastauksia. Tämä hoidetaan reittien avulla, jossa tietty HTTP-verbi (esimerkiksi GET tai POST) yhdistetään tiettyyn resurssiin URI (Uniform Resource Identifier). Reititys saadaan aikaiseksi, kun HTTP-verbin ja URIn reittiin lisätään mukaan pyyntö. Palvelin voi myös käyttää apunaan väliohjelmistoja (middleware), jotka suoritetaan palvelimen vastaanottaman pyynnön ja asiakkaalle lähetettävän vastauksen välillä. Middlewarella voi esimerkiksi lisätä käyttäjän tunnistautumistiedot tai tarkistaa käyttäjän oikeudet. (Back-End Web Architecture s.a.)

Vastaukset palvelimelta voivat olla useaa eri muotoa, kuten esimerkiksi HTML-tiedosto, JSON-data tai vain HTTP-tilakoodi. Yksi kenties kaikille tuttu tilakoodi on ”404 – Not Found”, joka tarkoittaa sitä, että käyttäjä on yrittänyt navigoida URIin, jota ei ole olemassa. Tämä on vain yksi esimerkki ja erilaisia tilakoodeja on olemassa paljon enemmän. Tilakoodit kertovat siitä, mitä tapahtui, kun palvelin vastaanotti pyynnön. (Back-End Web Architecture s.a.)

Tietokanta

Tietokannat ovat yleisesti käytössä webbisovellusten taustajärjestelmissä. Tietokannat tarjoavat mahdollisuuden datan pysyvään säilyttämiseen ja hakemiseen. Tämä vähentää palvelimen muistin kuormitusta ja mahdollistaa datan palauttamisen, vaikka palvelin kaatuisi. Monet pyynnöt asiakkailta voivat sisältää tietokantakyselyitä, joilla voidaan pyytää tietoa tietokannasta tai lähettää tietoa tallennettavaksi. (Back-End Web Architecture s.a.)

Rajapinta eli API

Web-rajapinta (API) on selkeästi määritelty joukko kommunikointimenetelmiä ohjelmiston komponenttien välillä. Tarkemmin sanottuna API on taustajärjestelmän luoma rajapinta, joka on joukko päätepisteitä ja näiden päätepisteiden tarjoamia resursseja. Web-rajapinnan toiminta määrittyy sen käsittelemien pyyntöjen tyypin perusteella, mikä taas määräytyy rajapinnan määrittelemien

reittien mukaan sekä sen perusteella, millaisia vastauksia asiakas voi odottaa saavansa näiltä reiteiltä. (Back-End Web Architecture s.a.)

Backend-kehittäjät luovat ja ylläpitävät järjestelmien toimintaa varmistaen, että verkkosovellukset toimivat tehokkaasti ja turvallisesti. Kehittäjät keskittyvät muun muassa datan käsittelyyn, tietoturvallisuuteen sekä palvelinpuolen toiminnallisuuteen. Taustajärjestelmää tehdessään kehittäjät tekevät tiivistä yhteistyötä frontend-kehittäjien, tuoteomistajien ja QA-tiimien (Quality Assurance) kanssa. (What Does a Front-End Developer Do? 2025.)

Courseran (What Does a Back-End Developer Do? 2024.) mukaan backend-kehittäjän päivittäisiin tehtäviin voivat kuulua verkkosovellusten rakentaminen ja ylläpito erilaisia työkaluja ja ohjelmointikieliä käyttäen. Keskeisimpiin työkaluihin ja teknologioihin lukeutuvat muun muassa nämä:

- **Ohjelmointikielet**: Python, PHP, JavaScript, Ruby, Java, C#
- **Frameworkit**: Laravel, Django, Spring, Ruby on Rails, Meteor, Node.js
- **Tietokannat**: MongoDB, MySQL, Oracle
- **Palvelimet**: Apache, NGINX, Lighttpd, Microsoft IIS

Lisäksi muun muassa koodin laadun varmistaminen, suorituskyvyn optimointi, testausprosessien suunnittelu ja toteutus sekä virheiden jäljitys ja korjaus voivat olla jokapäiväisiä tehtäviä. (What Does a Back-End Developer Do? 2024.)

REST ja CRUD

REST (Representational State Transfer) on ohjelmistoarkkitehtuurityyli, joka määrittelee standardit verkossa olevien erilaisten systeemien välille, jotta kommunikointi helpottuu. REST-mallissa asiakas ja palvelin toimivat toisistaan riippumattomina, jolloin koodia voidaan muuttaa kummallakin puolella ilman, että se vaikuttaa toisen toimintaan. Tämä mahdollistaa modulaarisuuden ja järjestelmän skaalautuvuuden. (What is REST? s.a.)

REST-arkkitehtuurin keskeinen ominaisuus on tilattomuus. Tämä tarkoittaa

sitä, että palvelimen ei tarvitsee säilyttää tietoa asiakkaan tilasta pyyntöjen välillä. Jokainen pyyntö sisältää kaiken tarvittavan tiedon, minkä ansiosta REST-sovellukset ovat luotettavia, suorityskykyisiä ja helposti hallittavia. (What is REST? s.a.)

Codecademyn (What is CRUD? s.a.) mukaan REST-ympäristössä on käytössä CRUD-toiminnot (Create, Read, Update, Delete), jotka toteutetaan HTTP-metodien avulla. CRUD on yleinen paradigma verkkosovellusten rakentamisessa, koska se tarjoaa kehittäjille selkeän ja muistettavan kehyksen toimintojen suunnitteluun. CRUD-toiminnot vastaavat seuraavia HTTP-menetelmiä REST-arkkitehtuurissa:

- **Create** (Luo) – POST
- **Read** (Lue) – GET
- **Update** (Päivitä) – PUT
- **Delete** (Poista) – DELETE

CRUD-toimintojen oikeaoppinen toteutus REST-ympäristössä takaa järjestelmän ennustettavuuden ja tietojen eheyden, sillä jokainen pyyntö vastaa tiettyä toiminnallisuutta. Tämä ei muuta järjestelmän tilaa, ellei sitä ole erikseen tarkoitettu muutettavaksi. REST-arkkitehtuurin etuja ovat sen yksinkertaisuus, joustavuus ja skaalautuvuus. Sovellukset voivat kehittyä itsenäisesti ja tehokkaasti ilman, että asiakas ja palvelin tarvitsevat tietoa toisistaan, kunhan ne vain noudattavat sovittuja viestintäformaatteja. (What is CRUD? s.a.)

5.1 Node.JS ja käytettävät teknologiat

Node.js on avoimen lähdekoodin alustariippumaton ympäristö, jonka Ryan Dahl julkaisi vuonna 2009 (Semah 2022). Se on asynkroninen, tapahtumapohjainen JavaScript-ajoympäristö, joka on suunniteltu skaalautuvien verkkosovellusten rakentamiseen. Sen avulla voidaan käsitellä useita eri yhteyksiä samanaikaisesti. Takaisinkutsufunktio (callback) käynnistyy jokaisen yhteyden kohdalla, mutta jos mitään työtä ei ole tehtävänä, Node.js siirtyy lepotilaan. (Node.js s.a.) Node.js mahdollistaa myös palvelinpuolen sovellusten rakentamisen samalla JavaScript-ohjelmointikielellä, jota käytetään myös asiakaspuolella (Semah 2022).

Node.js eroaa perinteisestä samanaikaisuusmallista, jossa käytetään käyttöjärjestelmän säikeitä (threads). Säiepohjainen verkkokäsittely voi olla kuitenkin suhteellisen tehotonta ja monimutkaista. Node.js:n käyttäjien ei myöskään tarvitse huolehtia prosessien lukituksista, sillä Node.js ei käytä lukkoja, eli prosessit eivät jää odottamaan vastauksia toisilta prosesseilta. Lähes mikään Node.js-funktio ei suorita suoraan I/O-operaatioita (input/output), joten prosessit eivät esty, paitsi silloin kun I/O-operaatioita suoritetaan synkronisesti. Koska prosessit eivät esty, skaalautuvien sovellusten kehitys on suhteellisen helppoa Node.js:ssä. (Node.js s.a.)

Node.js muistuttaa suunnittelultaan Rubyn Event Machinea sekä Pythonin Twistediä, joista se on ottanut vaikutteita. Se vie tapahtumamallin pidemmälle esittämällä tapahtumasilmukan (event loop) osana ajoympäristöä kirjaston sijaan. Muista järjestelmistä poiketen Node.js:ssä ei ole erillistä estävää kutsua tapahtumasilmukan käynnistämiseksi. Se siirtyy tapahtumasilmukkaan suoritetun koodin jälkeen ja poistuu siitä vasta, kun suoritettavia takaisinkutsuja ei enää ole. (Node.js s.a.)

HTTP on keskeinen osa Node.js:n toimintaa, ja se on suunniteltu erityisesti suoratoistoa ja alhaista latenssia silmällä pitäen. Tämä tekee Node.js:stä hyvin sopivan perustan verkkokirjastojen -tai kehysten rakentamiseen. (Node.js s.a.)

Yksi Node.js:n ominaisuus on NPM-pakettihallinta, joka tarjoaa pääsyn yli miljoonaan eri pakettiin. Paketit ovat uudelleenkäytettäviä koodikokonaisuuksia, joita kehittäjät voivat jakaa ja käyttää omissa projekteissaan. NPM:n avulla voidaan ottaa nopeasti käyttöön useita valmiita ratkaisuja eri ohjelmointitarpeisiin. (Semah 2022.)

5.1.1 Express.js

Express.js (Express) on suosituin web-kehys Node.js:lle, ja sitä pidetään käytännössä standardina palvelinsovellusten kehityksessä. Se on suunniteltu helpottamaan web-sovellusten ja APIen rakentamista tarjoamalla yksinkertaisen

ja helppokäyttöisen rajapinnan reitityksiin, middlewareen ja HTTP-apuohjelmiin. Expressin avulla voidaan säästää aikaa ja keskittyä liiketoimintalogiikan toteuttamiseen ilman turhaa pohjakoodin kirjoittamista. (What is Express.js? s.a.; Express.js Tutorial 2024)

Express on minimalistinen ja joustava kehys, joka ei pakota tiettyä sovellusarkkitehtuuria, vaan antaa kehittäjille mahdollisuuden valita omat lähestymistapansa. Se tukee sekä yksisivuisten sovellusten että RESTful APIen kehitystä. Lisäksi sitä voidaan käyttää myös reaaliaikaisten sovellusten, kuten chattien, rakentamiseen. (Express.js Tutorial 2024)

Express mahdollistaa middlewarejen käytön, joiden avulla voidaan suorittaa toimintoja pyyntöjen ja vastausten käsittelyn aikana. Middlewaret voivat olla sovellus- tai reittikohtaisia, ja ne voidaan ketjuttaa joustavasti. Melkein mikä tahansa yhteensopiva middleware voidaan lisätä pyynnön käsittelyketjuun lähes mihin tahansa järjestykseen. (What is Express.js? s.a.)

Express integroituu saumattomasti Node.js:n kanssa hyödyntäen sen tapahtumapohjaista ja asynkronista luonnetta. Se myös mahdollistaa muun muassa staattisten tiedostojen, kuten kuvien ja CSS-tiedostojen, vaivattoman palvelemisen. Lisäksi se tarjoaa tuen mallinnusmoottoreille, kuten EJS ja Jade, dynaamisen HTML-sisällön luomiseksi. (Express.js Tutorial 2024)

5.1.2 Sequelize

Sequelize on lupauspohjainen Node.js ORM (Object Relational Mapping) -työkalu useille eri tietokannoille, muun muassa PostgreSQL:lle, MySQL:lle ja Microsoft SQL Serverille (Sequelize 2025). ORM suorittaa toimintoja, kuten tietokantatietueiden käsittelyä, esittämällä tiedot olioina. Sequelize tarjoaa tehokkaan migraatiomekanismin, jonka avulla olemassa olevat tietokantaskeemat voidaan muuntaa uusiin versioihin. Lisäksi se tarjoaa vahvan tuen tietokannan synkronoinnille, ennakkoon lataamiselle, assosiaatioille, transaktioille ja migraatioille, mikä vähentää sovelluksen kehitysaikaa ja auttaa ehkäisemään SQL (Structured Query Language) -injektioita. (Dedigama & Lee 2022.)

Sequelizen tarjoamat metodit ovat pääosin asynkronisia, mikä tarkoittaa sitä, että sovelluksessa voidaan suorittaa prosesseja samanaikaisesti, kun asynkroninen koodilohko on käynnissä. Kun asynkroninen koodilohko on suoritettu onnistuneesti, se palauttaa lupauksen (promise), jonka avulla käsitelty tieto saadaan takaisin käyttäen then()-, finally()- ja catch() -menetelmiä. (Dedigama & Lee 2022.)

Sequelize tukee useita ominaisuuksia, kuten vankkaa transaktiotukea ja aiemmin mainittuja tietokantarelaatiota ja ennakkoon latausta. Se standardoi ORM-käytännöt tarjoamalla yhden skeemamäärityksen koodissa, mikä tekee sen ymmärtämisestä ja muokkaamisesta yksinkertaista ja selkeää. Lisäksi sen avulla ei ole välttämätöntä oppia SQL:ää, sillä kyselyt voidaan kirjoittaa suoraan JavaScriptillä. (How to use Sequelize in Node.js 2020.)

5.1.3 Nodemon

Nodemon on työkalu, joka auttaa Node.js-pohjaisten sovellusten kehittämisessä. Se käynnistää Node.js-sovelluksen automaattisesti uudelleen, kun hakemistossa havaitaan muutoksia tiedostoihin. (NPM 2024.)

Lisäksi Nodemonin toimintoina on listattu sen virallisilla sivuilla (Nodemon s.a.):

- Tunnistaa oletustiedostopäätteet seurattaviksi.
- Oletustuki Node.js:lle, mutta helppo ajaa mitä tahansa suoritettavaa tiedostoa, kuten Pythonia, Rubya, Makea jne.
- Mahdollisuus ohittaa tietyt tiedostot tai hakemistot.
- Seuraa tiettyjä hakemistoja.
- Toimii sekä palvelinsovellusten että kertaluonteisten ohjelmien ja REPL-ympäristöjen kanssa.
- Skriptattavissa Node.js:n require-lauseella.
- Avoimen lähdekoodin projekti, saatavilla GitHubissa.

5.1.4 Bcrypt

Bcrypt on Blowfish-salausalgoritmiin perustuva salasanojen hajautusmenetelmä, joka on suunniteltu tekemään salasanojen murtamisesta laskennallisesti hyvin vaativaa. Se sisältää suolan (salt), joka suojaa rainbow table -hyökkäyksiltä ja tekee jokaisesta hajautuksesta uniikin. Lisäksi Bcrypt tuottaa kiinteän mittaisen hajautusarvon (hash), joka varmistaa tietojen eheyden. (Twingate Team 2024.) Bcrypt on yksisuuntainen hajautusfunktio, mikä tarkoittaa, että hajautettua salasanaa ei voida enää palauttaa alkuperäiseen muotoonsa (Grigutytė 2023).

Bcryptin tärkeimpiin ominaisuuksiin kuuluvat suola, kustannustekijä (cost factor), hidas suoritusaika sekä mukautuvuus teknologian kehitykseen (Twingate Team 2024). Suola on 16-tavun satunnaisarvo, joka lisätään salasanaan ennen sen hajauttamista, jolloin jokaisesta hajautuksesta tulee yksilöllinen (Grigutytė 2023). Kustannustekijä määrittää hajautuskierrosten määrän, ja sen lisääminen kasvattaa laskennallista työkuormaa täten hidastaen mahdollisia brute force -hyökkäyksiä (Twingate Team 2024).

Bcryptin käyttäminen sovelluksissa tapahtuu kolmessa eri vaiheessa. Ensiksi luodaan yksilöllinen suola, jonka jälkeen salasana hajautetaan Bcrypt-algoritmilla. Lopuksi tuloksena saatu hajautusarvo ja suola tallennetaan tietokantaan. (Twingate Team 2024.) Salasanan tarkistaminen tapahtuu hajauttamalla annettu salasana uudelleen samalla suolalla ja vertaamalla tulosta tietokantaan tallennettuun hajautukseen. Kustannustekijä tulisi valita siten, että se on tasapainossa suorituskyvyn ja turvallisuuden kanssa. (Grigutytė 2023.)

Eroa muihin hajautusalgoritmeihin, kuten SHA256:een, löytyy etenkin Bcryptin mukautuvuuden ja suolan käytön ansiosta. Bcryptin kustannustekijä mahdollistaa laskennallisen vaativuuden kasvattamisen ajan myötä, kun taas SHA256 on suunniteltu olemaan nopea ilman mukautuvuutta. (Twingate Team 2024.) SHA256 ei myöskään sisällä oletusarvoisesti suolaa, mikä tekee siitä alttiimman sanakirjahyökkäyksille. Lisäksi SHA256 tuottaa aina saman hash-arvon samalle syötteelle, kun taas Bcrypt lisää kustannustekijän ja suolan tähän hajautusprosessiin. (Grigutytė 2023.)

5.2 JavaScript

JavaScript on tekstipohjainen ohjelmointikieli, jota käytetään sekä verkkoselaimessa asiakaspuolella (client-side) että palvelimella (server-side). Sen avulla voidaan lisätä verkkosivuille vuorovaikutteisia ja dynaamisia toimintoja, kuten animaatioita ja käyttäjän toimintaan reagoivaa sisältöä. Käytännön esimerkkejä JavaScriptin käytöstä voisi olla automaattinen uutisvirran päivitys tai verkkosivun hakukenttä. (Hackreactor 2022; What is JavaScript (JS)? s.a.)

Asiakaspuolen JavaScript

Asiakaspuolen JavaScript viittaa tapaan, jolla JavaScript toimii selaimessa. JavaScript-koodi suoritetaan JavaScript-moottorilla, joka on sisäänrakennettu kaikissa suurissa selaimissa kuten Chromessa, Firefoxissa ja Safarissa. Selaimen ladatessa verkkosivun JavaScript-moottori muuttaa sisällön DOM-rakenteeksi (Document Object Model), ja tämän jälkeen JavaScript voi muokata sitä reaaliaikaisesti käyttäjän toimien mukaan. (What is JavaScript (JS)? s.a.)

Palvelinpuolen JavaScript

Palvelinpuolen JavaScript viittaa ohjelmointikielen käyttöön taustapalvelimen logiikassa. Tässä tapauksessa JavaScript-moottori sijaitsee suoraan palvelimella. Palvelinpuolen JavaScript-funktio voi käyttää tietokantaa, suorittaa erilaisia loogisia operaatioita ja reagoida palvelimen käyttöjärjestelmän käynnistämiin tapahtumiin. Palvelinpuolen skriptauksen suurin etu on, että sen avulla voi mukauttaa verkkosivun vastauksia erittäin tarkasti tarpeiden, käyttöoikeuksien ja verkkosivulta tehtyjen tietopyyntöjen perusteella. (What is JavaScript (JS)? s.a.)

JavaScriptin hyödyt

JavaScript on melko helppo oppia käyttämään, ja se toimii käytännössä kaikissa selaimissa ja alustoissa. Se mahdollistaa nopean reagoinnin ilman pal-

velimen ylimääräistä kuormittamista, mikä parantaa merkittävästi käyttökokemusta. Lisäksi JavaScript tukee rinnakkaista suorittamista, erityisesti Node.js:n avulla. (Hackreactor 2022; What is JavaScript (JS)? s.a.)

JavaScriptiä varten on luotu paljon kirjastoja ja kehyksiä, jotka voivat nopeuttaa web-kehitystä tarjoamalla valmiita ratkaisuja yleisiin ohjelmointitehtäviin. Kirjastot koostuvat valmiista koodinpätkistä, joita kehittäjät voivat hyödyntää esimerkiksi lomakkeiden validointiin, animaatioihin tai datan visualisointiin. Esimerkkejä tunnetuista kirjastoista ovat jQuery, Chart.js ja Date.js. Kehykset puolestaan tarjoavat laajemman rakenteen koko sovelluksen rakentamiseen. Ne määrittävät, miten eri osat, kuten logiikka, näkymät ja tiedonhallinta järjestetään sovelluksessa. Suosittuja JavaScript-kehyksiä ovat muun muassa Angular, React, Vue ja Node.js. Kehysten avulla koodin organisointi helpottuu ja kehitysprosessi tehostuu. (Hackreactor 2022; What is JavaScript (JS)? s.a.)

JavaScriptin rajoitteet

JavaScript on heikosti tyypitetty kieli, eli muuttujien tyyppejä ei määritellä etukäteen. Muuttujan tyyppi määrää, minkälaista tietoa, kuten numeroita, merkkijonoja tai taulukoita, muuttuja voi sisältää. Koska tyyppiä ei rajoiteta, tämä voi johtaa virheisiin – esimerkiksi tilanteeseen, jossa numero ja teksti yhdistyvät odottamattomasti. Tällaiset virheet voivat aiheuttaa bugeja, joita on vaikea havaita ennen ohjelman suorittamista. (Hackreactor 2022; What is JavaScript (JS)? s.a.)

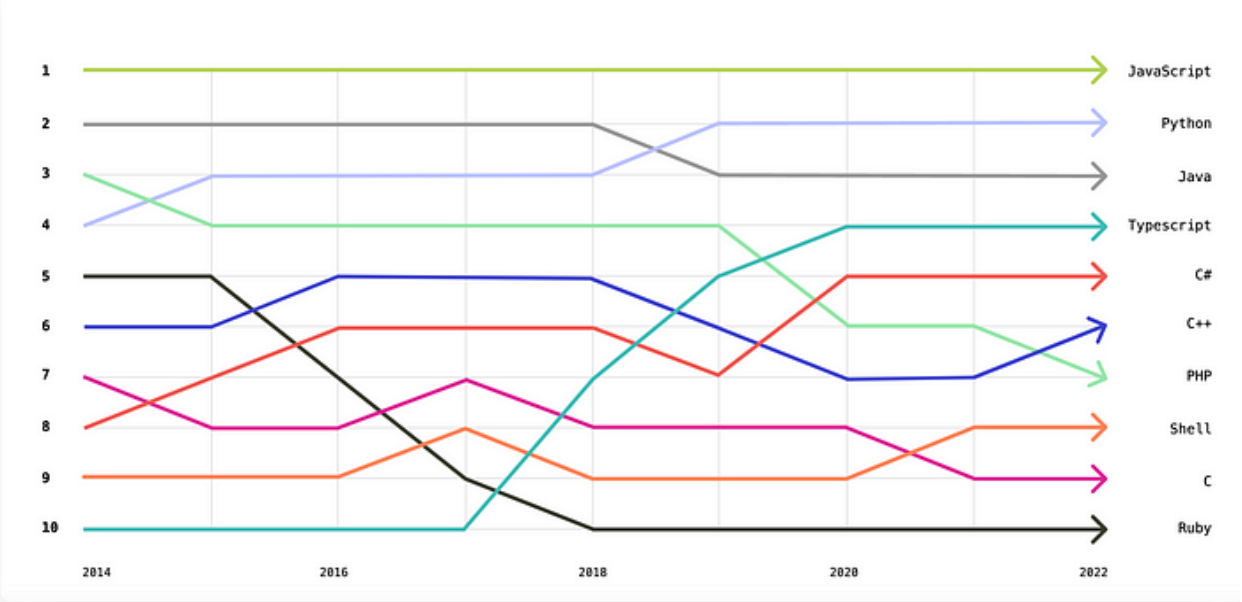
5.3 TypeScript

TypeScript on staattisesti tyypitetty ohjelmointikieli ja JavaScriptin ylijoukko (superset), mikä tarkoittaa, että kaikki validi JavaScript-koodi on myös validia TypeScript-koodia. TypeScript laajentaa JavaScriptin syntaksia ja toiminnallisuutta tarjoamalla muun muassa mahdollisuuden määritellä muuttujien ja funktioiden tyyppejä. Tämä auttaa ehkäisemään virheitä jo kehitysvaiheessa, sillä tyypitysten ansioista virheitä voidaan havaita ennen koodin suorittamista. Sen sijaan puhdas JavaScript-koodi ei välttämättä toimi TypeScript-projektissa

ilman tarvittavia muutoksia, koska JavaScriptistä puuttuu TypeScriptin tarjoamia ominaisuuksia (Agrawal & Fateh 2022; Motunrayo 2024.)

TypeScript kehitysympäristöissä

TypeScriptiä käytetään laajasti moderneissa kehitysympäristöissä. Sitä voidaan hyödyntää niin frontend- kuin backend-kehityksessä, esimerkiksi React-, Angular-, Node.js-, Fastify-, ja NestJS-projekteissa. Sen suosio on kasvanut merkittävästi, ja monet suuret kehittäjäyhteisöt ja yritykset tukevat sitä aktiivisesti. TypeScriptin käytön kasvu näkyy konkreettisesti myös GitHubin tilastoissa: vuonna 2022 se oli neljänneksi käytetyin ohjelmointikieli alustalla (kuva 2). (Motunrayo 2024.)



Kuva 2. Kymmenen käytetyintä ohjelmointikieltä GitHubin 2022 raportin mukaan (Garcia 2023)

TypeScriptin edut

TypeScriptin suurin etu on sen tarjoama ennakoiva virheidenhallinta staattisen tyypityksen avulla. Tämä parantaa koodin luotettavuutta ja helpottaa suurien sovellusten ylläpitoa. TypeScript myös parantaa kehittäjäkokemusta, koska tyypit mahdollistavat älykkäämmät koodieditorin ominaisuudet, kuten auto-

maattisen täydennyksen, virheilmoitukset reaaliajassa ja helpomman refaktoroinnin. Tämä tekee koodin kirjoittamisesta tehokkaampaa ja vähemmän virhealtista. (Agrawal & Fateh 2022.)

TypeScriptin kääntäminen ja yhteensopivuus

TypeScript koodi täytyy kääntää (transpiloida) tavalliseksi JavaScriptiksi, jotta se toimii selaimessa tai JavaScript-ympäristössä. Tämä tehdään yleensä automaattisesti build-työkalujen avulla. Kääntämisen ansiosta TypeScriptin tyypitykset eivät vaikuta suoraan sovelluksen suoritukseen, mutta ne mahdollistavat kehitysvaiheessa paremman virheentarkastelun ja selkeämmän koodin. TypeScriptin yhteensopivuus JavaScriptin kanssa helpottaa sen käyttöönottoa olemassa olevissa projekteissa vaiheittain. (Agrawal & Fateh 2022; Motunrayo 2024.)

5.4 JavaScript — TypeScript -migraatio

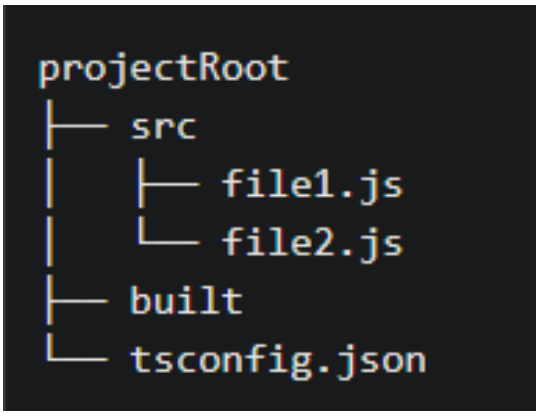
Migraation tavoitteena on parantaa koodin laatua ja ennakoitavuutta hyödyntämällä TypeScriptin tyyppijärjestelmää. Käytännössä tämä tarkoittaa virheiden tarkempaa havaitsemista kehitysvaiheessa sekä parempaa editoritukea. Tässä kappaleessa käsitellään tyypillisimmät migraatioon liittyvät ongelmatilanteet ja ratkaisut, joita on listattu TypeScriptin virallisille sivuille.

JavaScriptistä TypeScriptiin siirtyminen ei tapahdu tyhjiössä. TypeScript on suunniteltu ajatellen JavaScript-ekosysteemiä, ja suurin osa olemassa olevasta koodista on yhä JavaScriptiä. Siirtyminen JavaScriptistä voi silti olla työlästä, mutta sen ei pitäisi olla erityisen vaikeaa. (Migrating from JavaScript 2025.)

Projektin hakemistorakenne

Ensimmäinen askel migraatiossa on asetella hakemistot järkevästi. Jos on aikaisemmin suorittanut JavaScript-tiedostoja esimerkiksi *src*-hakemistosta, kannattaa erottaa alkuperäiset JavaScript-tiedostot TypeScriptin tuottamista

tiedostoista. (Migrating from JavaScript 2025.) Alla olevassa kuvassa 3 on esimerkki tyypillisestä hakemistorakenteesta.



Kuva 3. TypeScript-hakemistorakenne (Migrating from JavaScript 2025)

TypeScriptin konfiguraatio

Seuraava vaihe on TypeScriptin konfiguraatio. TypeScript vaatii asetustiedoston (tsconfig.json), jossa määritellään, miten projektia käsitellään. (Migrating from JavaScript 2025.) Alla olevassa kuvassa (kuva 4) on esimerkki hyvin yksinkertaisesta konfiguraatiotiedostosta.

```
{
  "compilerOptions": {
    "outDir": "./built",
    "allowJs": true,
    "target": "es5"
  },
  "include": ["./src/**/*"]
}
```

Kuva 4. TypeScript-konfiguraatiotiedosto (tsconfig.json) (Migrating from JavaScript 2025)

Kuvassa 4 näkyvä tiedosto määrittelee:

- Built-hakemisto Typescriptin ulostulolle
- JavaScript tiedostot sallitaan.
- Moderni JavaScript muunnetaan ES5-tasolle, jotta se toimii laajasti.
- TypeScript kääntää kaikki src-hakemiston tiedostot.

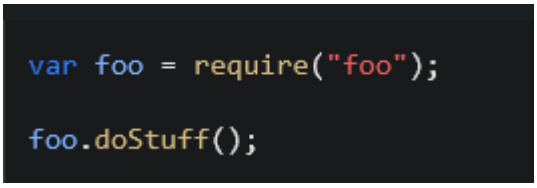
Siirtyminen TypeScript-tiedostoihin

Seuraavaksi voidaan alkaa nimetä js-päätteisiä tiedostoja ts-päätteisiksi. Jos tiedostossa käytetään JSX-syntaksia, tulee tiedostolle laittaa tsx-pääte. JSX on JavaScript laajennus, joka antaa mahdollisuuden kirjoittaa HTML:n kaltaista koodia JavaScript-kielen joukkoon. JSX on käytössä esimerkiksi Reactissa. (Writing Markup with JSX s.a.; Migrating from JavaScript 2025.)

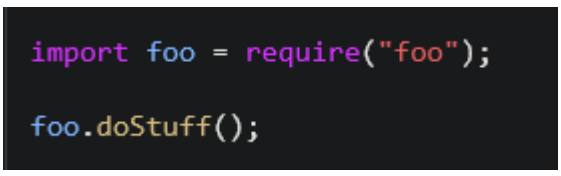
Kun avataan TypeScriptiin siirretty tiedosto editorissa, saattaa näkyä useita punaisia virheilmoituksia. Tämä on TypeScriptin tapa kertoa mahdollisista ongelmista, mutta koodi kääntyy silti, ellei konfiguraatiotiedostoon ole määritelty tiukempaa tarkistusta (noEmitOnError).

Moduulien tuonti ja vienti

JavaScriptissä käytetään require- tai define -komentoja. Nämä kannattaa korvata TypeScriptin import-syntaksilla. Seuraavat kaksi kuvaa (kuvat 5 ja 6) antavat esimerkin JavaScriptin ja TypeScriptin välisestä muutoksesta. (Migrating from JavaScript 2025.)



Kuva 5. JavaScript require -komento (Migrating from JavaScript 2025)



Kuva 6. TypeScript import -komento (Migrating from JavaScript 2025)

Kun TypeScriptin import-syntaksin käyttö on aloitettu, on mahdollista saada virheilmoituksia, kuten ”Cannot find module ’foo’.”. Tämä johtuu siitä, että TypeScript ei löydä moduulin tyypityksiä, ja ratkaisu on asentaa tarvittava tyypityspaketti NPM:n kautta. (Migrating from JavaScript 2025.)

Argumentit

Joskus funktiota kutsutaan liian paljolla tai vähällä määrällä argumentteja. Tässä tapauksessa on mahdollista, että funktio on tehty käyttämään joustavia argumenttimääriä käyttämällä argument-objektia. TypeScriptissä käytetään funktioylikuormituksia (function overloads), joilla voidaan määritellä tarkkaan, miten funktiota voidaan kutsua. (Migrating from JavaScript 2025.)

Oliot

JavaScriptissä on joskus tapana luoda tyhjä olio ja lisätä sen ominaisuudet jälkeenpäin. Alla olevassa kuvassa 7 on esimerkki toimintatavasta.

```
var options = {};
options.color = "red";
options.volume = 11;
```

Kuva 7. Olioon ominaisuuksien lisääminen luonnin jälkeen (Migrating from JavaScript 2025)

TypeScript ajattelee, että (kuvassa 7) options on tyypiltään tyhjä olio ({ }), jolla ei ole ominaisuuksia ja tämän takia tuloksena on virhe. Korjauksena tähän toimii ominaisuuksien listaaminen heti luontivaiheessa (kuva 8) tai vaihtoehtoisesti voi kertoa TypeScriptille options-olion tyypin ja käyttää tyyppimuunnosta (kuva 9). (Migrating from JavaScript 2025.)

```
let options = {
  color: "red",
  volume: 11,
};
```

Kuva 8. Olion ominaisuuksien lisääminen luontivaiheessa (Migrating from JavaScript 2025)

Kuvassa 8 osoitetaan, kuinka options-olio voidaan luoda valmiiksi halutuilla kentillä, tässä tapauksessa color ja volume. Näin TypeScript tietää heti, millaisia arvoja odottaa.

```
interface Options {
  color: string;
  volume: number;
}

let options = {} as Options;
options.color = "red";
options.volume = 11;
```

Kuva 9. Tyyppimuunnoksen käyttö (Migrating from JavaScript 2025)

Edellisessä esimerkissä (kuva 9) esitellään Options-rajapinta sekä tyyppimuunnos (as Options), jonka avulla tyhjä objekti muuttuu rajapinnan mukaiseksi ja kenttiä voi lisätä vaiheittain ilman tyypitysongelmia.

6 KEHITYSYMPÄRISTÖ

6.1 Unreal Engine 5

Unreal Engine on Epic Gamesin kehittämä pelimoottori, joka tarjoaa kehittäjille laajat työkalut visuaalisesti vaikuttavien ja immersiivisten 3D-pelien ja virtuaalikokemusten luomiseen. Unreal Engine on tunnettu monipuolisuudestaan ja tehokkuudestaan, ja sitä käytetään laajasti esimerkiksi pelikehityksessä, arkkitehtuurin visualisoinnissa, virtuaalitodellisuuskokemuksissa ja elokuvatuotannossa. (What Is Unreal Engine? 2024; Erolin s.a.)

Pelimoottorin merkittäviä ominaisuuksia ovat muun muassa dynaaminen valaistus, joka mahdollistaa etäisyyskentän pehmeiden varjojen ja liikkuvien valojen käytön, sekä edistyneet fysiikkasimulaatiot, kuten neste- ja hiusfysiikan ja tuhoutumissimuloinnit. Moottori perustuu modulaariseen arkkitehtuuriin, jonka ansiosta kehittäjien on helppo pitää koodinsa selkeänä ja helposti päivitettävänä ilman pelin rikkoutumista. Unreal Enginen Blueprints-työkalu mahdollistaa visuaalisen ohjelmoinnin, mikä tekee moottorista helposti saavutettavan myös vähemmän ohjelmointikokemusta omaaville kehittäjille. (What Is Unreal Engine? 2024.)

Epic Games julkisti Unreal Engine 5:n vuonna 2020, ja se toi mukanaan merkittäviä parannuksia edelliseen versioon, Unreal Engine 4:ään. Uuden version keskeisiä ominaisuuksia ovat Lumen, joka mahdollistaa reaaliaikaisen dynaamisen valaistuksen; Nanite, visuaalisen tarkkuuden parantamiseen tehty mikropolygoni-geometriajärjestelmä; Temportal Super Resolution -ominaisuus, joka mahdollistaa korkean resoluution matalilla laskentatehoilla; sekä kehittynyt animaatiotyökalupaketti pelihahmojen luomiseen ja muokkaamiseen. (What Is Unreal Engine? 2024; Unreal Engine 2022.)

Unreal Engine on ilmainen ja käyttäjäystävällinen uusille käyttäjille. Lisäksi moottori tukee useita alustoja, kuten iOS, Android, Windows, PlayStation ja Xbox, mikä tekee siitä monipuolisen valinnan eri tarpeisiin. (Erolin s.a.) On kuitenkin huomioitava, että Epic Games perii 5 prosentin rojaltimaksun, jos kehitetty tuote ansaitsee yli miljoona dollaria (What Is Unreal Engine? 2024.)

6.2 Visual Studio Code

Visual Studio Code (VS Code) on ilmainen, kevyt, ja erittäin tehokas lähdekoodieditori, joka tarjoaa käyttäjilleen yksinkertaiset muokkaustoiminnot yhdessä kehittyneiden kehittäjätyökalujen kanssa. VS Code sisältää muun muassa älykkään IntelliSense-koodin täydennyksen, interaktiivisen debuggaustyökalun ja sisäänrakennetun Git-tuen. Nämä kaikki edistävät koodauksen sujuvuutta ja parantavat kehittäjien tuottavuutta. (Heller 2022; Visual Studio 2024a.)

Editori on suunniteltu toimimaan saumattomasti eri käyttöjärjestelmillä, kuten Windowsilla, macOS:llä ja Linuxilla, mikä mahdollistaa sen joustavan käytön monipuolisissa kehitysympäristöissä (Heller 2022; Visual Studio 2024a). Sen modulaarinen arkkitehtuuri hyödyntää moderneja web-teknologioita, kuten JavaScriptiä, Node.js:ää ja TypeScriptiä ja tarjoaa kevyen rakenteen ja nopean suorituskyvyn (Visual Studio 2024b). Näin VS Code pystyy yhdistämään natiivien sovellusten tehokkuuden ja web-teknologioiden joustavuuden, mikä mahdollistaa sekä pienten että suurten projektien sujuvan hallinnan (Heller 2022).

Lisäksi Heller (2022) korostaa, että VS Code yhdistää lähdekoodieditorin yksinkertaisuuden ja kehittyneet kehittäjätyökalut siten, että kehittäjät voivat käyttää aikaansa tehokkaasti ideoidensa toteuttamiseen. Editorin laaja laajennusekosysteemi mahdollistaa ominaisuuksien räätälöinnin omiin tarpeisiin, mikä tekee siitä erittäin mukautettavan työkalun. Kehittäjät voivat itse lisätä uusia toimintoja ja integroida kolmannen osapuolen työkaluja, mikä edistää jatkuvaa oppimista ja tehokasta työskentelyä. VS Code on suunniteltu poistamaan koodauksen, rakentamisen ja debuggaamisen väliset esteet, jolloin kehittäjät voivat keskittyä tekemiseen ilman, että aikaa kuluu koodieditorin ja debuggaamisen välisiin ongelmiin. (Heller 2022.)

7 FRONTEND-TEKNOLOGIAT JA KEHITYS

Frontend viittaa kaikkeen siihen, mitä käyttäjä näkee sovellusta käyttäessä omassa käyttöliittymässään: napit, kuvat, navigaatioelementit, kuvaajat, tekstit ja niin edelleen. (What’s the Difference Between Frontend and Backend in Application Development? s.a.) Frontend-kehityksessä keskitytään käyttäjäkokemukseen. Tavoitteena on sovellus, joka on helppo käyttää ja joka on turvallinen, miellyttävän näköinen ja nopea. (Cloudinary 2024.)

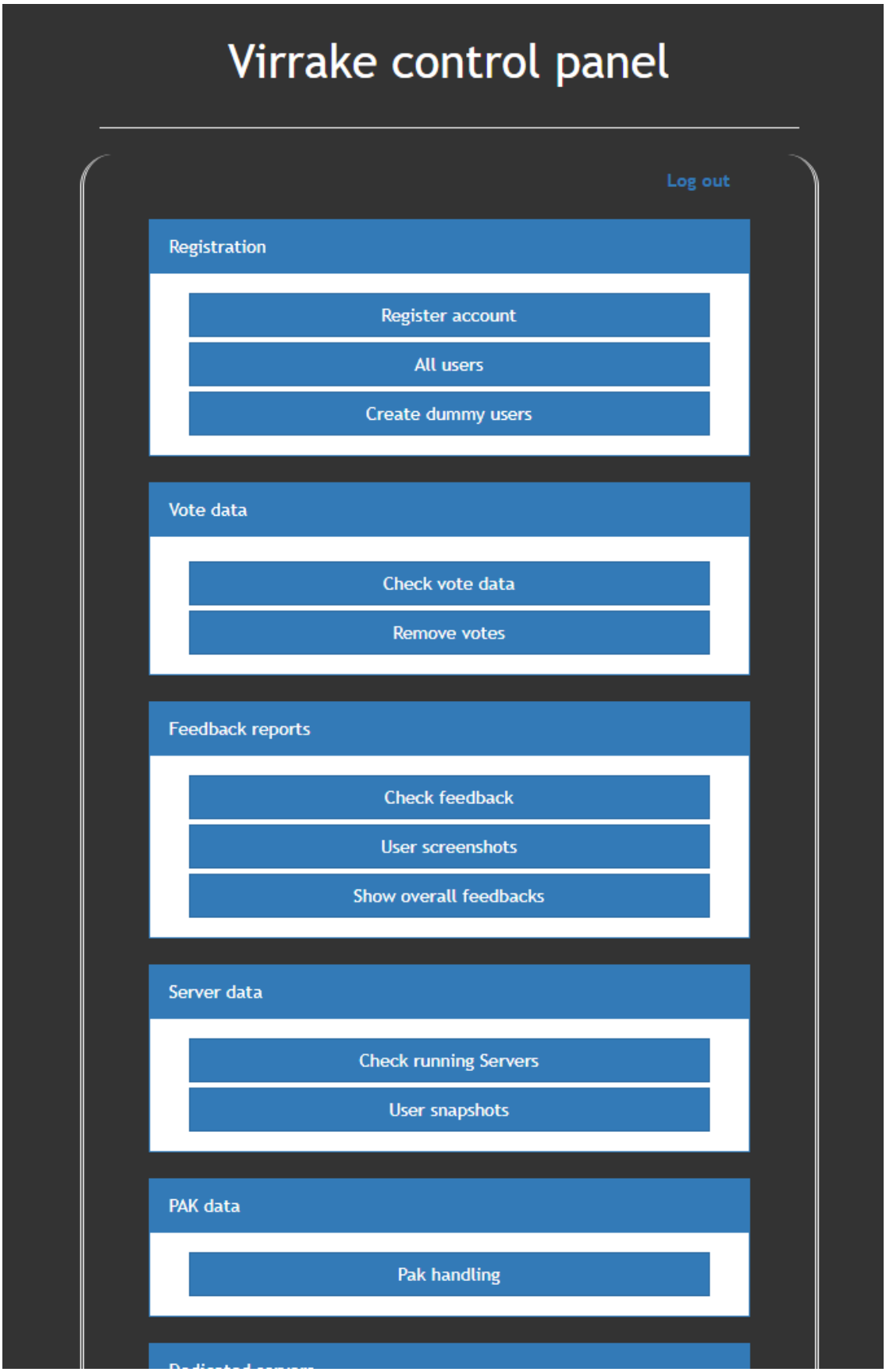
Frontend-ohjelmistokehykset ja -kirjastot

Frontend-ohjelmistokehykset ovat ohjelmistokehityksessä auttavia työkaluja, jotka sisältävät tiettyjä perusfunktioita tai muita koodikomponentteja valmiiksi tehtyinä ja paketoituina kirjastoihin. Tämä tarkoittaa sitä, että kaikkea sovelluksen toiminnallisuutta ei välttämättä tarvitse tehdä tyhjästä. Suosittuihin kehyksiin kuuluvat Angular, React, Svelte, Bootstrap, Vue.js ja jQuery. (Cloudinary 2024.)

Virrake-ohjauspaneeli

Virrakkeen backend sisältää oman selainpohjaisen ohjauspaneelinsa (Virrake control panel), josta voi hallita käyttäjiä ja hallita tai seurata muuta Virrakkeen toimintadataa. Voidaan esimerkiksi luoda uusia käyttäjiä tai luoda palvelimia

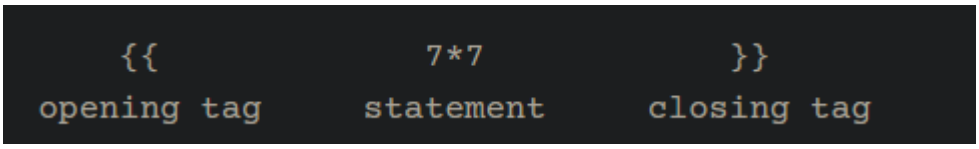
sessioita varten. Alla olevasta kuvasta (kuva 10) käy ilmi Virrakkeen ohjauspaneelin ulkonäkö selaimessa ja sen sisältämät toiminnot. Ohjauspaneeli on tehty käyttäen Handlebars-mallimoottoria (template engine). (Ojala 2025.)



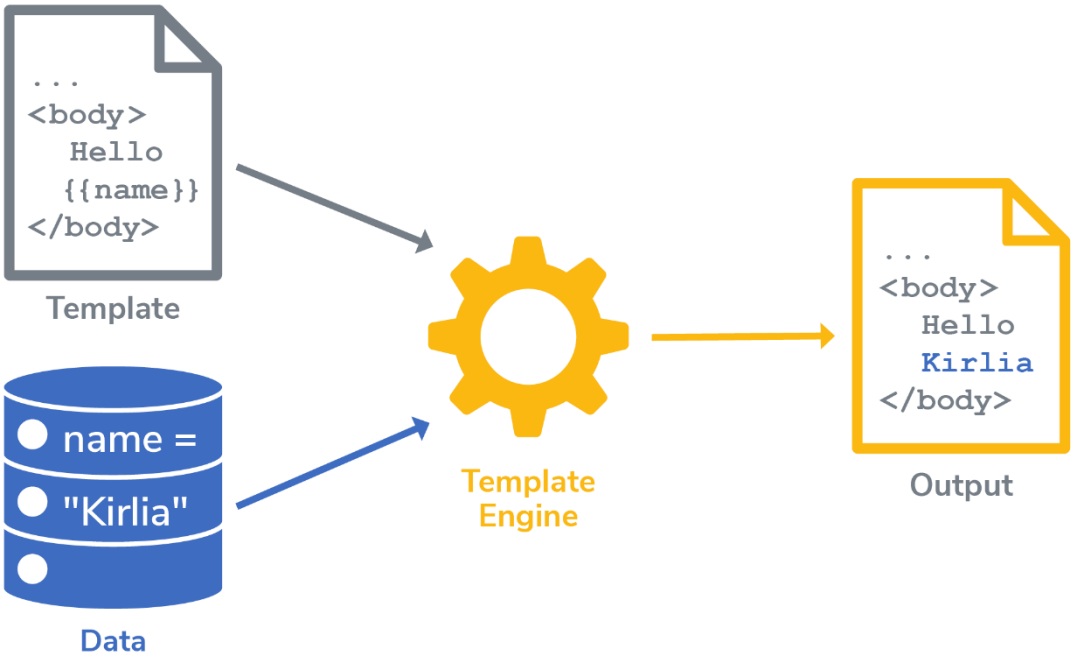
Kuva 10. Virrakkeen ohjauspaneelin näkymä selaimessa (Ojala 2025)

Mallimoottorit

Mallimoottori (kuva 12) on ohjelmistotyökalu, joka yhdistää olemassa olevaa dataa esimerkiksi tietokannasta valmiiksi luotuihin malleihin ja siten luo esimerkiksi HTML-sivuja dynaamisesti sovelluksen suorittamisen aikana. Yksittäisiä mallimoottorille tehtyjä ohjeita kutsutaan mallilausekkeiksi. Lausekkeet koostuvat yleisesti ottaen avaus- ja lopetusmerkistä ja niiden sisälle kirjoitetusta komennosta (kuva 11). (Hildebrand & Selhausen 2024; The Codest s.a.)



Kuva 11. Kuvakaappausesimerkki mallilausekkeesta (Hildebrand & Selhausen 2024)

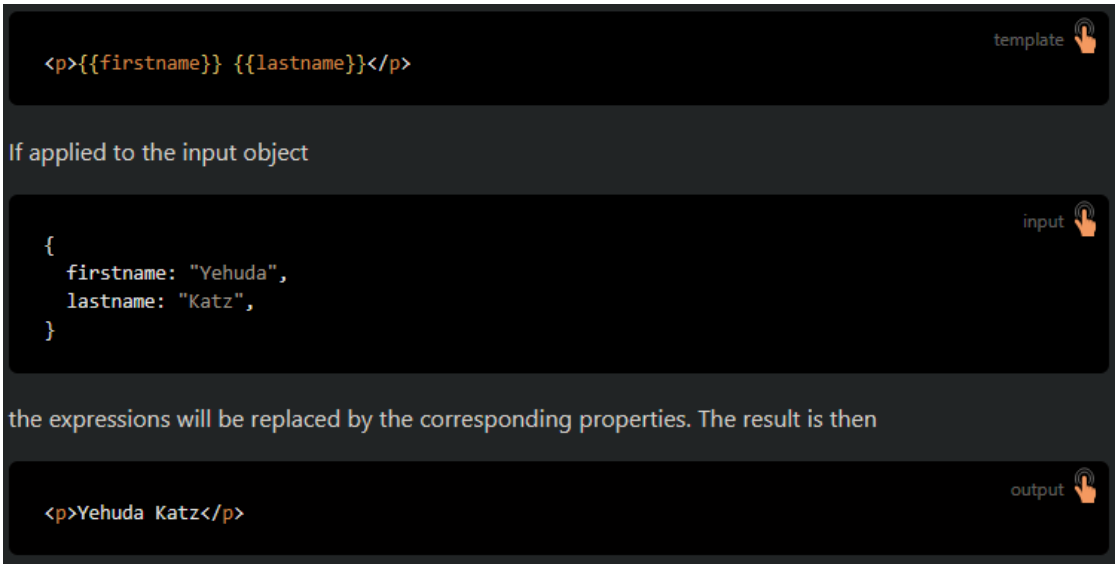


Kuva 12. Mallimoottorin toiminta (Hildebrand & Selhausen 2024)

Handlebars

Kuten edellä mainittu, Virrakkeen ohjauspaneeli on tehty käyttäen handlebarsjs-mallimoottoria. Handlebars on yksinkertainen mallinnuskieli. Se käyttää mallia ja syöteobjektia generoidakseen HTML:ää tai jotain muuta tekstimuotoa

(kuva 13). Handlebars-lausekkeen syntaksi mukailee yleistä mallia (kuva 11). (Handlebars.js 2021.)



Kuva 13. Kuvakaappaus Handlebars.js:n toiminnasta (Handlebars.js 2021)

Edellä olevassa kuvassa 13 näkyy esimerkki Handlebars-mallipohjan toiminnasta. Yläosassa on yksinkertainen Handlebars-mallilauseke, joka viittaa muuttujien firstname ja lastname arvoihin. Keskellä on syöteobjekti (JSON-muotoinen data), jossa on annettu muuttujien arvot. Alhaalla on mallin tuottama lopputulos, jossa Handlebars on korvannut muuttujat vastaavilla arvoilla ja luonut lopullisen HTML-sisällön. Tämä havainnollistaa Handlebarsin perusideaa: datan ja mallin yhdistämistä HTML-tuotoksen luomiseksi.

8 TIETOKANTARATKAISUT JA -YMPÄRISTÖT

Virrakkeessa on valmius toimia useamman eri relaatiotietokannan kanssa. Relaatiotietokanta on tapa jäsentää tietoa taulukoihin, riveihin ja sarakkeisiin, jolloin eri tietokantarakenteiden välisiä suhteita voidaan hahmottaa ja hyödyntää tehokkaasti. E.F. Codd kehitti relaatiotietokannan mallin IBM:llä 1970-luvulla. Tämä korvasi hierarkkiset tietorakenteet aiempaa joustavammalla ja skaalautuvammalla ratkaisulla. Relaatiotietokannoissa taulukot yhdistetään toisiinsa yhteisten attribuuttien, kuten ensisijaisten ja vieraiden avainten avulla. Tämä mahdollistaa tehokkaan tiedonhallinnan ja joustavat kyselyt, joissa käyttäjä voi hakea tietoa määrittelemällä, mitä haluaa, ilman että täytyy tietää, miten kyseinen järjestelmä käsittelee haun. (What is relational database? s.a.)

8.1 Tietokantaratkaisujen vertailu

Relaatiotietokantoja käytetään laajasti erilaisissa sovelluksissa, ja niiden valintaan voivat vaikuttaa useat tekijät, kuten suorituskyky, kustannukset ja yhteensopivuus. Tässä luvussa vertailemme MySQL:ää, Microsoft SQL Serveriä, MariaDB:tä sekä PostgreSQL:ää eri näkökulmista.

MySQL

MySQL on yksi suosituimmista relaatiotietokannoista, joka on alun perin kehitetty avoimen lähdekoodin ratkaisuna, mutta nykyään se on Oraclen omistuksessa (Altexsoft 2023). Sen vahvuuksia ovat korkea suorituskyky, helppokäyttöisyys ja ACID-transaktioiden tuki (atomicity, consistency, isolation ja durability), mikä mahdollistaa luotettavan datan hallinnan aina pienistä web-sovelluksista suurten, liikennöityjen järjestelmien tarpeisiin (Oracle 2024). Lisäksi MySQL tarjoaa ilmaisen yhteisöversion, selkeän ja ihmismäisen syntaksin sekä laajan pilvipalvelutuen, mikä tekee siitä kustannustehokkaan vaihtoehdon erityisesti pienille yrityksille (Altexsoft 2023)

Toisaalta järjestelmän haasteina ovat skaalautuvuuden rajoitteet, osittain rajoitettu avoimen lähdekoodin tuki sekä SQL-standardien puutteellinen noudattaminen, mikä voi vaikuttaa tietokannan joustavuuteen ja laajennettavuuteen laajemmissa käyttöympäristöissä (Altexsoft 2023).

Microsoft SQL Server

Microsoft SQL Server (MSSQL) on kaupallinen relaatiotietokantajärjestelmä, joka hyödyntää Transact-SQL (T-SQL) -kieltä tietojen tallennukseen, muokkaamiseen ja hallintaan. Järjestelmästä löytyy useita eri versioita: ilmainen Express-versio, joka sopii pienempien sovellusten kehittämiseen, sekä Developer-, Web-, Standard- ja Enterprise-versiot, jotka vastaavat erilaisten liiketoimintaympäristöjen tarpeita. MSSQL tarjoaa kokonaisvaltaisen ratkaisun liike-

toimintatiedon hallintaan: sen avulla voidaan mm. hallita tietovarastoja, suorittaa online-analyyttistä käsittelyä, tehdä tiedonlouhintaa sekä tuottaa raportteja ja visualisointeja. (Altexsoft 2023)

Lisäksi MSSQL on suunniteltu toimimaan monipuolisesti eri ympäristöissä. Se voidaan asentaa Windows- ja Linux-käyttöjärjestelmiin, ajaa Linux-konteissa tai hyödyntää pilvipohjaisina ratkaisuina esimerkiksi Azure Virtual Machinesin avulla, mikä tuo palveluun joustavuutta ja skaalautuvuutta (Microsoft 2024). Haasteina palvelulle ovat korkeat kustannukset, erityisesti Enterprise-version osalta, monimutkaiset ja vaihtelevat lisenssiehdot, jotka vaikeuttavat hinnoittelua ja lisenssien hallintaa, sekä suorituskyvyn optimointi suuria tietomääriä käsiteltäessä (Altexsoft 2023)

MariaDB

MariaDB on avoimen lähdekoodin haara MySQL:stä alkuperäisen MySQL:n kehittäjiltä ja se on yksi suosituimmista tietokantapalvelimista maailmassa (Altexsoft 2023; MariaDB s.a.). Se on hyvä esimerkki tietokannasta, jolla on kaupallinen tuki. Sen vahvuuksia ovat tietojen salaus: sisäisten turva- ja salasanatarkistusten lisäksi MariaDB tarjoaa useita eri ominaisuuksia tietoturvan parantamiseksi. MariaDB:n vahvuuksiin kuuluvat myös laaja toiminnallisuus ja mahdollisuus laajentamiseen lisäosien avulla sekä vahva suorituskyky. (Altexsoft 2023)

Heikkouksina mainittakoon pieni, yhä kasvava yhteisö sekä MySQL- ja MariaDB-päivitysversioiden väliset aukot. MariaDB:llä on merkittävä avoimen lähdekoodin panostus, mutta sen yhteisö on yhä suhteellisen pieni ja asiantuntijoiden määrä rajallinen, sillä tietokannan hallintajärjestelmä on vielä suhteellisen nuori. Vaikka MariaDB:n tiimi pyrkii jatkuvasti yhdistämään koodiaan MySQL:n kanssa, päivitysten välillä on jo eroja, ja tulevaisuudessa erojen ennakoidaan vain kasvavan. MariaDB:n ja MySQL:n välillä ilmenevät yhteensopivuusongelmat voivat aiheuttaa haasteita erityisesti siirrettäessä dataa MariaDB:stä MySQL:ään. (Altexsoft 2023)

PostgreSQL

PostgreSQL on tehokas avoimen lähdekoodin objektirelaatiotietokantajärjestelmä, joka yhdistää perinteisen relaatiomallin laajennettavuuteen (PostgreSQL s.a.). Lisäksi se tukee sekä relaatiotyyppisiä että ei-relaatiotyyppisiä tietotyyppejä (IBM 2021). Sen etuja ovat muun muassa erinomainen skaalautuvuus ja suorituskyky, jotka mahdollistavat sekä yksittäismallisten järjestelmien että laajojen, monimutkaisten sovellusten tehokkaan tietojenkäsittelyn (IBM 2021; PostgreSQL s.a.). Lisäksi järjestelmä tarjoaa natiivin tuen useille eri tietotyypeille, kuten JSON ja XML, sekä mahdollisuuden määrittää omia tietotyyppejä, mikä lisää sen monipuolisuutta. PostgreSQL hyödyntää myös laajasti yhteisön tukemia kolmannen osapuolen työkaluja, jotka helpottavat hallintaa, monitorointia ja skaalautuvuutta. (Altexsoft 2023)

Haittapuolena PostgreSQL:ssä on havaittavissa epäjohdonmukaisuuksia dokumentaatiossa sekä puutteita raportointi- ja auditointityökaluissa, mikä saattaa viivästyttää ongelmien havaitsemista. (Altexsoft 2023) Lisäksi Rathbone (2023) tuo esiin, että PostgreSQL:llä on puutteita vaakasuorassa skaalautuvuudessa, tietyissä NoSQL-ominaisuuksissa sekä hallinnan helppoudessa. Täten se ei välttämättä sovellu parhaiten suuriin analytiikkakyselyihin (Rathbone 2023).

8.2 Laajennettavuus ja integrointi

Virrake toimii yleisenä alustana, joka on suunniteltu integroitumaan asiakkaiden olemassa oleviin tietokantaratkaisuihin. Koska eri tahoilla voi olla jo vakiintunut tietokantainfrastruktuuri, Virrake ei edellytä uuden tietokantaratkaisun rakentamista, vaan mahdollistaa sen, että käyttäjä voi halutessaan hyödyntää nykyistä, jo olemassa olevaa ratkaisuaan. Tämän takia Virrakkeen refaktoroinnissa laajennettavuus on tärkeässä roolissa. (Ojala 2025.)

Laajennettavuus on tärkeää huomioida Virrakkeen refaktoroinnin yhteydessä, jotta sovellus voidaan sovittaa eri tahojen tietokantaratkaisuihin ilman merkittäviä muutoksia. Tämä mahdollistaa joustavuuden sovelluksessa: Virrake voi

toimia eri tietokantajärjestelmien kanssa ilman erityisiä rajoituksia. Lisäksi yhteensopivuuden avulla asiakkaiden ei tarvitse muokata omaa tietokantaansa Virrakkeen käyttöä varten. Koska tarvetta investoida uuteen infrastruktuuriin tai henkilökunnan kouluttamiseen uuteen tietokantaympäristöön ei ole, laajennettavuudesta huolehtiminen on hyvin kustannustehokasta. Lisäksi käyttäjäkokemus on huomioitu: on helpompaa ja mukavampaa jatkaa olemassa olevien työkalujen ja hallintamenetelmien käyttämistä niin halutessaan. (Ojala 2025.)

Vaikka Virrake tarjoaakin joustavuutta ja yhteensopivuutta, liittyy siihen myös haasteita (Ojala 2025.):

- **Tietoturva:** Koska sovellus voi liittyä eri tietokantoihin, on varmistettava, ettei se muodosta tietoturva-aukkoja esimerkiksi heikon autentikoinnin tai valtuutuksen kautta.

- **Yhteensopivuusongelmat:** Eri tietokannat tukevat eri SQL-käskyjä ja ominaisuuksia, mikä voi aiheuttaa haasteita, jos ohjelmistoa ei suunnitella riittävän yleispäteväksi.

- **Suorituskyvyn optimointi:** Tietokantaratkaisujen väliset erot voivat vaikuttaa suorituskykyyn, ja Virrakkeen on pystyttävä optimoimaan toimintansa eri tietokantaympäristöihin sopivaksi.

- **Ylläpidon monimutkaisuus:** Laajennettavuuden ja integraation tukeminen tarkoittaa sitä, että Virrake saattaa tarvita säännöllisiä päivityksiä eri tietokantaversioiden ja teknologioiden tuen varmistamiseksi.

Integraation ja laajennettavuuden huomioiminen refaktoroinnin yhteydessä mahdollistaa Virrakkeen tehokkaan hyödyntämisen monenlaisissa tietokantaympäristöissä ja parantaa sen arvoa asiakkaiden näkökulmasta.

9 VERSIONHALLINTA

Versionhallinta on palvelu, joka tallentaa tiedostojen muutokset muistiin ja mahdollistaa niiden palauttamisen aiempiin versioihin tarvittaessa. Keskeisiä hyötyjä ovat etenkin varmuuskopiointi, koodin jakaminen sekä useampiin projekteihin osallistuminen ja ryhmätyöskentelyn helpottaminen. Koska versionhallinnan avulla voidaan tallentaa useampi versio projektista, voi uusia ominaisuuksia rakentaessa ja kokeiltaessa helposti palata aikaisempaan versioon, jos ominaisuus ei ollutkaan halutun kaltainen. (Versionhallinta ja Git s.a.)

Versionhallinnassa on mahdollista luoda useita haaroja (branch), jotka toimivat kopioina lähdekoodin eri versioille ja joihin voidaan tehdä muutoksia itsenäisesti. Näin jokainen kehittäjä voi työskennellä omien ominaisuuksiensa parissa tai korjata virheitä ilman, että muutokset vaikuttavat muiden kehittäjien työhön. Muutokset yhdistetään alkuperäiseen lähdekoodiin vasta, kun kaikki muutokset ovat tarkistettu ja hyväksytty. Näin koodi pysyy hyvin organisoituna ja samalla kehitysprosessi ja tiimityöskentely paranee. (Version Control Systems 2022.)

Versionhallinta on keskeinen osa ohjelmistokehityksen elinkaarta (SDLC), jonka avulla voidaan parantaa projektinhallintaa ja ohjelmistotuotantoa. Ilman versionhallintaa riskeinä ovat mm. tietojen menetys, hidas kehitys sekä heikompi koodin laatu. Keskeisimpiä ominaisuuksia versionhallintajärjestelmässä ovat (What is Version Control? s.a.):

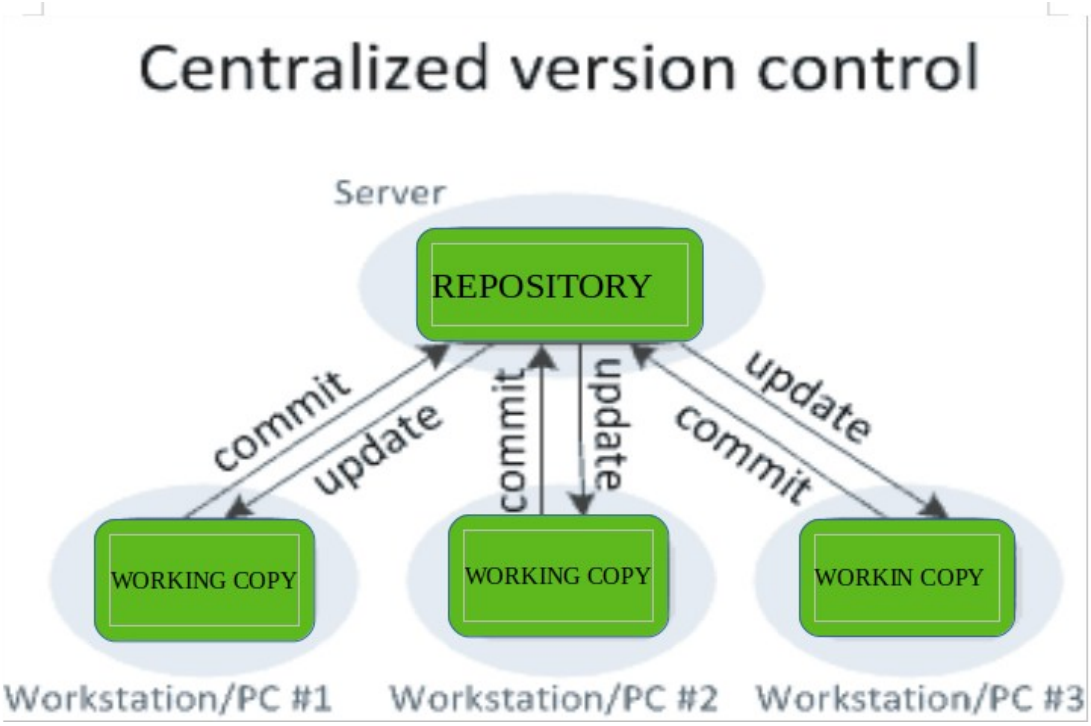
- **Repository (repo, tietovarasto)**: Keskitetty tietokanta, joka varastoi kaikki kooditiedostot, kansiot ja niiden muutoshistorian.
- **Pull Request:** Kehittäjien tapa ehdottaa, tarkastella ja keskustella muutoksia ennen niiden yhdistämistä pääkoodiin.
- **Commit (versiotallenne):** Muutosten tallentaminen paikalliseen arkistoon, voi sisältää viestejä ja muistiinpanoja muilta kehittäjiltä.
- **Branch (haara):** Oma versionsa koodista, jossa kehittäjät voivat tehdä muutoksia itsenäisesti vaikuttamatta pääkoodiin. Käytetään esimerkiksi uusien ominaisuuksien kehittämiseen ja testaamiseen.
- **Merge (yhdistäminen):** Koodimuutosten liittäminen yhteen haaraan tai pääkoodiin.
- **Conflict (konflikti):** Konflikti voi syntyä, jos usea kehittäjä tekee samanaikaisesti muutoksia samaan kohtaan koodissa, ja muutokset ovat ristiriidassa toistensa kanssa. Versionhallintatyökalu auttaa ratkaisemaan tällaiset ristiriidat.
- **Checkout (nouto, tarkastelu):** Kun kehittäjä hakee ja avaa tietyn version koodista muokatakseen sitä.
- **Tag (tunniste):** Merkintä, jolla voidaan merkitä tärkeitä kohtia koodin historiassa, kuten julkaistut versiot.

- **Remote (etätietovarasto, etäkehitys):** Mahdollisuus työskennellä koodin parissa etänä, esimerkiksi pilvipalvelimella.

- **Fork (haaroitus):** Olemassa olevan projektin kopioiminen erilliseen arkistoon. Näin voidaan tehdä muutoksia ilman vaikutusta alkuperäiseen projektiin.

- **Revert (palautus):** Tämä toiminto voi kumota viimeisimmät muutokset ja palata aiempaan versioon.

Versionhallintajärjestelmät voidaan jakaa (kuva 14) paikallisiin, keskitettyihin ja hajautettuihin järjestelmiin. Hajautettu versionhallinta (DVCS), kuten Git, on suosituin, sillä se mahdollistaa kehittäjille työskentelyn missä tahansa ja tarjoaa jokaiselle käyttäjälle oman kopion tietovarastosta (repository). (What is Version Control? s.a.)



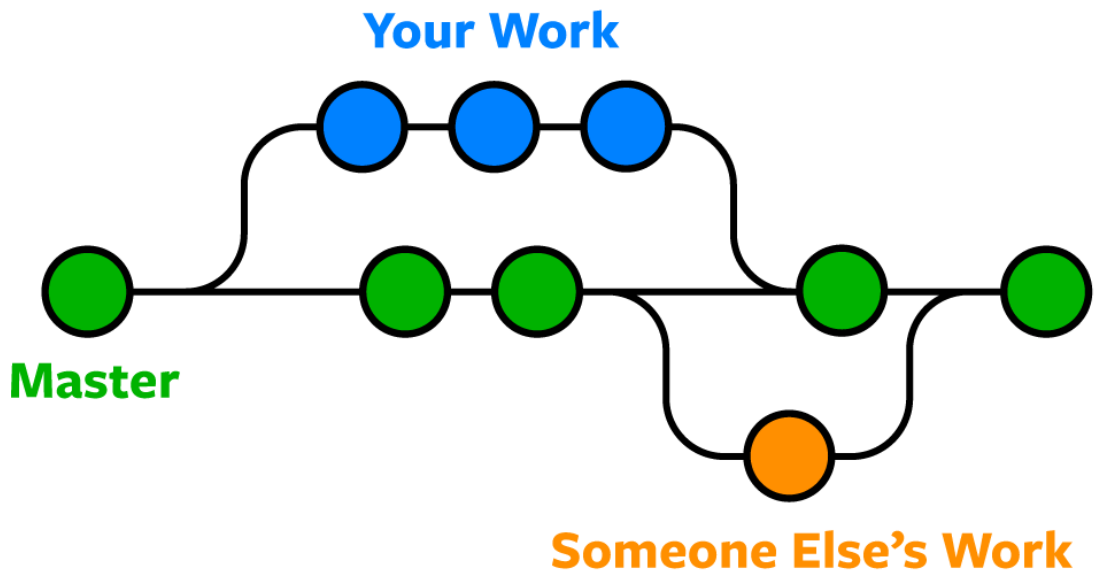
Kuva 14. Versionhallinnan toimintaa havainnollistava kuva (Version Control Systems 2022)

Kuvassa 14 esitetään, miten keskitetyssä versionhallinnassa kehittäjät työskentelevät paikallisilla kopioilla ja synkronoivat muutoksensa yhteen keskitettyyn tietovarastoon. Jokainen kehittäjä voi lähettää omat päivityksensä palvelimelle ja hakea sieltä muiden tekemät muutokset. Tämä takaa, että kaikki näkevät aina ajantasaisen projektiversion ilman erillisiä paikallisia säikeitä.

9.1 GIT

Git on vuonna 2005 kehitetty (Devmountain s.a.) yleisimmin käytössä oleva versionhallintajärjestelmä, joka seuraa tiedostojen muutoksia ja tallentaa projektin versiohistorian ja tekee yhteistyöstä useampien kehittäjien kesken helpompaa. Git antaa kehittäjille mahdollisuuden tarkastella, palauttaa tai yhdistää koodimuutoksia helposti. Haaroittamisen avulla kehittäjät voivat luoda itsenäisiä haaroja pohjakoodista kokeillakseen uusia ideoita toisessa haarassa, mutta tarvittaessa palata takaisin aikaisempiin haaroihin. Järjestelmä ajetaan paikallisesti tietokoneella, joten kaikki tiedostot ja historia ovat tallennettuna paikallisesti oman koneen tietovarastoon. Näin Gitiä voidaan hyödyntää myös ilman internetyhteyttä. (Noble Desktop 2023.)

Tietovarasto voi myös olla palvelimella tai toisen kehittäjän tietokoneella, jolloin sitä voidaan käyttää myös etänä (kuva 15). Tämä antaa tiimeille mahdollisuuden työskennellä useampien asioiden parissa samanaikaisesti, mikä tekee yhteistyöstä helpompaa ja nopeampaa. Git kuitenkin myös lisää hieman haasteellisuutta: koodimuutosten paikallinen versiotallenne (commit) sekä koodimuutosten puskeminen (push) etänä olevaan tietovarastoon ovat eri asioita. Tämän unohtaminen voi luoda ongelmia, joissa kehittäjät työstävät eroavaa pohjakoodia. (Heller 2024.)



Kuva 15. Gitin toiminta visualisoituna (Noble Desktop 2023)

Gitissä tiedostoilla on kolme tilaa: committed, modified ja staged. Commit on yksi tärkeimmistä asioista Gitin käytön kannalta, sillä sen avulla tieto tallennetaan projektiin versiotallenteena. Kun tiedoston tila on committed, tarkoittaa se sitä, että tiedosto on tallennettu paikallisesti juuri sellaisena, kuin se tallennushetkellään on. Jos tiedoston tila on modified, tiedostoon on tehty tallentamattomia muutoksia. Staged puolestaan kertoo, että muokattu tiedosto on merkitty tallennettavaksi seuraavassa commitissa paikallisesti. (Versionhallinta ja Git s.a.)

9.2 GitHub

GitHub on pilvipohjainen alusta, jonka avulla kehittäjät voivat tallentaa, jakaa ja tehdä yhteistyötä koodin parissa. GitHubissa koodi säilytetään tietovarastossa (repository), joka sisältää projektin tiedostot sekä niiden versiohistorian. Palvelu mahdollistaa koodin esittelyn, muutosten hallinnan sekä muiden kehittäjien tekemien ehdotusten tarkastelun ja kommentoinnin, mikä tukee avoimen lähdekoodin yhteisöä ja tiimityöskentelyä. (GitHub s.a.)

Lisäksi GitHub tarjoaa monipuolisia työkaluja, jotka tukevat koodin hallintaa ja tiimityötä, kuten pull requestit, joiden avulla kehittäjät voivat ehdottaa muutoksia, tarkastella koodimuutoksia ja keskustella niistä ennen niiden yhdistämistä pääkoodiin. GitHub mahdollistaa myös projektinhallinnan, versionhallinnan laajennukset ja automaattiset CI/CD-prosessit (Continuous Integration/Continuous Delivery), jotka parantavat koodin laatua ja tehostavat kehitysprosessia. Näin GitHub toimii keskeisenä välineenä sekä yksittäisten kehittäjien että kokonaisten tiimien yhteistyössä. (Heller 2024.)

Gitin ja GitHubin ero

Gitin ja GitHubin ero on tärkeää ymmärtää. Git on versionhallintajärjestelmä, joka seuraa tiedostojen muutoksia ja tallentaa koko projektin historian, kun taas GitHub on pilvipalvelu, joka isännöi Git-tietovarastoja ja tarjoaa lisätoimintoja, kuten tehtävienhallinnan ja integraatiot muihin kehitystyökaluihin. Näin

GitHub yhdistää Gitin perustoiminnot ja laajentaa niitä intuitiivisella käyttöliittymällä ja sosiaalisen koodauksen ominaisuuksilla, mikä tekee yhteistyöstä entistä sujuvampaa ja projektinhallinnasta tehokkaampaa. (Devmountain s.a.)

10 CLEAN ARCHITECTURE

Clean Architecture on ohjelmistoarkkitehtuurin malli, joka on saavuttanut laajaa suosiota viime vuosina. Mallia noudattaen pystytään tuottamaan joustavia, skaalautuvia ja helposti ylläpidettäviä järjestelmiä. Mallin perusajatus on erottaa sovelluksen ydinliiketoimintalogiikka ulkoisista teknisistä yksityiskohdista, mikä mahdollistaa ulkoisten ratkaisujen, kuten tietokantojen ja käyttöliittymien, muokkaamisen vaikuttamatta sovelluksen ydintoimintaan. (Bitloops s.a.; Martin 2012.)

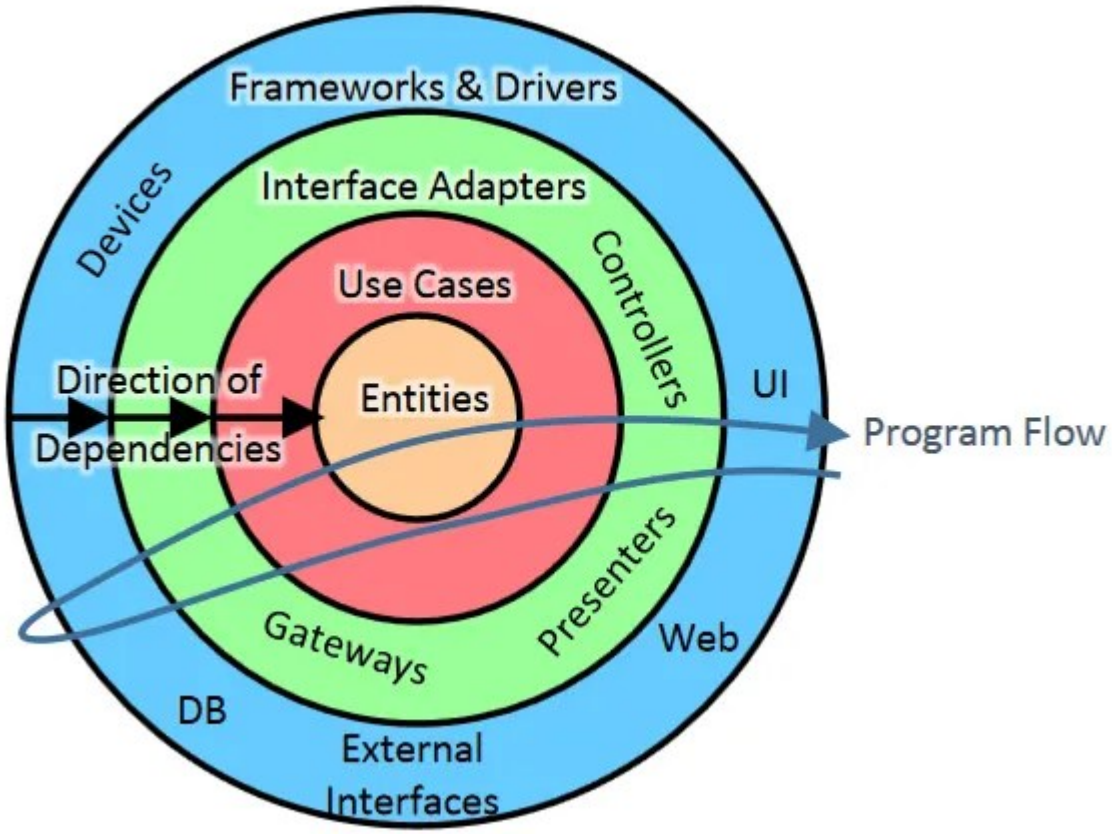
Clean Architecture jakaa sovelluksen useisiin kerroksiin, joissa jokainen kerros huolehtii omasta vastuualueestaan. Tämä kerroksellisuus varmistaa, että ulkoiset riippuvuudet ja tekniset toteutukset eivät sotke ydinliiketoimintaa, mikä puolestaan parantaa sovelluksen selkeyttä ja ylläpidettävyyttä. (Bitloops s.a.) Stackademicin (2024) mukaan Node.js-projektissa toteutuksen rakenne voisi näyttää tältä (kuva 16):

- **Entiteetit (entities)**: Määritellään sovelluksen ydinliiketoiminnan entiteetit ja niiden käyttäytyminen. Pidetään riippumattomina ulkoisista kehyksistä ja kirjastoista.

- **Käyttötapaukset (use cases)**: Toteutetaan sovelluskohtaiset liiketoimintasäännöt käyttötapauksissa. Palveluita käytetään monimutkaisen liiketoimintalogiikan kapselointiin. Hyödynnetään riippuvuuksien injektiota (dependency injection) käyttötapausten irrottamiseksi ulkoisista riippuvuuksista.

- **Rajapintasovittimet (interface adapters)**: Sovitetaan data käyttötapausten ja ulkoisten järjestelmien (esim. tietokannat, kolmannen osapuolen APIt) välillä. Toteutetaan kontrollerit, esittäjät (presenters) ja datamuuntimet (data mappers) kerrosten välisen viestinnän helpottamiseksi. Kehyskohtainen koodi pidetään eristettynä tässä kerroksessa.

- **Kehykset ja ohjaimet (frameworks and drivers):** Sijoitetaan ulkoiset kehykset (kuten Express.js ja tietokannat) ja ohjaimet sovelluksen uloimpaan kerrokseen. Käytetään sovittimia (adapters) ulkoisten komponenttien integroimiseksi sovelluksen muihin osiin.



Kuva 16. Clean Architecture kuvattuna Node.js-ympäristössä (Stackademic 2024)

Clean Architecture on erityisen hyödyllinen monimutkaisissa järjestelmissä, joissa liiketoimintalogiikan vakaus on avainasemassa. Tämä arkkitehtuurimalli mahdollistaa myös sen, että järjestelmän eri osat voidaan testata erikseen, mikä vähentää virheiden todennäköisyyttä ja helpottaa laajennusten tekemistä. Clean Architecture tarjoaa systemaattisen tavan hallita ohjelmiston rakennetta siten, että koodin ylläpito ja kehittäminen onnistuvat tehokkaasti pitkälläkin aikavälillä. (Bitloops s.a.; Martin 2012.)

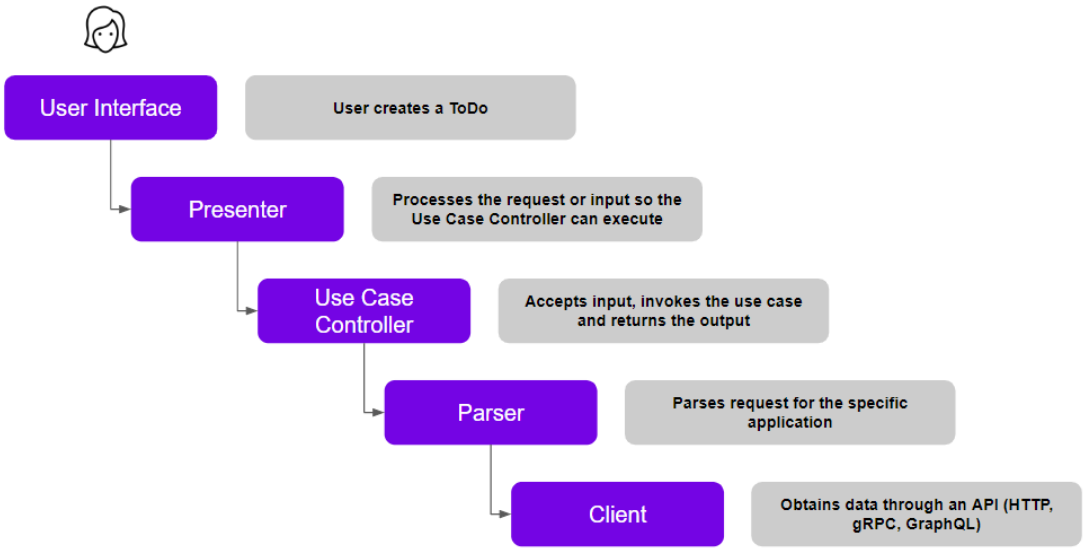
10.1 Periaatteet

Clean Architecture noudattaa useita keskeisiä periaatteita, joista dependency rule on kaikista tärkein. Dependency rule määrittelee, että kaikki ulkoiset riippuvuudet tulee osoittaa vain sisäpuolelle, jolloin sovelluksen ydinkerros pysyy erillisenä ulkoisista teknisistä yksityiskohdista. Minkään sisäkerroksen asian ei tulisi tietää yhtään mitään ulommista kerroksista. (Martin 2012.)

On tärkeää käyttää rajapintoja abstraktioiden esittämiseen, jotta sisäkerrokset

eivät ole riippuvaisia konkreettisista toteutuksista. Dependency rule edellyttää, että kaikki riippuvuudet suuntautuvat kohti vakaita abstraktioita ja poispäin muuttuvista konkreettisista toteutuksista. Tämän ansiosta ulkoisten komponenttien, kuten tietokantojen tai käyttöliittymäkerrosten, muutokset eivät vaikuta sovelluksen ydinlogiikkaan. (Bitloops s.a.; Martin 2012.)

Lisäksi dependency rule (kuva 17) ehkäisee kaksisuuntaisia riippuvuuksia, joissa kaksi kerrosta ovat toisistaan riippuvaisia. Tällaiset riippuvuudet johtavat tiukkaan kytkentään ja vaikeuttavat järjestelmän ylläpitoa ja laajentamista. Oikein toteutettuna riippuvuudet kulkevat aina ulkokerroksista sisäkerroksiin, eivätkä toisinpäin. (Bitloops s.a.)



Kuva 17. Dependency rule kuvattuna yksinkertaisesti (Bitloops s.a.)

10.2 Hyödyt

Clean Architecture tarjoaa merkittäviä etuja ohjelmistokehityksessä. Ensinnäkin se parantaa koodin testattavuutta, koska liiketoimintalogiikka on eristetty ulkoisista riippuvuuksista. Tämä mahdollistaa yksittäisten komponenttien itsenäisen testauksen ja nopean virheiden tunnistamisen millä tahansa sovelluksen tasolla. (Stackademic 2024.)

Toiseksi Clean Architecture edistää ylläpidettävyyttä ja modulaarisuutta. Koska ulkoiset teknologiat ja riippuvuudet pidetään erillään ydintoiminnasta,

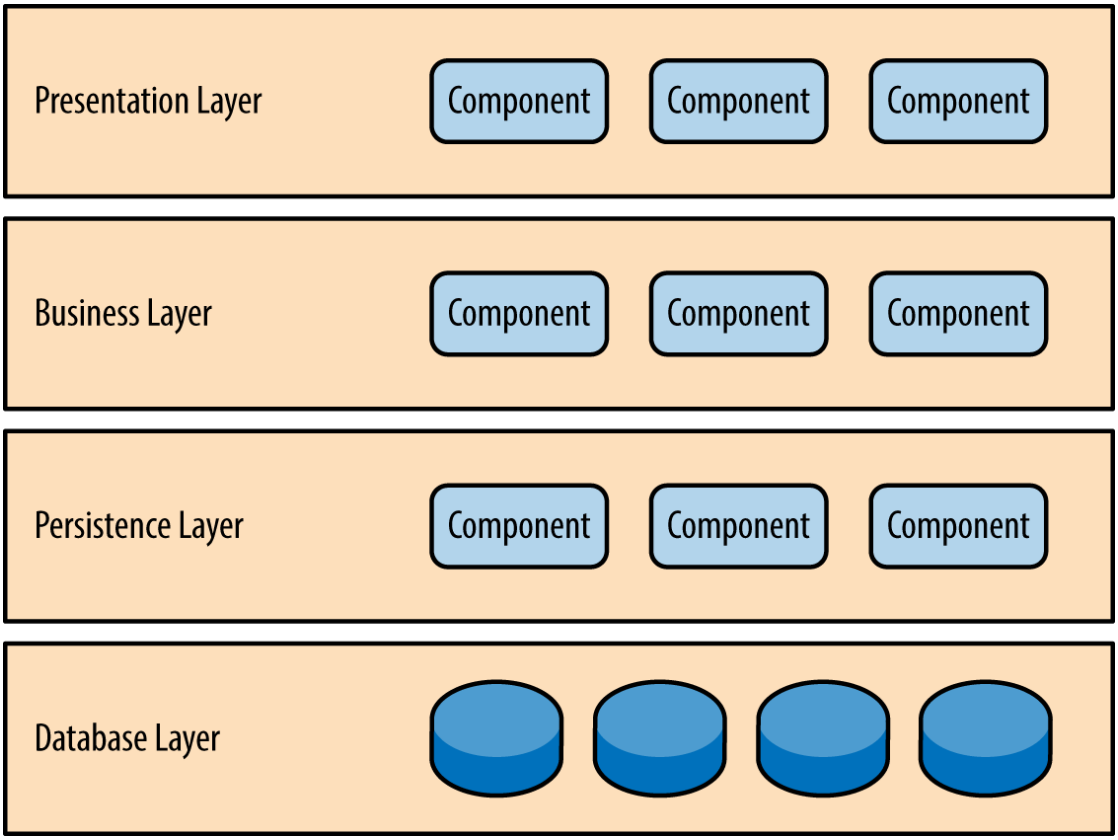
niiden päivittäminen tai vaihtaminen ei vaikuta koko järjestelmän toimintaan. (Bitloops s.a.) Kolmanneksi Gunawan (2024) korostaa, että malli parantaa myös sovelluksen joustavuutta juuri edellä mainituista syistä.

Lisäksi Clean Architecture mahdollistaa riippumattomuuden kehyksistä ja työkaluista, jolloin yhden komponentin toteutuksen voi vaihtaa vaikuttamatta muuhun järjestelmään. Malli myös tukee järjestelmän skaalautuvuutta, koska selkeät rajapinnat ja kerrosarkkitehtuuri mahdollistavat uusien ominaisuuksien lisäämisen ilman, että muut järjestelmän osat kärsivät. Myös ketterä kehitys ja jatkuva toimitus ovat hyvin tuettuna. (Bitloops s.a.)

10.3 Sovelluksen kerrokset

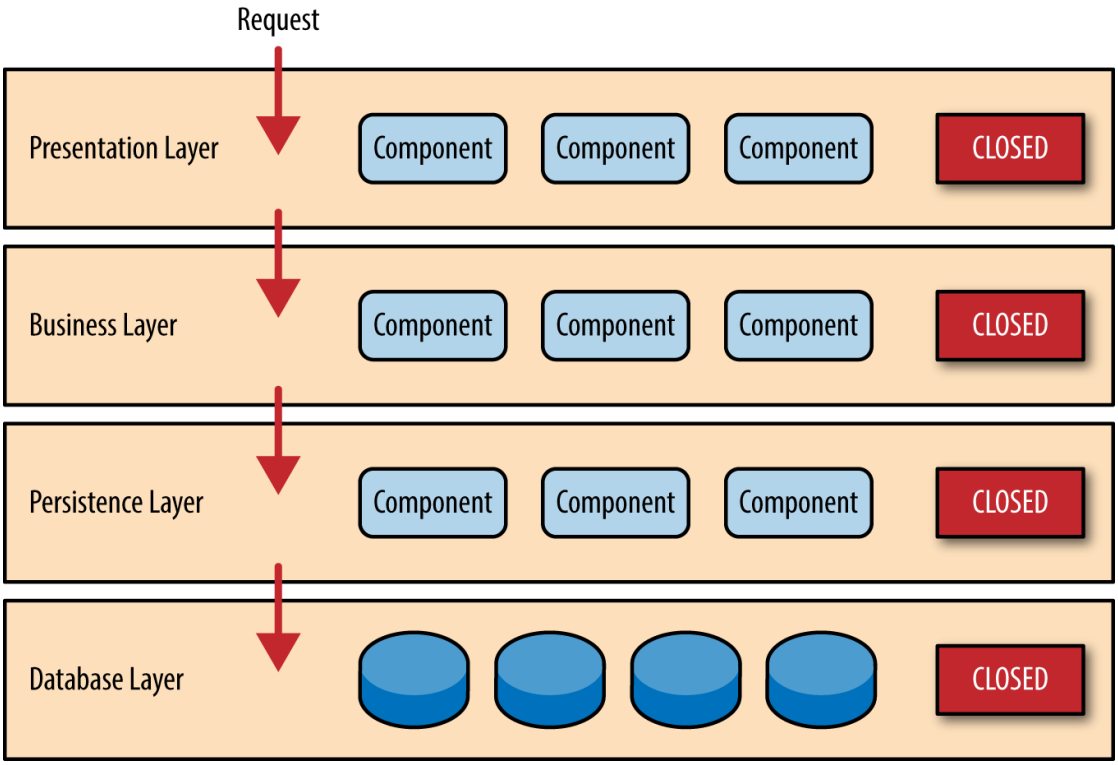
Kun sovellus mukailee kerrostettua arkkitehtuurimallia, se on jaettu päällekkäisiin kerroksiin. Jokaisella kerroksella on nimetty oma tehtävä, jonka sen komponentit täyttävät. Kerrostettua mallia noudattavat sovellukset sisältävät yleensä neljä kerrosta, kuten alla olevista kuvista (kuva 18–21) voi nähdä. Neljän kerroksen malli ei ole kuitenkaan ainoa lähestymistapa, sillä kerroksia voidaan yhdistää tai niitä voi olla enemmän kuin neljä sovelluksen tarpeitten mukaan. (Sofware Architecture Patterns s.a.)

Kerrostetussa sovelluksessa jokainen kerros on riippuvainen sen alapuolella olevista kerroksista. Samaan aikaan kerrokset ovat itsenäisiä niiden yläpuolella olevista kerroksista eivätkä ne ole ”tietoisia” näistä kerroksista. Kerrokset voidaan lisäksi erikseen tehdä suljetuiksi tai avoimiksi. Jos kerrokset ovat suljettuja, tietty kerros kykenee kommunikoimaan suoraan vain lähimpänä alla olevan kerroksen kanssa. Jos kerroksista esimerkiksi yksi on avoin, sen yllä oleva kerros kykenee kommunikoimaan myös avoimen kerroksen alla olevan kerroksen kanssa (kuvat 18–21). (Hgraca 2017; Software Architecture Patterns s.a.)



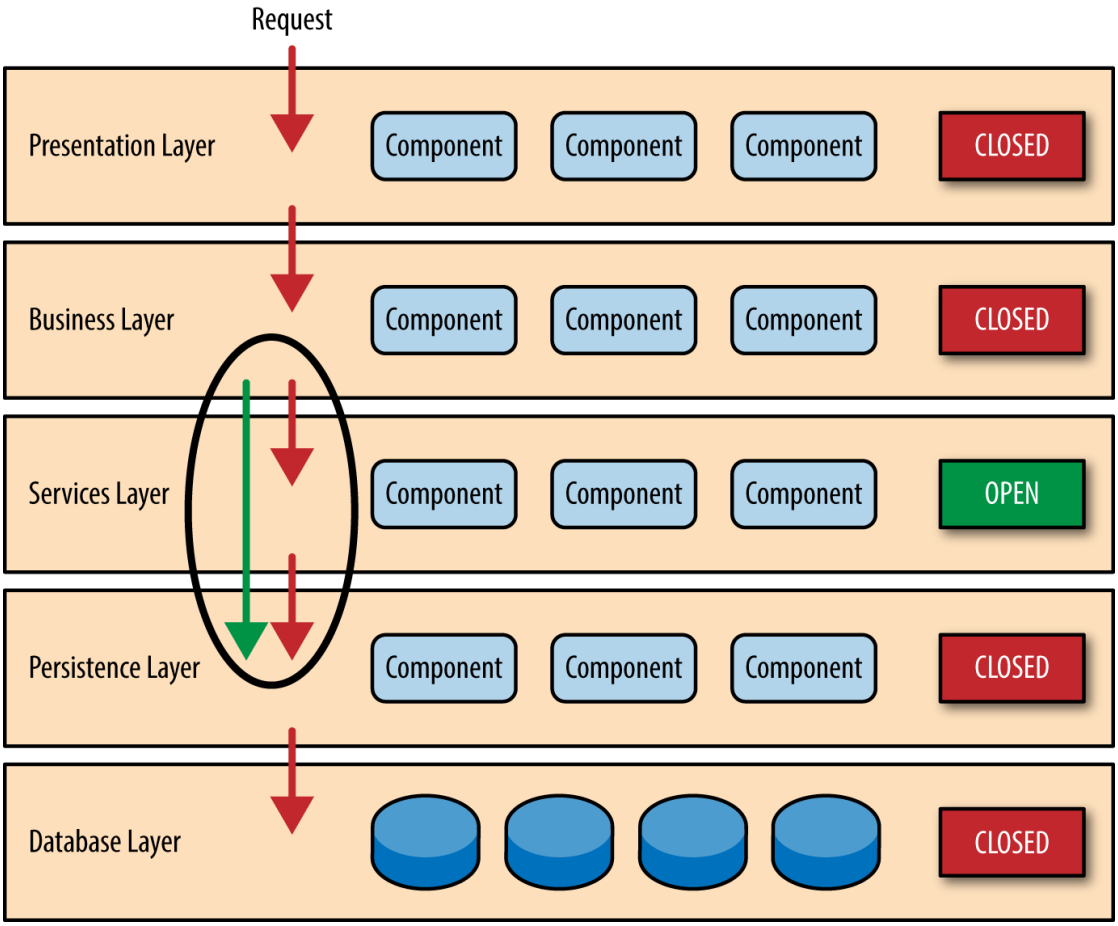
Kuva 18. Esimerkki sovelluksen kerroksista (Software Architecture Patterns s.a.)

Kuvassa 18 esitetään tyypillinen kerroksellinen sovellusarkkitehtuuri, jossa sovellus on jaettu loogisiin kerroksiin: käyttöliittymä (Presentation Layer), liiketoimintalogiikka (Business Layer), pysyväistallennus (Persistence Layer) ja tietokanta (Database Layer). Jokainen kerros sisältää omia komponenttejaan ja vastaa omasta tehtäväalueestaan. Tällainen rakenne parantaa sovelluksen modulaarisuutta, helpottaa ylläpitoa ja mahdollistaa vastuualueiden selkeän erottelun. (Software Architecture Patterns s.a.)



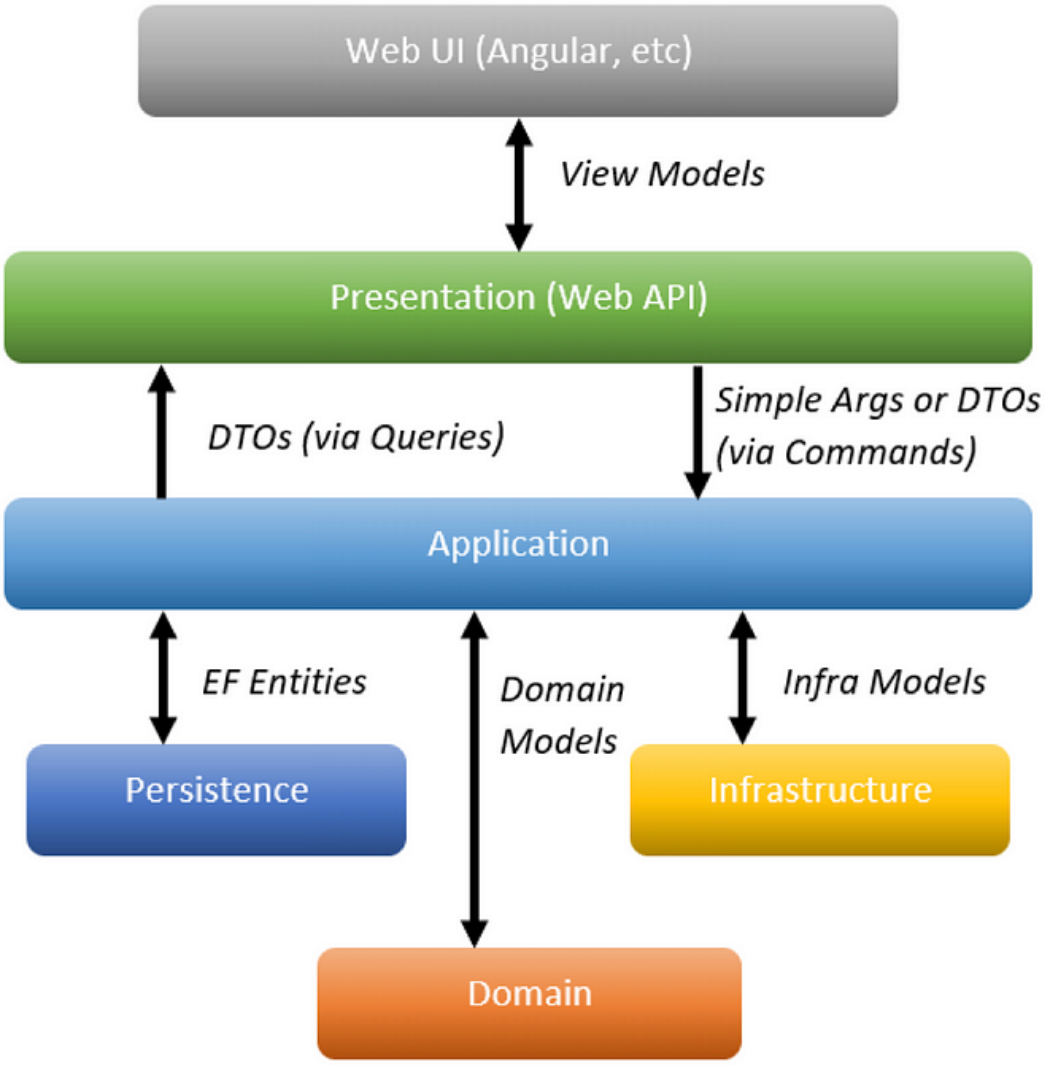
Kuva 19. Esimerkki, kun kaikki kerrokset ovat suljettuja (Software Architecture Patterns s.a.)

Kuvassa 19 jokainen kerros on suljettu (CLOSED), mikä tarkoittaa, että kerrokset voivat kommunikoida vain välittömästi alapuolella olevien kerrosten kanssa. Tämä arkkitehtuurimalli tukee vahvaa kapselointia ja selkeää kerrosvastuunjakoa, mikä helpottaa ylläpitoa ja testattavuutta. Haittapuolena on mahdollinen suorituskyvyn heikkeneminen, jos tietoa täytyy kuljettaa usean kerroksen läpi. (Software Architecture Patterns s.a.)



Kuva 20. Esimerkki, kun yksi kerroksista on avoin (Software Architecture Patterns s.a.)

Kuvassa 20 Services Layer -kerros on määritelty avoimeksi (OPEN), mikä mahdollistaa sen, että ylemmät kerrokset voivat ohittaa Business Layerin ja kommunikoida suoraan tämän kanssa. Tämä voi olla hyödyllistä, kun halutaan uudelleenkäyttää tiettyjä palvelukomponentteja useista eri kerroksista käsin. Toisaalta tämä rikkoo perinteistä kerrosarkkitehtuuria ja voi johtaa tiukempiin riippuvuuksiin ja vaikeampaan hallittavuuteen, jos käytetään liikaa. (Software Architecture Patterns s.a.)



Kuva 21. Kerroksien välinen kommunikointi (Celep 2022)

Syitä jakaa sovellus kerroksiin (Indeed Editorial Team 2024):

- Kehitysprosessi yksinkertaistuu, jolloin myös yleiskustannukset jäävät usein alhaisemmiksi.
- Testaaminen on helpompaa, koska sovellus koostuu erillisistä kerroksista, joita voi testata helposti erillisinä toisistaan.
- Virheiden tunnistaminen ja ongelmanratkaisu helpottuvat, koska voidaan keskittyä yhteen kerrokseen kerrallaan.
- Eri kerrosten eriyttäminen selkeyttää vastuualueita ja tekee siten ohjelmistoarkkitehtuurin roolien määrittelystä tehokkaampaa.

11 TIETOTURVA

Tietoturva eli kyberturvallisuus tarkoittaa toimenpiteitä, joilla suojataan tietokoneita, palvelimia, mobiililaitteita, sähköisiä järjestelmiä, verkkoja ja dataa haitallisilta hyökkäyksiltä. Tietoturvallisuuden tavoitteena varmistaa, että tiedot pysyvät salassa ja eheinä hyökkäyksiä vastaan. Lisäksi tietoturvallisuus pyrkii estämään haittaohjelmien, SQL-injektion, phishing-hyökkäysten sekä muiden tietoturvauhkien aiheuttamat vahingot. (F-Secure 2023; Kaspersky s.a.)

Nykyisessä digitaalisessa ympäristössä uhkat ovat yhä monimuotoisempia ja kehittyneempiä. Organisaatioihin kohdistuu muun muassa monivaiheisia hyökkäyksiä, toimitusketjun murtoja sekä kehittyneitä ransomware-hyökkäyksiä. Näiden uhkien torjumiseksi on tärkeää käyttää moderneja teknologioita ja noudattaa alan parhaita käytäntöjä, jotta järjestelmien ja sovelluksien turvallisuus voidaan varmistaa. (Checkpoint s.a.)

Ohjelmistokehityksessä TypeScriptin hyödyntäminen lisää koodin turvallisuutta tyypityksen avulla ja vähentää injektiohyökkäysten riskiä. Tämä lähestymistapa edistää koodin laadun parantamista ja mahdollisten haavoittuvuuksien havaitsemista ja kehitysvaiheessa. Lisäksi modernien työkalujen, kuten staattisen koodianalyysin ja automaattisen testauksen, käyttöönotto tukee entisestään turvallista ohjelmistokehitystä. (Ignacio s.a.)

Palvelinympäristössä, kuten Node.js-sovelluksissa, tietoturvan takaamiseksi tulee noudattaa tarkkoja turvallisuuskäytäntöjä. Tällaiset käytännöt sisältävät palvelunestohyökkäysten estämisen asianmukaisella virheenkäsittelyllä ja oikeilla aikakatkaisuilla. Lisäksi on tärkeää hallita kolmansien osapuolien riippuvuuksia ja varmistaa, että kaikki käytetyt komponentit ovat ajan tasalla. (Security Best Practices, s.a.)

Tietoturvan merkitys korostuu entisestään, kun yhä useammat kriittiset toiminnot ja infrastruktuurit ovat riippuvaisia tietokonepohjaisista järjestelmistä ja internet-yhteyksistä. Tämä asettaa useita vaatimuksia sekä teknisten ratkaisu-

jen että henkilöstön jatkuvalle koulutukselle, jotta uusia ja kehittyviä uhkia voidaan torjua tehokkaasti. Uusien teknologioiden, kuten tekoälyn ja pilvipalveluiden, käyttöönottaminen tuo mukanaan sekä uusia mahdollisuuksia että haasteita tietoturvan hallintaan. (F-Secure 2023; Kaspersky s.a.)

Jatkuva järjestelmien valvonta, säännölliset päivitykset sekä kokonaisvaltaisen suojauksen lähestymistapa ovat olennaisia tietoturvan kannalta, jotta pysytään askeleen edellä kyberhyökkäyksiä. Näin voidaan reagoida nopeasti mahdollisiin tietoturvauhkien muutoksiin. Integroitu turvallisuusarkkitehtuuri auttaa hallitsemaan monimutkaisia IT-ympäristöjä ja varmistaa, että kaikki suojausratkaisut toimivat saumattomasti yhdessä. (Checkpoint s.a.)

12 TESTAUS

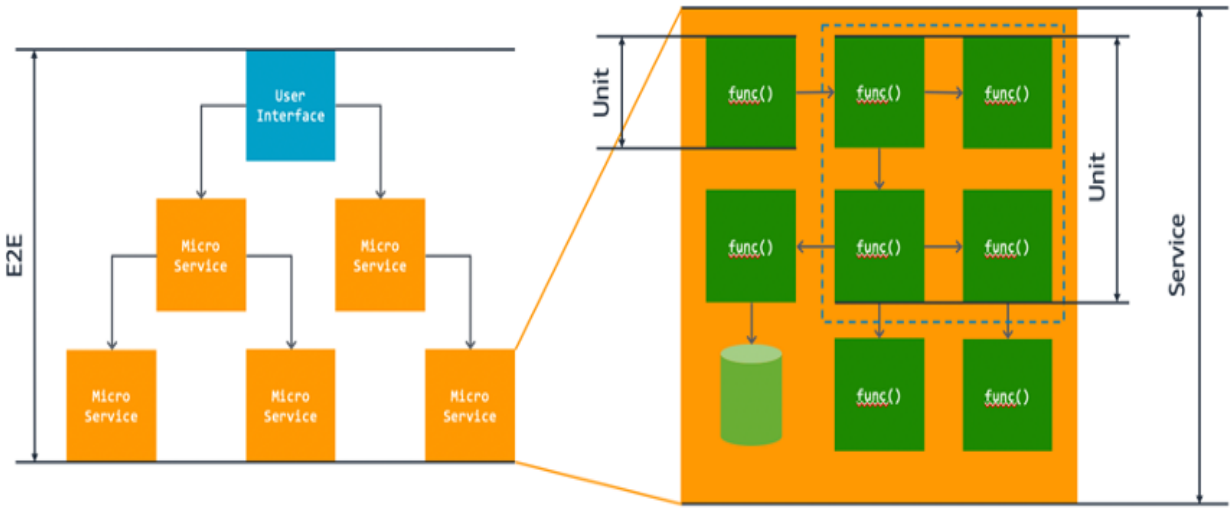
Ohjelmistotestaus on sen varmistamista, että ohjelma tuottaa odotettuja tuloksia rajattujen testitapauksien joukosta. Ohjelmistotestauksella varmistetaan ja validoidaan, että suunnittelun ja kehityksen asettamat vaatimukset täytetään. Tämä menettelytapa pyrkii takaamaan testattavan ohjelmiston kyvyn käsitellä erityis- ja reunatapaukset, jotta käyttäjäkokemus tuntuu mahdollisimman luotettavalta. (Bourque & Fairley 2014, 82; What is software testing? 2025.)

Ohjelmistotestauksen tarkoitus ei ole ainoastaan tuoda esiin virheitä ohjelmistossa vaan mitata sen tehokkuutta ja tarkkuutta eri asioissa. Mittaamalla ohjelmistoa pyritään myös löytämään tapoja parantaa ohjelmistoa entisestään. Testien rajaaminen on tärkeää, koska yksinkertaisissakin ohjelmistoissa on käytännössä niin suuri määrä testitapauksia, että kaiken perusteellinen testaus voi viedä kuukausia tai vuosia (Bourque & Fairley 2014, 82; What is software testing? 2025.)

Lisäksi ohjelmistotestauksessa on tärkeää kyetä päättämään, milloin havainnoidut testitulokset ovat tyydyttäviä. Muutoin testaaminen on hyödytöntä. On erilaisia yleistapoja määrittää, ovatko testitulokset tyydyttäviä: voidaan verrata esimerkiksi tiettyyn määrittelyyn (verifiointitestaus), käyttäjien tarpeisiin (validointitestaus) tai tiettyihin implisiittisiin vaatimuksiin perustuvaan odotettuun tulokseen (hyväksymistestaus). (Bourque & Fairley 2014, 82.)

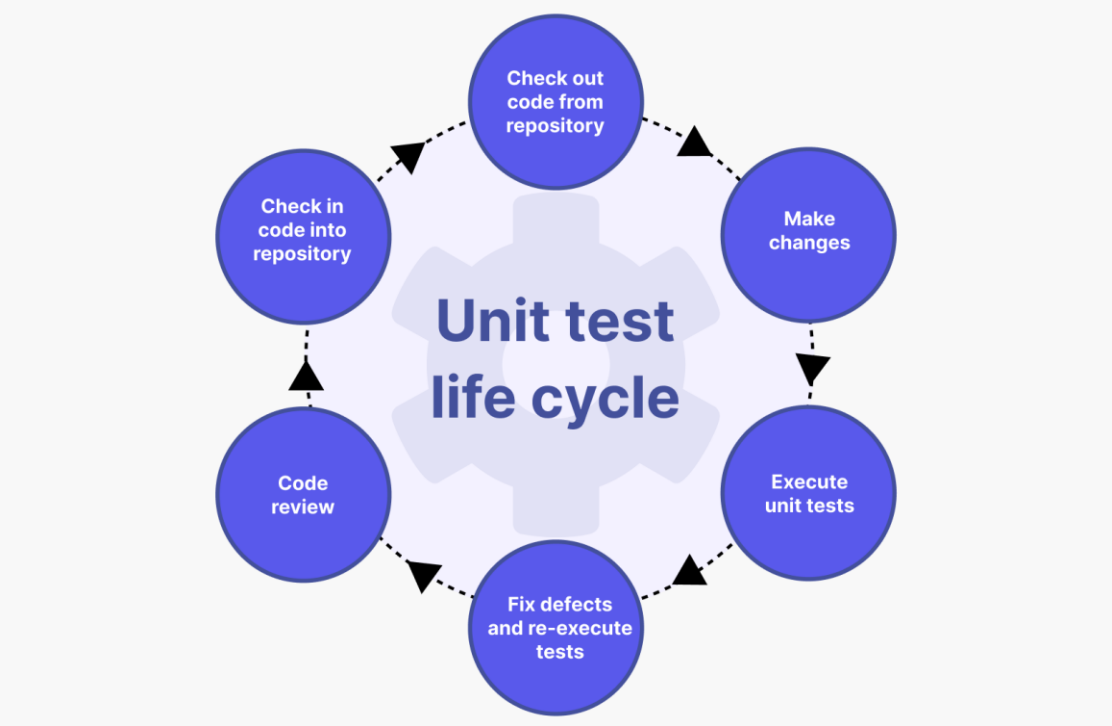
12.1 Yksikkötestaus

Yksikkötestauksessa (kuva 22) testataan pienimpiä toimivia yksikköjä koodista. Yksikkötesti on koodia, jonka tarkoitus on testata funktion, metodin tai muun pienen koodipalasen tehokkaan ja tarkan toiminnan muusta koodista eristetysti. Monet osat koodia vaativat toisen koodin tarjoamaa dataa toimiakseen, minkä takia käytetään ”tynkäversioita” datasta (datastump, mock object) simuloimaan muita koodin osia. Ihanteellista olisi kirjoittaa koodi pienikokoisina funktioina, joita on helppo myöhemmin muokata ja joille on helpompi kirjoittaa yksikkötestit (kuva 23) (What is unit testing? s.a.; Bakharev 2023.)



Kuva 22. Koodin ”yksiköt” kuvattuna (What is unit testing? s.a.)

Kuva 22 havainnollistaa eri testauskerrosten suhdetta: vasemmalla näkyvät laajemman järjestelmän osat (end-to-end, mikropalvelut), kun taas oikealla puolella esitellään tarkempi näkymä yksittäisistä funktioista ja loogisista moduuleista, joita yksikkötesteillä eristetään muusta koodista. Näin korostuu se, että yksikkötestaus kohdistuu vain pienen toiminnallisen palasen toimintaan.



Kuva 23. Yksikkötestaussykli visuaalistettuna (Katalon 2024)

Bakharevin (2023) mukaan yksikkötestauksen edut ovat seuraavat:

- Aikainen virheiden havainnointi
- Alhaisemmat kustannukset
- Helpompi koodin refaktorointi
- Ohjelmiston toiminnan dokumentaatio.

12.2 Testauksen työkalut

Ohjelmistotestauksen työkalut ovat olennaisia nykyaikaisessa ohjelmistokehityksessä. Ne auttavat varmistamaan, että sovellus toimii halutulla tavalla, mahdollistavat virheiden havaitsemisen varhaisessa vaiheessa ja tukevat laadunvalvontaa koko kehitysprosessin ajan. Työkaluja on erilaisia eri testauksen osa-alueille, kuten yksikkötestaukseen, integraatiotestaukseen, suorituskykytestaukseen ja käyttöliittymätestaukseen. (Hamilton 2025; BrowserStack 2025.)

Jest

Jest on JavaScript-testikirjasto, jota käytetään erityisesti yksikkötestaukseen. Sen on kehittänyt Meta (entinen Facebook), ja se on erityisen suosittu React-sovellusten testauksessa, vaikka se toimii hyvin minkä tahansa JavaScript- tai TypeScript-projektin kanssa. Seuraavana on esitelty Jestin keskeisimpiä ominaisuuksia, jotka tekevät siitä suositun valinnan React, JavaScript ja TypeScript projekteissa. (Testing with Jest 2025; Jest 2025.)

1. **Zero configuration**
   Jest toimii heti ilman erityistä määritystä useimmissa projekteissa, mikä tekee siitä helppokäyttöisen myös aloittelijoille.
2. **Snapshot testing**
   Mahdollistaa komponenttien ulkoasun "valokuvauksen" (snapshot) ja vertailee niitä myöhempiin testiajoihin. Tämä on hyödyllistä erityisesti React-komponenteille.
3. **Mocking- ja spy-toiminnot**
   Voit luoda helposti mokkauksia eli keinotekoisia toteutuksia funktioille tai moduuleille, jolloin voit eristää testattavan koodin riippuvuuksista.
4. **Yhdistetty testirunner ja asserttikirjasto**
   Jest sisältää kaiken tarvittavan testien suorittamiseen ja tulosten arviointiin, kuten expect()-funktion.
5. **Yhteensopivuus TypeScriptin kanssa**
   Jest toimii hyvin TypeScript-projektien kanssa esimerkiksi ts-jest-paketin avulla.

13 TOTEUTUS

Projektin ensimmäisessä vaiheessa keskityttiin Virrakkeen backendin perusrakenteen uudistamiseen. Aluksi muunsimme JavaScript-tiedostot TypeScript-tiedostoiksi ja lähdimme ratkomaan integraatio-ongelmia. Tavoitteena oli siirtyä pois suoraan koodissa olevista SQL-lauseista ja hyödyntää nykyaikaisia ORM-ratkaisuja, kuten Sequelizea, joka mahdollistaa joustavamman tietokantakyselyiden hallinnan ja vähentää SQL-injektioiden riskiä. Lisäturvaa saadaan TypeScriptin käytöstä, kuten myös selkeyttä koodin kirjoitukseen. Tämä auttoi välttämään tyypityksiin liittyviä virheitä sekä paransi koodin ylläpidettävyyttä. Tavoitteena oli myös hyödyntää Clean Architecturen tuomia hyötyjä

koodin refaktoroinnissa, mutta tiukan aikataulun takia emme tätä ehtineet toteuttamaan.

13.1 Refaktoroinnin vaiheet

Refaktorointiprosessi eteni vaiheittain viikko kerrallaan. Pidimme koko opinnäytetyön tekemisen ajan kerran viikossa Teams-tapaamisen, jossa puhuimme toimeksiantajan kanssa sen hetkisestä tilanteesta ja suunnitelmista tulevalle viikolle. Useimmiten joka viikolle otettiin yksi selkeä tavoite, jota kohti lähdimme työskentelemään (esimerkiksi yhteen tiedostoon tai pienempään kokonaisuuteen keskittyminen).

13.1.1 Lähtötilanteen kartoitus

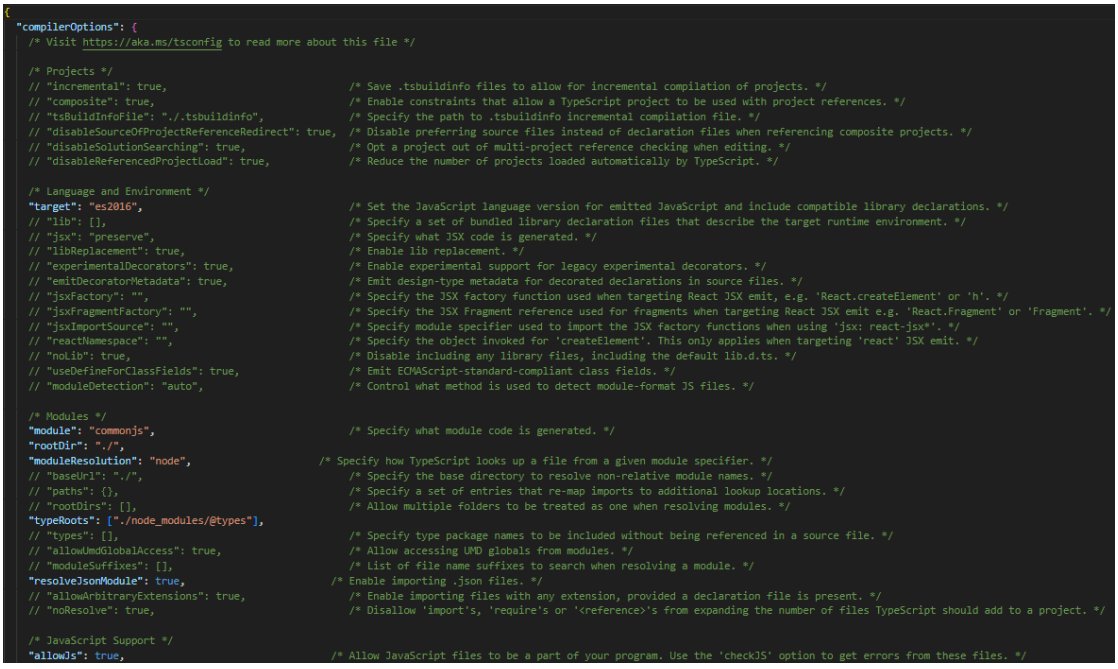
Ensimmäisenä tutustuimme olemassa olevaan JavaScript-koodikantaan ja sen rakenteeseen sekä itse Virrake-sovellukseen. Kävimme toimeksiantajan kanssa läpi suurimpia ongelmakohtia ja toiveita käytettävistä teknologioista. Kovakoodatut SQL-kyselyt olivat aiheuttaneet yhteensopivuusongelmia eri tietokantahallintajärjestelmien, kuten PostgreSQL:n kanssa.

Suunnitelmana oli, että tekisimme migraation JavaScript-koodikannasta TypeScriptiin tulevaisuuden kehitystä varten. Projektissa oli aloitushetkellä melko hajanainen tiedostorakenne ja tyypitysten puuttuminen vaikeutti sen eteenpäin kehittämistä, koska monet virheilmoitukset saataisiin vasta koodin suorituksen aikana.

Suunnitelma oli seuraava: Aloittaisimme migraation toteuttamisen tiedosto kerrallaan. Tiesimme migraation vievän reilusti aikaa joka tapauksessa, koska se oli molemmille uusi asia ja palvelimen koodikanta oli melko laaja käytössä olevaan aikaan nähden. Kun migraation on todettu toimivan, kuten alkuperäinen JavaScript-versio, voitaisiin jatkaa palvelinkoodin ongelmakohtiin.

13.1.2 Migraatio

Edeten suunnitelman mukaan aloitimme TypeScript-migraatiolla. Muunsimme siis tiedostot .ts-päätteisiksi ja lisäsimme projektiin TypeScript-konfiguraatiotiedoston (tsconfig.json, kuva 24). Muutimme myös palvelimen tiedostorakenteen mukailemaan TypeScriptin suosittelemaa rakennetta, jossa on src-kansio, johon siirsimme suurimman osan lähdetiedostoista. Lisäksi TypeScript vaatii ulostulokansion, jonne se voi kääntää .ts- ja .tsx-päätteiset tiedostot JavaScriptiksi tsc-komennolla. Konfiguraatiotiedostossa voidaan määrittää kansio, jonka sisältö halutaan kääntää (src) ja kansio, johon halutaan tallentaa JavaScriptiksi käännetyt tiedostot (built). Kansiolle voi antaa muunkin nimen, mutta yleisesti käytetyt nimet ovat dist, built ja out.



Kuva 24. Tsconfig.json-tiedosto

Kun TypeScript-migraation pohjustus oli tehty, alettiin yksi tiedosto kerrallaan kääntämään JavaScript-tiedostoja TypeScriptiin. Migraation aikana kohtasimme myös haasteita erityisesti tyypitysten yhteensovittamisessa olemassa olevan koodin kanssa. Monissa kohdin oli tarpeen käyttää tilapäisesti any-tyyppiä, jotta kehitystyö pysyi joustavana ja eteni sujuvasti. Nämä any-tyypit pyrittiin kuitenkin korvaamaan tarkemmilla tyyppimäärityksillä projektin edetessä. Lisäksi osa ulkoisista kirjastoista ei sisältänyt valmiita TypeScript-tyypi-

tyksiä, jolloin otimme käyttöön tarvittavat @types-paketit tai loimme omia tyypitystiedostoja (.d.ts.) täydentämään puuttuvia määrityksiä (kuva 25). Tsconfig.json-tiedostoa muokattiin myös myöhemmässä vaiheessa lisää, jotta se sopisi paremmin projektin tarpeisiin. Otimme käyttöön strict-tilan varmistaaksemme mahdollisimman tarkat tyyppitarkistukset.

```
declare module 'express' {
    export interface Request {
        user?: {
            Name?: string;
            Surname?: string;
            IsAdmin?: boolean;
            roles?: string[];
            isAdmin?: () => boolean;
            [key: string]: any;
        };
        isAdmin: () => boolean;
        files?: any;
        logIn?: (user: any, callback: (err: any) => void) => void;
        logout?: (callback: (err: any) => void) => void;
        session: Session & { destroy: (callback: (err: any) => void) => void };
        body: {
            apiKey?: string;
            id?: number | string;
            name?: string;
            department?: string;
            subject?: string;
            message?: string;
            location?: string;
            timestamp?: string;
            locationinworld?: string;
            owningplayer?: string;
            table?: string;
            encodedImage?: string;
            Update?: boolean;
            Delete?: boolean;
            state?: string;
            fromgame?: boolean;
            checkbox?: string;
            Department?: string;
            SelectedMap?: string;
            [key: string]: any;
        };
        params: {
            id?: string;
            [key: string]: any;
        };
    }
```

Kuva 25. Esimerkki .d.ts-tyypitystiedostosta express-moduulille

Kuten edellä on mainittu, TypeScript-migraatio toteutettiin vaiheittain. Aloitimme pienemmistä, irrallisista osista koodikantaa ja etenimme kohti kriittisempiä osia, kuten reititystä ja tietokantakyselyitä. Tämä lähestymistapa auttoi paikantamaan virheitä ja hallitsemaan muutoksia paremmin. Rinnakkain käytettiin hetken aikaa sekä JavaScript- että TypeScript-tiedostoja, jotta sovellus pysyi toimintakykyisenä koko migraation ajan.

13.1.3 Sequelize ORM

Aikaisemmin Virrakkeen tietokantayhteydet perustuivat osittain kovakoodattuihin SQL-kyselyihin, jotka oli kirjoitettu suoraan osaksi sovelluksen logiikkaa. Tämä lähestymistapa teki tietokantarakenteen hallinnasta ja muutosten toteuttamisesta työlästä ja lisäsi virheherkkyyttä erityisesti suurten kyselyiden ja yhdistelyiden kohdalla. Lisäksi kovakoodatut kyselyt vaikeuttivat koodin uudelleenkäyttöä ja testattavuutta, koska tietokantakäsittely oli vahvasti sidottu yksittäisiin funktioihin tai reitteihin.

Refaktoroinnin aikana päätimme siirtyä käyttämään pelkästään Sequelize ORM -kirjastoa, jota Virrake osittain jo käytti. Sequelize tarjoaa korkeamman tason abstraktion tietokannan käsittelyyn ja mahdollistaa kyselyiden rakentamisen ohjelmallisesti JavaScriptin tai TypeScriptin avulla. ORM:n käyttöönotto edellytti tietomallien (models) määrittelyä, joiden pohjalta Sequelize muodostaa taulut ja mahdollistaa tietokantaoperaatiot ilman manuaalista SQL-koodia (kuva 26). Tämä toi selkeyttä ja johdonmukaisuutta sovelluksen tietokantakutsuihin.

```
import { Model, DataTypes, Sequelize } from 'sequelize';

'use strict';

interface DeviceAttributes {
  id?: number;
  name: string;
}

module.exports = (sequelize: Sequelize) => {
  class Device extends Model<DeviceAttributes> implements DeviceAttributes {
    public id!: number;
    public name!: string;

    /**
     * Helper method for defining associations.
     * This method is not a part of Sequelize lifecycle.
     * The `models/index` file will call this method automatically.
     */
    static associate(models: any) {
      // define association here
    }
  }

  Device.init(
    {
      id: {
        type: DataTypes.INTEGER.UNSIGNED,
        autoIncrement: true,
        primaryKey: true,
      },
      name: {
        type: DataTypes.STRING,
        allowNull: false,
      },
    },
    {
      sequelize,
      modelName: 'Device',
    }
  );

  return Device;
};
```

Kuva 26. Esimerkki Sequelizea varten luodusta tietomallista

Muutoksen yhteydessä märittelimme uudet Sequelize-mallit olemassa olevien taulujen pohjalta ja korvasimme vanhat SQL-kyselyt kutsuilla, kuten findAll, findOne, create ja update (kuva 27). ORM:n käyttö vähensi huomattavasti

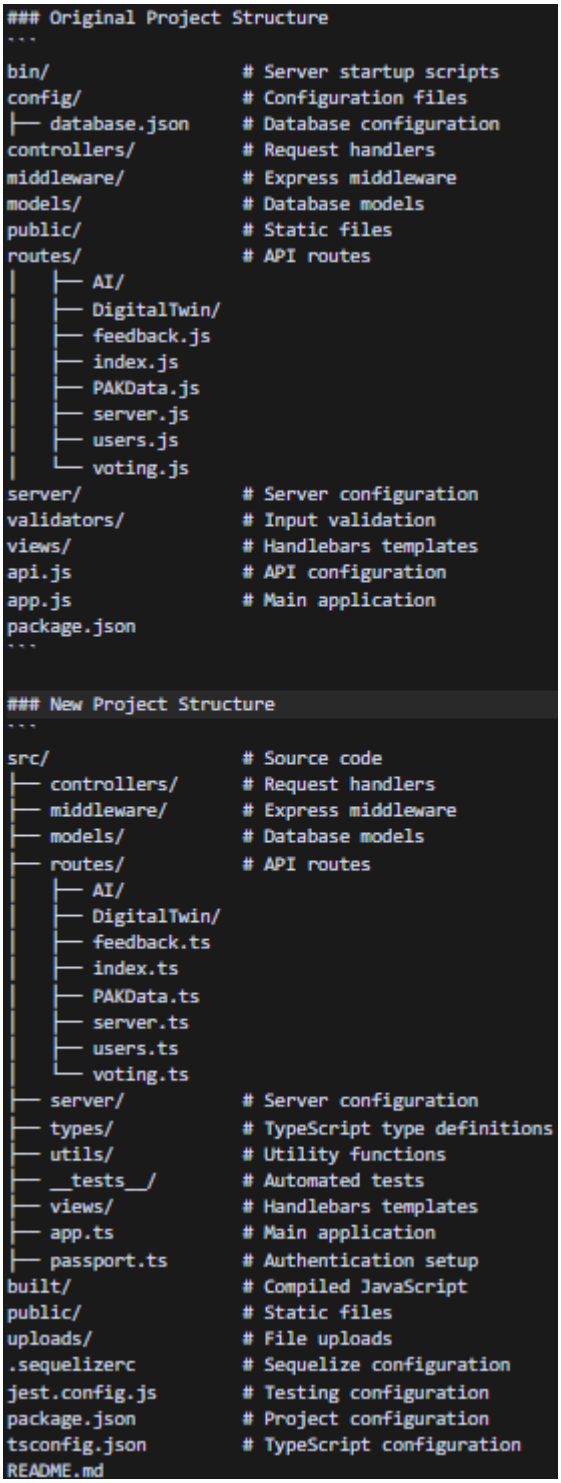
mahdollisuuksia SQL-injektioihin, lisäsi luettavuutta ja helpotti koodin testaamista. Lisäksi ohjelmallinen kyselyiden rakentaminen TypeScriptin kanssa toi tyypityksen tuoman turvan myös tietokantatoimintoihin. Kokonaisuudessaan siirtymä Sequelizeen paransi sovelluksen ylläpidettävyyttä ja mahdollisti tehokkaamman kehitystyön jatkoa varten sekä loi luotettavan pohjan tuleville tietokantamuutoksille.

```
export const getFeedBackMessages = async (
    req: CustomRequest,
    res: Response,
    next: NextFunction
): Promise<Response | void> => {
    try {
        if (!req.body.table) {
            throw new APIError(400, "Missing table parameter");
        }

        const feedbacks = await models.Feedback.findAll({
            where: {
                State: { [Op.ne]: 'Removed' },
                LevelName: req.body.table
            },
            attributes: ['ID', 'LocationInWorld', 'OwningPlayer', 'LevelName']
        });

        return res.json({ Data: feedbacks });
    } catch (error) {
        console.log(clc.red("Error retrieving feedback messages: " + error));

        if (error instanceof APIError) {
            return res.status(error.statusCode).json({
                error: error.message,
                details: error.details
            });
        }

        return res.status(500).json({
            error: "Error retrieving feedback messages",
            details: error instanceof Error ? error.message : 'Unknown error'
        });
    }
};
```

Kuva 27. Esimerkki kontrollerifunktiosta, jossa käytetään Sequelizea

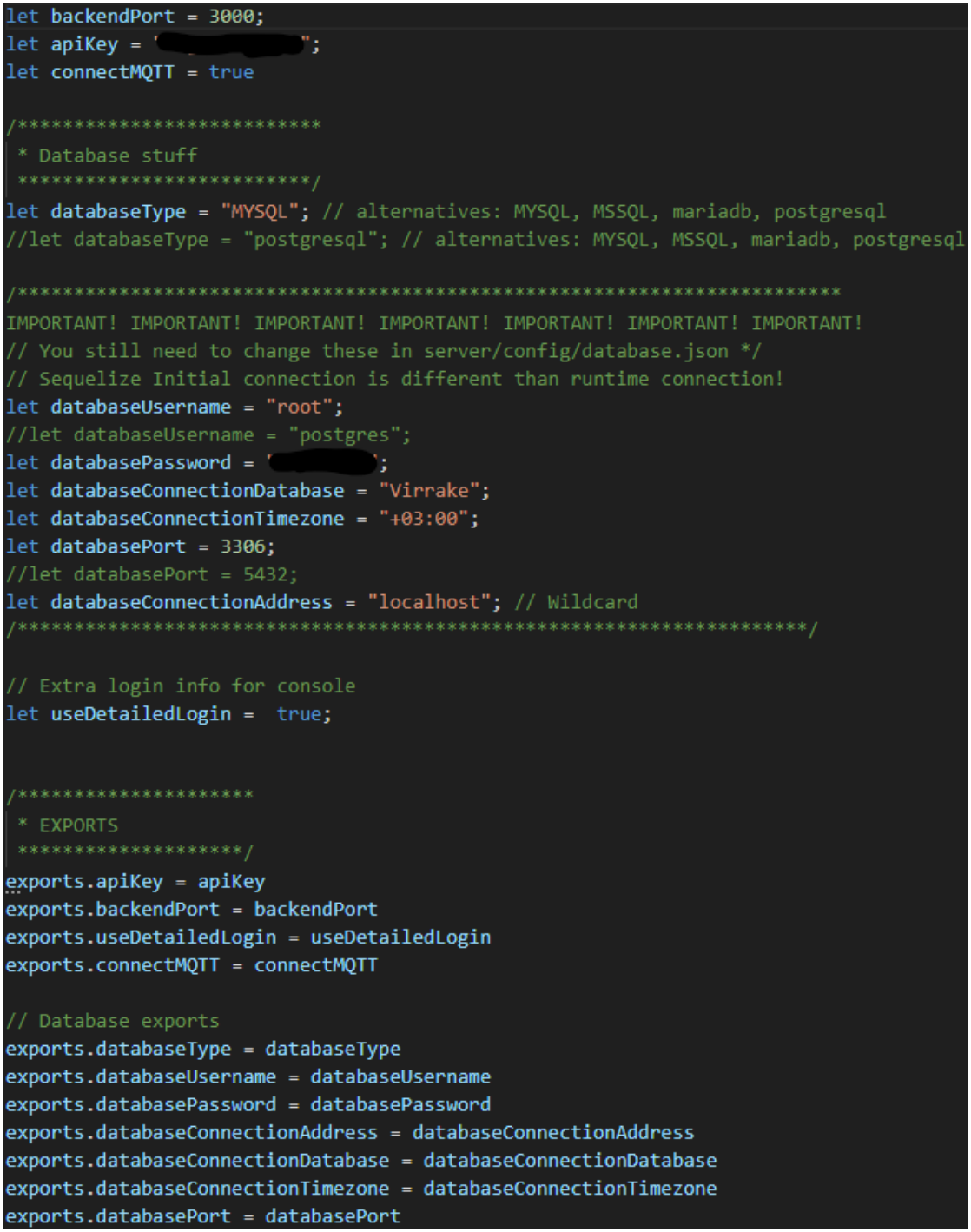
13.1.4 Backend-arkkitehtuuri

Alkuperäisessä versiossa kooditiedostot olivat pääasiassa omissa kansioissaan, mutta kokonaisuuden ja ylläpidon selkeyttämiseksi loimme src-kansion (kuva 28), jonka alle koodi järjesteltiin loogisesti alikansioihin. Näin saimme otettua yhden askeleen kohti Clean Architecturen periaatteita.



Kuva 28. Alkuperäisen version ja uuden version tiedostorakenteen vertailua

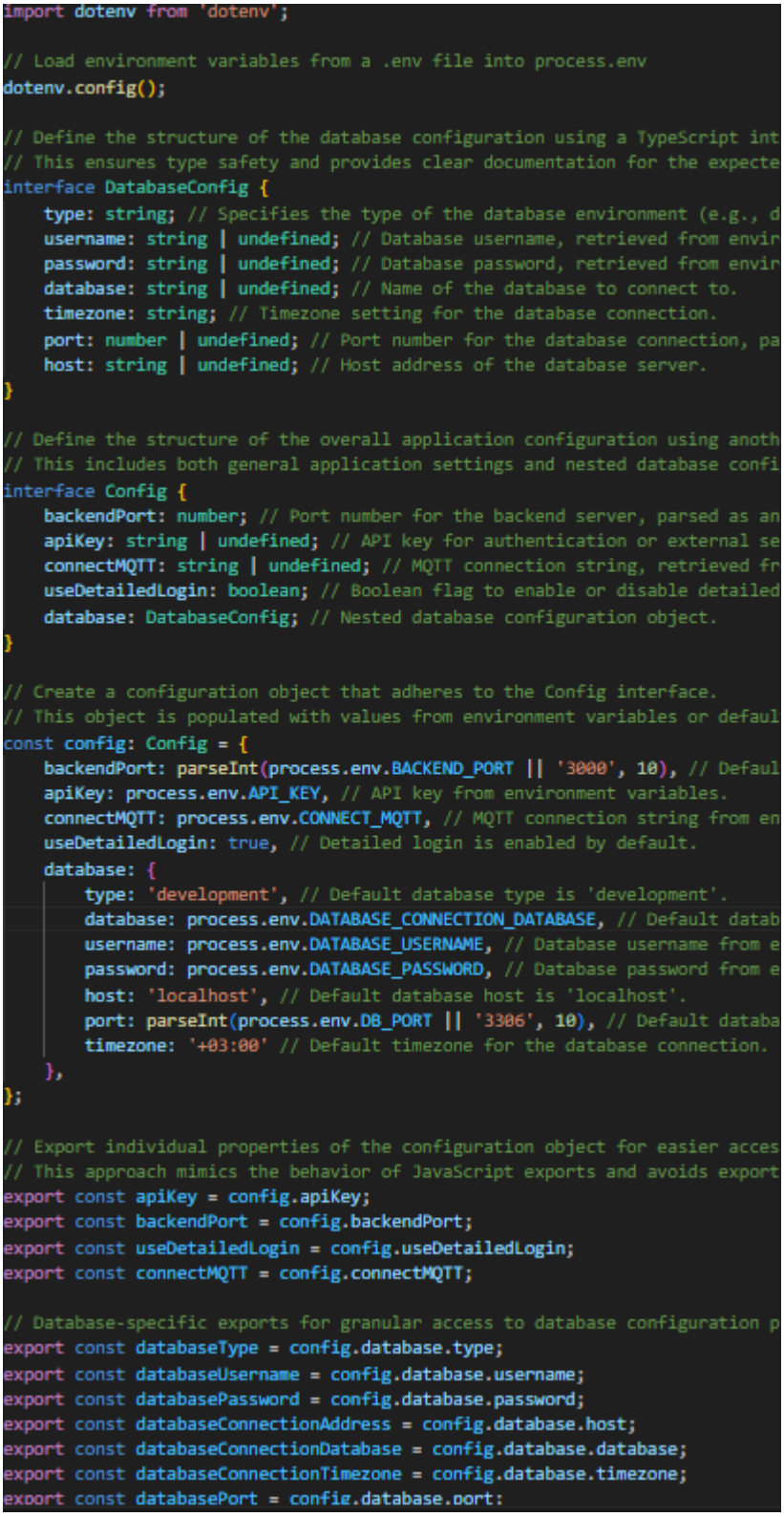
Kuvassa 29 on alkuperäisen version api.js-tiedoston rakennetta. Kuvasta huomaamme heti, että alkuperäisessä versiossa api-avain sekä tietokannan salasana on kirjoitettu suoraan koodiin, mikä heikentää sovelluksen tietoturvaa.



```
let backendPort = 3000;
let apiKey = '[redacted]';
let connectMQTT = true

/**************************
 * Database stuff
 *************************/
let databaseType = "MYSQL"; // alternatives: MYSQL, MSSQL, mariadb, postgresql
//let databaseType = "postgresql"; // alternatives: MYSQL, MSSQL, mariadb, postgresql

/*****************************************************************************
IMPORTANT! IMPORTANT! IMPORTANT! IMPORTANT! IMPORTANT! IMPORTANT! IMPORTANT!
// You still need to change these in server/config/database.json */
// Sequelize Initial connection is different than runtime connection!
let databaseUsername = "root";
//let databaseUsername = "postgres";
let databasePassword = '[redacted]';
let databaseConnectionDatabase = "Virrake";
let databaseConnectionTimezone = "+03:00";
let databasePort = 3306;
//let databasePort = 5432;
let databaseConnectionAddress = "localhost"; // Wildcard
/*****************************************************************************/

// Extra login info for console
let useDetailedLogin =  true;


/*********************
 * EXPORTS
 ********************/
exports.apiKey = apiKey
exports.backendPort = backendPort
exports.useDetailedLogin = useDetailedLogin
exports.connectMQTT = connectMQTT

// Database exports
exports.databaseType = databaseType
exports.databaseUsername = databaseUsername
exports.databasePassword = databasePassword
exports.databaseConnectionAddress = databaseConnectionAddress
exports.databaseConnectionDatabase = databaseConnectionDatabase
exports.databaseConnectionTimezone = databaseConnectionTimezone
exports.databasePort = databasePort
```

Kuva 29. Api.js-tiedoston rakenne alkuperäisessä versiossa

Uudessa versiossa (kuva 30) on otettu käyttöön TypeScript-tyypitykset sekä käytetty ympäristömuuttujia .env-tiedoston kautta. Tämä parantaa tietoturvaa, mahdollistaa kehitys- ja tuotantoympäristöjen erottelun sekä parantaa koodin virheenkäsittelyä jo käännösvaiheessa. Näin saamme arkaluontoiset asiat piilotettua koodista.

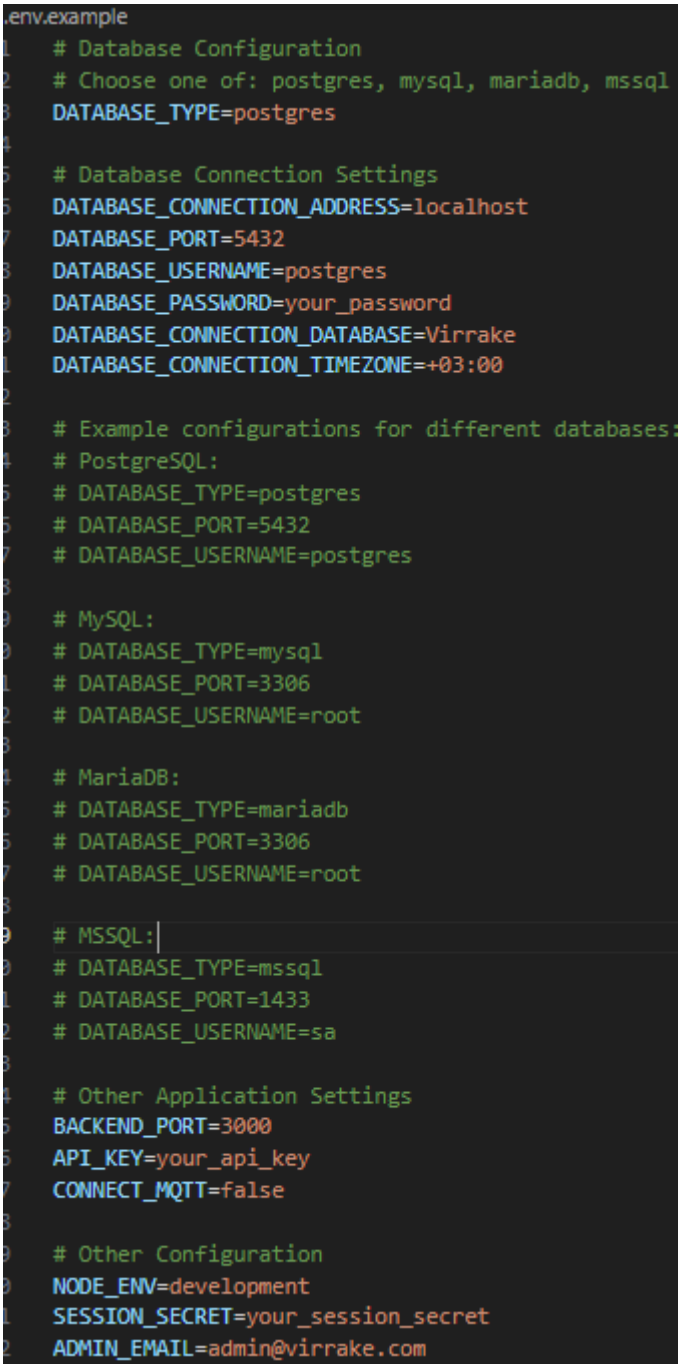


```
import dotenv from 'dotenv';

// Load environment variables from a .env file into process.env
dotenv.config();

// Define the structure of the database configuration using a TypeScript int
// This ensures type safety and provides clear documentation for the expecte
interface DatabaseConfig {
    type: string; // Specifies the type of the database environment (e.g., d
    username: string | undefined; // Database username, retrieved from envir
    password: string | undefined; // Database password, retrieved from envir
    database: string | undefined; // Name of the database to connect to.
    timezone: string; // Timezone setting for the database connection.
    port: number | undefined; // Port number for the database connection, pa
    host: string | undefined; // Host address of the database server.
}

// Define the structure of the overall application configuration using anoth
// This includes both general application settings and nested database confi
interface Config {
    backendPort: number; // Port number for the backend server, parsed as an
    apiKey: string | undefined; // API key for authentication or external se
    connectMQTT: string | undefined; // MQTT connection string, retrieved fr
    useDetailedLogin: boolean; // Boolean flag to enable or disable detailed
    database: DatabaseConfig; // Nested database configuration object.
}

// Create a configuration object that adheres to the Config interface.
// This object is populated with values from environment variables or defaul
const config: Config = {
    backendPort: parseInt(process.env.BACKEND_PORT || '3000', 10), // Defaul
    apiKey: process.env.API_KEY, // API key from environment variables.
    connectMQTT: process.env.CONNECT_MQTT, // MQTT connection string from en
    useDetailedLogin: true, // Detailed login is enabled by default.
    database: {
        type: 'development', // Default database type is 'development'.
        database: process.env.DATABASE_CONNECTION_DATABASE, // Default datab
        username: process.env.DATABASE_USERNAME, // Database username from e
        password: process.env.DATABASE_PASSWORD, // Database password from e
        host: 'localhost', // Default database host is 'localhost'.
        port: parseInt(process.env.DB_PORT || '3306', 10), // Default databa
        timezone: '+03:00' // Default timezone for the database connection.
    },
};

// Export individual properties of the configuration object for easier acces
// This approach mimics the behavior of JavaScript exports and avoids export
export const apiKey = config.apiKey;
export const backendPort = config.backendPort;
export const useDetailedLogin = config.useDetailedLogin;
export const connectMQTT = config.connectMQTT;

// Database-specific exports for granular access to database configuration p
export const databaseType = config.database.type;
export const databaseUsername = config.database.username;
export const databasePassword = config.database.password;
export const databaseConnectionAddress = config.database.host;
export const databaseConnectionDatabase = config.database.database;
export const databaseConnectionTimezone = config.database.timezone;
export const databasePort = config.database.port;
```

Kuva 30. Päivitetty, tietoturvallisempi versio api.ts-tiedostosta

Kuvassa 31 näemme edellä mainitun .env-tiedoston esimerkkirakenteen. Tiedostossa määritellään muun muassa tietokantayhteyden asetukset, portit sekä muut ympäristötiedot. Näiden tietojen kovakoodaaminen suoraan tiedostoon

aiheuttaa suuria tietoturvariskejä ja vaikeuttaa siirtymistä eri ympäristöihin (development/production). .env-tiedostoja käytetään useasti juuri salaisten avainten sekä ympäristökohtaisten asetusten turvalliseen käsittelyyn.



Kuva 31. Esimerkki .env-tiedostosta

Alkuperäisessä koodissa oli myös kovakoodattuja tietokantakyselyitä ja monimutkaisia if/else if -rakenteita (kuva 32). Kovakoodatut tietokantakyselyt heikentävät tietoturvallisuutta ja voivat altistaa sovelluksen SQL-injektiohyökkäyk-

sille. Lisäksi, kuten kuvasta jo huomaakin, pitkät ja haarautuvat if/else if -rakenteet tekevät koodista hyvin vaikealukuista ja ylläpidoltaan haasteellista.

Kuvassa 33 näemme päivitetyn version samasta hasAccess-funktiosta, jossa on otettu käyttöön TypeScriptin tarjoama tyypitys, async/await-syntaksi sekä korvattu kovakoodatut tietokantakyselyt Sequelizen ORM:llä. Lupien tarkastus suoritetaan nyt dynaamisesti ORM:n avulla aikaisemman manuaalisen tarkastelun sijaan, mikä parantaa sekä sovelluksen tietoturvaa että koodin luettavuutta ja jatkokehitettävyyttä merkittävästi.

```
exports.hasAccess = function(menu_name) {
    return function(req, res, next) {
        models.sequelize.query("SELECT MenuName, IsAccessible FROM AccessSettings", { type: Sequelize.QueryTypes.SELECT})
        .then(results => {
        if(menu_name == 'registerUser' && results[0].IsAccessible == 'true' || (req.user && req.user.IsAdmin))
           {
            next();
           }

        else if(menu_name == 'all_users' && results[1].IsAccessible == 'true' || (req.user && req.user.IsAdmin))
        {
            next();
        }

        else if(menu_name == 'create_dummy_users' && results[2].IsAccessible == 'true' || (req.user && req.user.IsAdmin))
        {
            next();
        }

        else if(menu_name == 'check_vote_data' && results[3].IsAccessible == 'true' || (req.user && req.user.IsAdmin))
        {
            next();
        }

        else if(menu_name == 'remove_votes' && results[4].IsAccessible == 'true' || (req.user && req.user.IsAdmin))
        {
            next();
        }

        else if(menu_name == 'check_feedback' && results[5].IsAccessible == 'true' || (req.user && req.user.IsAdmin))
        {
            next();
        }

        else if(menu_name == 'user_screenshots' && results[6].IsAccessible == 'true' || (req.user && req.user.IsAdmin))
        {
            next();
        }
        else if (menu_name == 'overallFeedbacks' && results[7].IsAccessible == 'true' || (req.user && req.user.IsAdmin)) {
           next();
        }

        else if(menu_name == 'check_running_servers' && results[8].IsAccessible == 'true' || (req.user && req.user.IsAdmin))
        {
            next();
        }

        else if(menu_name == 'user_snapshots' && results[9].IsAccessible == 'true' || (req.user && req.user.IsAdmin))
        {
            next();
        }
        else if(menu_name == 'Pakhandling' && (req.user && req.user.IsAdmin))
        {
            next();
        }
        else if(menu_name == 'showServers' && (req.user && req.user.IsAdmin))
        {
            next();
        }
        else if(menu_name == 'showAccessibilitySettings' && (req.user && req.user.IsAdmin))
        {
            next();
        }
        else if(menu_name == 'showRegistrationKeySettings' && (req.user && req.user.IsAdmin))
        {
                next();
        }
        else
            res.redirect('/');
        })
```

Kuva 32. hasAuth.js-tiedoston funktion rakenne alkuperäisessä versiossa

```
export const hasAccess = (permission: string): RequestHandler => {
  return async (req: CustomRequest, res: Response, next: NextFunction): Promise<void> => {
    // Add isAdmin function to request object
    req.isAdmin = function() {
      return this.user?.IsAdmin === true;
    };

    if (!req.user) {
      res.status(401).json({ error: 'Authentication required' });
      return;
    }

    // Admins have access to everything
    if (req.isAdmin()) {
      next();
      return;
    }

    try {
      // Check if user has the specific permission
      const accessSetting = await models.AccessSettings.findOne({
        where: {
          MenuName: permission,
          [Op.or]: [
            { IsAccessible: true },
            { IsAccessible: 'true' }
          ]
        }
      });

      if (accessSetting) {
        next();
      } else {
        res.status(403).json({ error: 'Access denied' });
      }
    } catch (error) {
      console.error('Error checking permissions:', error);
      res.status(500).json({ error: 'Internal server error' });
    }
  };
};
```

Kuva 33. hasAuth.ts-tiedoston funktion päivitetty versio

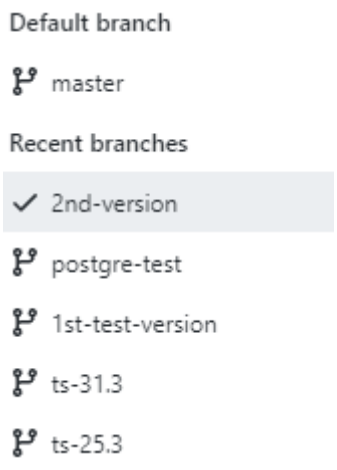
Samankaltaista prosessia toistettiin paljon, jolloin backendin koodista saatiin loppujen lopuksi kauttaaltaan helpommin luettavaa ja ylläpidettävää.

13.2 Tietokantaratkaisut

Yksi suurimmista haasteista opinnäytetyössä oli saada sovellus yhteensopivaksi luvussa 8.1 mainittujen tietokantojen kanssa. Ensimmäinen toimiva refaktoroitu versio luotiin toimimaan etenkin MySQL:llä (ja täten myös MariaDB:llä ja MSSQL:llä). Kun tämä versio todettiin toimivaksi kaikkien tärkeim-

pien toiminnallisuuksien osalta, aloimme työstää yhteensopivuutta PostgreSQL:n osalta.

Alkuun loimme uuden haaran GitHubiin uusimman version pohjalta ja lähdimme työstämään versiota, joka toimisi myös PostgreSQL:n kanssa (kuva 34). Kuvassa 35 näemme esimerkin, kuinka yksinkertaisesti sovellus voi toimia, kun ei tarvitse miettiä kuin yhtä tietokantaa.



Kuva 34. Projektin versionhallintaa

```
export default (sequelize: Sequelize) => {
    AccessSettings.init(
        {
            ID: {
                type: DataTypes.INTEGER,
                primaryKey: true,
                autoIncrement: true,
                allowNull: false
            },
            MenuName: {
                allowNull: false,
                type: DataTypes.STRING
            },
            IsAccessible: {
                allowNull: false,
                type: DataTypes.STRING
            }
        },
        {
            sequelize,
            modelName: 'AccessSettings'
        }
    );

    return AccessSettings;
```

Kuva 35. Esimerkki yhden tietokannan versiosta

Onnistuimme saamaan aikaiseksi omat versiot sovelluksesta, jotka toimivat joko MySQL:n tai PostgreSQL:n kanssa, mutta suurin ongelma oli näiden kahden eri version yhdistäminen yhdeksi sovellukseksi, joka toimisi kaikkien luvussa 8.1 mainittujen tietokantojen kanssa ilman erillisiä koodimuutoksia. Kuvassa 36 näemme esimerkin, jossa muuttujien arvot vaihtelevat sen mukaan, mikä tietokanta on käytössä.

```
export default (sequelize: Sequelize) => {
    const dialect = sequelize.getDialect();

    // Determine column names and types based on dialect
    const isAccessibleType = dialect === 'postgres' ? DataTypes.BOOLEAN : DataTypes.STRING;

    AccessSettings.init(
        {
            ID: {
                type: dialect === 'postgres' ? DataTypes.BIGINT : DataTypes.INTEGER,
                primaryKey: true,
                autoIncrement: true,
                allowNull: false
            },
            MenuName: {
                type: DataTypes.STRING(255),
                allowNull: false
            },
            IsAccessible: {
                type: isAccessibleType,
                allowNull: false,
                get() {
                    const value = this.getDataValue('IsAccessible');
                    if (typeof value === 'string') {
                        return value.toLowerCase() === 'true';
                    }
                    return value;
                },
                set(value: boolean | string) {
                    if (typeof value === 'string') {
                        this.setDataValue('IsAccessible', value.toLowerCase() === 'true' ? 'true' : 'false');
                    } else {
                        this.setDataValue('IsAccessible', value ? 'true' : 'false');
                    }
                }
            },
        },
        {
            sequelize,
            modelName: 'AccessSettings',
            tableName: dialect === 'postgres' ? 'access_settings' : 'AccessSettings',
            underscored: false,
            freezeTableName: true
        }
    );
    return AccessSettings;
```

Kuva 36. Esimerkki useampaa tietokantaa tukevasta koodista

Koska syntaksi vaihtelee käytetyn tietokannan mukaan, toi tämä mukanaan omia ongelmiaan. Siksi toteutuksen tässä vaiheessa tämä tietokannasta riippumaton versio on työn alla.

13.3 Testit

Testejä tehdessämme keskityimme lähinnä yksikkötesteihin ja integraatiotesteihin, jotka testaavat sovelluksen tärkeimpiä toimintoja. Keskityimme etenkin testeihin, jotka varmistavat, että sovelluksen päätoiminnot toimivat ongelmitta ja tietoturvallisesti. Näihin päätoimintoihin kuuluu muun muassa sisäänkirjautuminen, palautteen lähettäminen sekä SQL-injektioilta suojautuminen.

Kuvassa 37 näemme yhden esimerkin testistä, jossa testataan, estääkö sovellus SQL-injektiot, jos käyttäjä rekisteröityessään yrittää injektioita toteuttaa. Testattavia SQL-lauseita on useampi, ja testi varmistaa, että yksikään näistä ei käy käyttäjänimenä.

```
describe('Security Tests - SQL Injection', () => {
  beforeAll(() => {
    const { models } = initModels(mockSequelize);
    setModels(models);
  });

  beforeEach(async () => {
    await mockSequelize.sync({ force: true });
  });

  // SQL Injection test cases for user registration
  describe('User Registration SQL Injection Tests', () => {
    const sqlInjectionPayloads = [
      "' OR '1'='1",
      "'; DROP TABLE Users; --",
      "' UNION SELECT * FROM Users; --",
      "'; WAITFOR DELAY '0:0:5'; --",
      "' OR 1=1; --",
      "admin' --",
      "' OR '1'='1' --",
      "' OR '1'='1' #",
      "' OR '1'='1'/*",
      "'; SELECT * FROM Users; --"
    ];

    test.each(sqlInjectionPayloads)('should reject SQL injection attempt in username: %s', async (payload) => {
      const response = await request(app)
        .post('/users/registerUser')
        .send({
          name: 'Test',
          surname: 'User',
          username: payload,
          email: 'test@example.com',
          password: 'testpass123',
          department: 'Engineering',
          bio: 'Test bio',
          phone: '123456789',
          gender: 'Other',
          departmentkey: '12345'
        });
```

Kuva 37. SQL-injektiotesti

Jotta testi menee hyväksytysti läpi, täytyy jokainen SQL-lause evätä rekisteröi-

tymisvaiheessa. Samankaltaisia SQL-injektiotestejä teimme myös muun muassa palautetoiminnolle.

Palautetoiminnolle teimme myös testejä, jotka varmistavat, että vain oikeanlainen palaute, jossa on kaikki tarvittavat tiedot, pääsee läpi tietokantaan. Edellä mainittujen SQL-injektioiden lisäksi testit muun muassa varmistavat, että esimerkiksi API-avain on toimiva, jotta palaute voidaan lähettää ja tallentaa tietokantaan (kuva 38). Jos API-avain ei ole validi, ei palautetta hyväksytä lähetettäväksi, ja käyttäjälle palautetaan vikakoodi 403 (Forbidden).

```
test('should reject feedback with invalid API key', async () => {
  // Test that feedback is rejected if the API key is invalid
  const feedbackPayload: FeedbackPayload = {
    id: 1003,
    apiKey: 'invalid-key',
    name: 'Test User',
    department: 'Engineering',
    subject: 'Test Subject',
    message: 'This is a test feedback',
    location: 'Test Location',
    timestamp: new Date().toISOString(),
    locationinworld: 'TestWorld',
    owningplayer: 'TestPlayer',
    table: 'TestLevel'
  };
  const response = await request(app)
    .post('/feedback/insertFeedbackToDatabase')
    .send(feedbackPayload);
  expect(response.status).toBe(403);
});
```

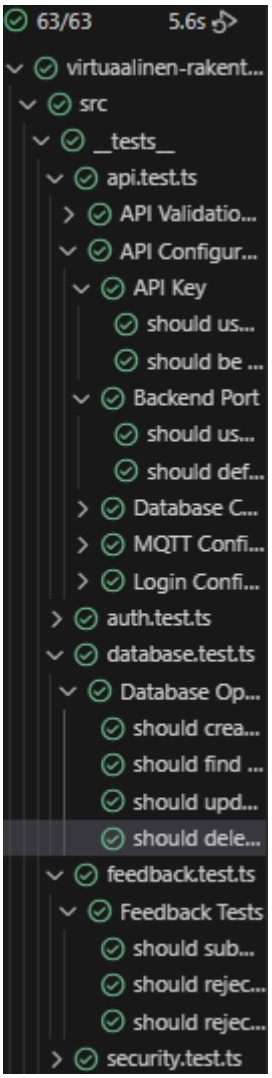
Kuva 38. API-avaintesti palautteelle

Lisäksi teimme testejä käyttäjän luomisen, rekisteröitymisen ja sisäänkirjautumisen testaamiseen. Käyttäjän rekisteröityminen jo käytössä olevalla sähköpostilla (kuva 39) estetään, ja ilmoitetaan, että sähköposti tai käyttäjänimi on jo käytössä. Tämän lisäksi testit myös varmistavat, että kaikki tarvittavat kentät on täytetty, käyttäjä voi rekisteröityä sekä kirjautua sisään ja väärillä tunnuksilla kirjautuminen estetään.

```
test('should reject registration with existing email', async () => {
  // Test that registering with an existing email/username is rejected
  const testUser = createTestUser('3');
  await request(app)
    .post('/users/registerUser')
    .send(testUser);
  // Try to register again with same email/username
  const response = await request(app)
    .post('/users/registerUser')
    .send(testUser);
  expect(response.status).toBe(200); // Should not redirect
  expect(response.text).toContain('Email or username is already in use. Please try again');
});
```

Kuva 39. Estää käyttäjän rekisteröitymisen, jos sähköposti on jo käytössä

Kaiken kaikkiaan testitiedostoja syntyi viisi kappaletta. Testitapauksia tuli kaikkinensa 63 kappaletta (kuva 40), koska jotkin testit testaavat useammalla eri syötteellä samaa asiaa. Etenkin SQL-injektiot paisuttivat testitapauksien määrää. Kuvasta 40 näemme myös, että kaikki 63 testitapausta on suoritettu onnistuneesti.



Kuva 40. Läpi menneet testit

Vaikka testitapauksia syntyikin runsaasti, ei testien kattavuus silti päässyt aivan haluamallemme tasolle (kuva 41). Saimme suoritettua testit sovelluksen avainosioille, johon voimme kuitenkin olla tyytyväisiä. Puutteita testien kattavuuteen kuitenkin edelleen jäi.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	38.16	12.64	18.81	36.34	
src	80.8	38.46	36.36	80.64	
HandlebarHelpers.ts	47.05	100	10	47.05	14,20-41
api.ts	100	75	100	100	31
app.ts	88.23	16.66	60	88.05	46-47,107,129,135-140
passport_setup.ts	73.07	33.33	57.14	73.07	30,35,44-51

Kuva 41. Testien kattavuus

Aikataulun tiukkuuden vuoksi emme ehtineet toteuttamaan testejä, jotka kattaisivat sovelluksen kokonaisuudessaan. Juurikansion koodin testikattavuus on kuitenkin 80,8 %, kuten edellä olevasta kuvasta 41 näemme, ja olemme varmistaneet, että tärkeimmät toiminnallisuudet sisältyivät testaukseen.

14 LOPETUS

Opinnäytetyön tekeminen alkoi vuoden alussa erittäin nopeasti ja sujuvasti. Aiheen saimme ohjaajaltamme, joka toimi samaan aikaan myös toimeksiantajana. Päätös tehdä opinnäytetyö parina oli todella hyvä. Projektin aikana kävi selväksi, kuinka tärkeää ajankäytön arviointi ja oikeiden työkalujen valitseminen on.

Refaktorointiprosessi sujui pääosin suunnitellusti, mutta työmäärän arviointi oli haastavaa. Erityisesti TypeScriptin käyttöönotto vei suurimman osan ajasta. Vaikka odotimme sen vievän reilusti aikaa, emme osanneet odottaa lähes kaiken ajan kuluvan siihen. Toisaalta toteutukseen jäi aikaa noin kaksi kuukautta. TypeScript toi myös esiin ongelmakohtia, joita ei ollut havaittu dynaamisessa JavaScript-koodissa. Tuloksena virheiden oikomisen jälkeen oli kuitenkin laadukkaampi koodikanta.

Ajan käytön arvioinnin ongelmat näkyvät myös siinä, että Clean Architecturen mukaista rakennetta ei ollut aikaa toteuttaa, vaikka se teoriaosiossa on melko laajana lukuna. Loppupuolella huomasimme, että joudumme tekemään kompromisseja. Alusta asti oli kuitenkin tiedossa, että täydellistä refaktorointia meillä ei ole aikaa tehdä. Tavoitteena oli kuitenkin päästä sellaiseen vaiheeseen, että projekti voidaan jättää seuraavan ohjelmistokehittäjän käsiin sellaisessa tilassa, että sitä on sujuvampi kehittää kuin ennen.

Kokonaisuutena projekti saavutti sille asetetut tavoitteet ja sen kautta opittiin paljon niin teknisestä kuin projektihallinnan sekä raportoinnin näkökulmasta. Ylipäätään olemme tyytyväisiä lopputulokseen.

LÄHTEET

Agraval, H. & Fateh, D. 2022. What is TypeScript and why should you use it? Blogi. Päivitetty 5.3.2024. Saatavissa: https://www.contentful.com/blog/what-is-typescript-and-why-should-you-use-it/ [viitattu 5.2.2025].

Altexsoft. 2023. Comparing Database Management Systems: MySQL, PostgreSQL, MSSQL Server, MongoDB, Elasticsearch, and others. Blogi. Päivitetty 16.5.2023. Saatavissa: https://www.altexsoft.com/blog/comparing-database-management-systems-mysql-postgresql-mssql-server-mongodb-elasticsearch-and-others/ [viitattu 4.3.2025].

Back-End Web Architecture s.a. Codecademy. Back-End Web Architecture. WWW-dokumentti. Saatavissa: https://www.codecademy.com/article/back-end-architecture [viitattu 21.1.2025].

Bakharev, N. 2023. Unit Testing: Definition, Examples, and Critical Best Practices. Blogi. Päivitetty 14.2.2024. Saatavissa: https://brightsec.com/blog/unit-testing/ [viitattu 17.2.2025].

Bitloops s.a. Introduction to Clean Architecture. WWW-dokumentti. Saatavissa: Clean Architecture Reference Guide: Everything You Need to Know About Clean Architecture | Bitloops Docs [viitattu 12.2.2025].

Bourque, P. & Fairley, R. 2014. Guide to the Software Engineering Body of Knowledge. E-kirja. ISBN-13 978-0-7695-5166-1. Saatavissa: https://www.researchgate.net/publication/342452008_Guide_to_the_Software_Engineering_Body_of_Knowledge_-_SWEBOK_V30 [viitattu 26.3.2025].

BrowserStack. 2025. What are different Software Testing Tools? Blogi. Päivitetty 18.3.2025. Saatavissa: https://www.browserstack.com/guide/what-are-testing-tools [viitattu 21.4.2025].

Celep, B. 2022. Introduction to Clean Architecture. Blogi. Päivitetty 18.2.2022. Saatavissa: https://celepbeyza.medium.com/introduction-to-clean-architecture-acf25ffe0310 [viitattu 5.3.2025].

Checkpoint s.a. What is Cyber Security? WWW-dokumentti. Saatavissa: https://www.checkpoint.com/cyber-hub/cyber-security/what-is-cybersecurity/ [viitattu 19.2.2025].

Cloudinary. 2024. Front-End Development: The Complete Guide. WWW-dokumentti. Päivitetty 23.10.2024. Saatavissa: https://cloudinary.com/guides/front-end-development/front-end-development-the-complete-guide [viitattu 4.3.2025].

CodeSee s.a. Code Refactoring: 6 Techniques and 5 Critical Best Practices. WWW-dokumentti. Saatavissa: https://www.codesee.io/learning-center/code-refactoring [viitattu 22.1.2025].

Dedigama, M. & Lee, R. 2022. How To Use Sequelize with Node.js and MySQL. WWW-dokumentti. Päivitetty 20.7.2022. Saatavissa: https://www.digitalocean.com/community/tutorials/how-to-use-sequelize-with-node-js-and-mysql [viitattu 22.1.2025].

Devmountain s.a. Git vs. GitHub: What's the Difference? Blogi. Saatavissa: https://devmountain.com/blog/git-vs-github-whats-the-difference/ [viitattu 10.2.2025].

Erolin, J. s.a. What Is Unreal Engine? Blogi. Saatavissa: https://www.bairesdev.com/blog/what-is-unreal-engine/ [viitattu 24.1.2025].

Fowler, M. s.a. Refactoring. WWW-dokumentti. Saatavissa: https://refactoring.com/ [viitattu 22.1.2025].

F-Secure. 2023. What is cyber security? WWW-dokumentti. Päivitetty 17.5.2023. Saatavissa: https://www.f-secure.com/en/articles/what-is-cyber-security [viitattu 19.2.2025].

Express.js Tutorial. 2024. GeeksForGeeks. WWW-dokumentti. Päivitetty 16.12.2024. Saatavissa: https://www.geeksforgeeks.org/express-js/ [viitattu 22.1.2025].

Garcia D. 2023. The top 10 programming languages (according to 2022's GitHub Octoverse report). Blogi. Päivitetty 1.5.2023. Saatavissa: https://david-garcia.medium.com/the-top-10-programming-languages-according-to-2022s-github-octoverse-report-c63ea6b7dcd2 [viitattu 16.4.2025].

GitHub s.a. About GitHub and Git. WWW-dokumentti. Saatavissa: https://docs.github.com/en/get-started/start-your-journey/about-github-and-git [viitattu 11.2.2025].

Grigutytė, M. 2023. What is bcrypt and how does it work? Blogi. Päivitetty 16.6.2023. Saatavissa: https://nordvpn.com/fi/blog/what-is-bcrypt/ [viitattu 23.1.2025].

Gunawan, E. 2024. Clean Architecture in Node.js: An Approach with TypeScript and Dependency Injection. Blogi. Päivitetty 8.7.2024. Saatavissa: https://dev.to/evangunawan/clean-architecture-in-nodejs-an-approach-with-typescript-and-dependency-injection-16o [viitattu 12.2.2025].

Hackreactor. 2022. What is JavaScript used for? WWW-dokumentti. Päivitetty 4.5.2022. Saatavissa: https://www.hackreactor.com/resources/what-is-javascript-used-for/ [viitattu 11.2.2025].

Hamilton, T. 2025. 40 BEST software testing tools list (2025). Blogi. Päivitetty 4.4.2025. Saatavissa: https://www.guru99.com/testing-tools.html [viitattu 21.4.2025].

Handlebars.js. 2021. What is Handlebars? WWW-dokumentti. Päivitetty 19.10.2021. Saatavissa: https://handlebarsjs.com/guide/#what-is-handlebars [viitattu 5.3.2025].

Heller, M. 2022. What is Visual Studio Code? Microsoft's extensible code editor. WWW-dokumentti. Päivitetty 8.7.2022. Saatavissa: https://www.infoworld.com/article/2335960/what-is-visual-studio-code-microsofts-extensible-code-editor.html [viitattu 4.2.2025].

Heller, M. 2024. What is GitHub? More than Git version control in the cloud. WWW-dokumentti. Päivitetty 6.9.2024. Saatavissa: https://www.infoworld.com/article/2266566/what-is-github-more-than-git-version-control-in-the-cloud.html [viitattu 10.2.2025].

Hgraca. 2017. Layered Architecture. Blogi. Päivitetty 3.8.2017. Saatavissa: https://herbertograca.com/2017/08/03/layered-architecture/ [viitattu 18.2.2025].

Hildebrand, M. & Selhausen, K. 2024. What are Template Engines? Blogi. Päivitetty 14.2.2024. Saatavissa: https://www.hackmanit.de/en/blog-en/178-template-injection-vulnerabilities-understand-detect-identify [viitattu 5.3.2025].

How to use Sequelize in Node.js? 2020. GeeksForGeeks. WWW-dokumentti. Päivitetty 20.5.2020. Saatavissa: https://www.geeksforgeeks.org/how-to-use-sequelize-in-node-js/ [viitattu 18.3.2025].

IBM. 2021. What is PostgreSQL? WWW-dokumentti. Päivitetty 15.10.2021. Saatavissa: https://www.ibm.com/think/topics/postgresql [viitattu 7.3.2025].

Ignacio s.a. TypeScript and Cybersecurity: Protecting Your Applications. WWW-dokumentti. Saatavissa: https://clouddevs.com/typescript/cybersecurity/ [viitattu 19.2.2025].

Indeed Editorial Team. 2024. What Are the 5 Primary Layers in Software Architecture? WWW-dokumentti. Päivitetty 16.8.2024. Saatavissa: https://www.indeed.com/career-advice/career-development/what-are-the-layers-in-software-architecture [viitattu 18.2.2025].

Jest s.a. WWW-dokumentti. Saatavissa: https://jestjs.io/ [viitattu 21.4.2025].

Kaspersky s.a. What is Cybersecurity? Types, Threats and Cyber Safety Tips. WWW-dokumentti. Saatavissa: https://www.kaspersky.com/resource-center/definitions/what-is-cyber-security [viitattu 19.2.2025].

Katalon. 2024. Unit Testing vs Integration Testing: Key differences. Blogi. Päivitetty 9.10.2024. Saatavissa: https://katalon.com/resources-center/blog/unit-testing-integration-testing [viitattu 17.2.2025].

Korolev, M. 2019. JavaScript quirks in one image from the Internet. Blogi. Päivitetty 1.3.2019. Saatavissa: https://dev.to/mkrl/javascript-quirks-in-one-image-from-the-internet-52m7 [viitattu 19.2.2025].

MariaDB s.a. About MariaDB Server. WWW-dokumentti. Saatavissa: https://mariadb.org/about/ [viitattu 5.3.2025].

Martin, R. 2012. The Clean Architecture. Blogi. Päivitetty 13.8.2012. Saatavissa: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html [viitattu 12.2.2025].

Microsoft. 2024. What is SQL Server? WWW-dokumentti. Päivitetty 9.4.2024. Saatavissa: https://learn.microsoft.com/en-us/sql/sql-server/what-is-sql-server?view=sql-server-ver16 [viitattu 5.3.2025].

Migrating from JavaScript. 2025. TypeScript. WWW-dokumentti. Päivitetty 1.4.2025. Saatavissa: https://www.typescriptlang.org/docs/handbook/migrating-from-javascript.html [viitattu 2.4.2025].

Motunrayo. 2024. What is TypeScript? Blogi. Päivitetty 10.5.2024. Saatavissa: https://hygraph.com/blog/what-is-typescript [viitattu 5.2.2025].

Noble Desktop. 2023. What is Git and Why Should You Use it? WWW-dokumentti. Päivitetty 24.7.2023. Saatavissa: https://www.noble-desktop.com/learn/git/what-is-git [viitattu 7.2.2025].

Node.js s.a. About Node.js. WWW-dokumentti. Saatavissa: https://nodejs.org/en/about [viitattu 22.1.2025].

Nodemon s.a. WWW-dokumentti. Saatavissa: https://nodemon.io/ [viitattu 22.1.2025].

NPM. 2024. Nodemon. WWW-dokumentti. Päivitetty 12.2024. Saatavissa: https://www.npmjs.com/package/nodemon [viitattu 22.1.2025].

Ojala, J. 2025. Toimeksiantaja. Haastattelu 13.3.2025. Kaakkois-Suomen Ammattikorkeakoulu Oy. [viitattu 17.3.2025].

Oracle. 2024. MySQL: Understanding What It Is and How It's Used. WWW-dokumentti. Päivitetty 29.8.2024. Saatavissa: https://www.oracle.com/mysql/what-is-mysql/ [viitattu 4.3.2025].

PostgreSQL s.a. What is PostgreSQL? WWW-dokumentti. Saatavissa: https://www.postgresql.org/about/ [viitattu 7.3.2025].

Rathbone. 2023. PostgreSQL limitations. Blogi. Päivitetty 8.2.2023. Saatavissa: https://www.beekeeperstudio.io/blog/postgresql-limitations [viitattu 7.3.2025].

Security Best Practices s.a. Node.js. WWW-dokumentti. Saatavissa: https://nodejs.org/en/learn/getting-started/security-best-practices [viitattu 19.2.2025].

Semah, B. 2022. What Exactly is Node.js? Explained for Beginners. Blogi. Päivitetty 5.12.2022. Saatavissa: https://www.freecodecamp.org/news/what-is-node-js/ [viitattu 22.1.2025].

Sequelize. 2025. Sequelize v6. WWW-dokumentti. Päivitetty 19.1.2025. Saatavissa: https://sequelize.org/docs/v6/ [viitattu 22.1.2025].

Software Architecture Patterns s.a. O'Reilly. E-kirja. Saatavissa: https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html [viitattu 25.2.2025].

Sokolov, A. 2021. What is Code Refactoring? Blogi. Päivitetty 6.12.2021. Saatavissa: https://duecode.io/blog/what-is-code-refactoring/ [viitattu 22.1.2025].

Stackademic. 2024. Building Robust Backend Systems with Clean Architecture: A Node.js Case Study. Blogi. Päivitetty 8.1.2024. Saatavissa: https://blog.stackademic.com/building-robust-backend-systems-with-clean-architecture-a-node-js-case-study-96059d1ca1e0 [viitattu 12.2.2025].

Testing with Jest. 2025. GeeksForGeeks. Blogi. Päivitetty 7.2.2025. Saatavissa: https://www.geeksforgeeks.org/testing-with-jest/ [viitattu 21.4.2025].

The Codest s.a. TEMPLATE ENGINE. WWW-dokumentti. Saatavissa: https://thecodest.co/fi/sanakirja/template-engine/ [viitattu 4.3.2025].

Twingate Team. 2024. What is BCrypt? Blogi. Päivitetty 21.8.2024. Saatavissa: https://www.twingate.com/blog/glossary/BCrypt [viitattu 23.1.2025].

Unreal Engine 2022. Unreal Engine 5.1 is now available! WWW-dokumentti. Päivitetty 15.11.2022. Saatavissa: https://www.unrealengine.com/en-US/blog/unreal-engine-5-1-is-now-available [viitattu 24.1.2025].

Version Control Systems. 2022. GeeksForGeeks. WWW-dokumentti. Päivitetty 29.6.2022. Saatavissa: https://www.geeksforgeeks.org/version-control-systems/ [viitattu 5.2.2025].

Versionhallinta ja Git s.a. OS LevelUP Koodarit. WWW-dokumentti. Saatavissa: https://oslevelupkoodarit.github.io/materials/versionhallinta-ja-git.html [viitattu 5.2.2025].

Visual Studio. 2024a. Why did we build Visual Studio Code? WWW-dokumentti. Päivitetty 5.3.2025. Saatavissa: https://code.visualstudio.com/docs/editor/whyvscode [viitattu 4.2.2025].

Visual Studio. 2024b. Working with JavaScript. WWW-dokumentti. Päivitetty 5.3.2025. Saatavissa: https://code.visualstudio.com/docs/nodejs/working-with-javascript [viitattu 4.2.2025].

What Does a Back-End Developer Do? 2024. Coursera. WWW-dokumentti. Päivitetty 3.4.2024. Saatavissa: https://www.coursera.org/articles/back-end-developer [viitattu 21.1.2025].

What Does a Front-End Developer Do? 2025. Coursera. WWW-dokumentti. Päivitetty 11.2.2025. Saatavissa: https://www.coursera.org/articles/front-end-developer [viitattu 4.3.2025].

What is CRUD? s.a. Codecademy. WWW-dokumentti. Saatavissa: https://www.codeacademy.com/article/what-is-crud [viitattu 27.1.2025].

What is Express.js? s.a. Codecademy. WWW-dokumentti. Saatavissa: https://www.codecademy.com/article/what-is-express-js [viitattu 22.1.2025].

What is JavaScript (JS)? s.a. AWS. WWW-dokumentti. Saatavissa: https://aws.amazon.com/what-is/javascript/ [viitattu 11.2.2025].

What is relational database? s.a. Google. WWW-dokumentti. Saatavissa: https://cloud.google.com/learn/what-is-a-relational-database [viitattu 4.3.2025].

What is REST? s.a. Codecademy. WWW-dokumentti. Saatavissa: https://www.codecademy.com/article/what-is-rest [viitattu 27.1.2025].

What is software testing? 2025. GeeksForGeeks. WWW-dokumentti. Päivitetty 10.2.2025. Saatavissa: https://www.geeksforgeeks.org/software-testing-basics/ [viitattu 13.2.2025].

What is unit testing? s.a. AWS. WWW-dokumentti. Saatavissa: https://aws.amazon.com/what-is/unit-testing/ [viitattu 13.2.2025].

What Is Unreal Engine? 2024. Coursera. WWW-dokumentti. Päivitetty 30.9.2024. Saatavissa: https://www.coursera.org/articles/what-is-unreal-engine [viitattu 24.1.2025].

What is Version Control? s.a. Github. WWW-dokumentti. Saatavissa: https://github.com/resources/articles/software-development/what-is-version-control [viitattu 5.2.2025].

What's the Difference Between Frontend and Backend in Application Development? s.a. AWS. WWW-dokumentti. Saatavissa: https://aws.amazon.com/compare/the-difference-between-frontend-and-backend/ [viitattu 4.3.2025].

Writing Markup with JSX s.a. React. WWW-dokumentti. Saatavissa: https://react.dev/learn/writing-markup-with-jsx [viitattu 9.4.2025].