

Infojärjestelmän taustapalvelun ja hallintapaneelin toteutus

LAB-ammattikorkeakoulu
Insinööri (AMK)
2024
Jari Kainulainen

Tiivistelmä

Tekijä(t) Jari Kainulainen	Julkaisun laji	Valmistumisaika
	Opinnäytetyö, AMK	2024
	Sivumäärä 41	
Työn nimi Infojärjestelmän taustapalvelun ja hallintapaneelin toteutus		
Tutkinto ja koulutusala Insinööri (AMK), Tieto- ja viestintätekniikka		
Toimeksiantajaorganisaatio (jos opinnäytetyöllä on toimeksiantaja) Harri Valkeinen, Tilakoordinaattori, LUT		
Tiivistelmä Työn tavoitteena oli toteuttaa LAB-ammattikorkeakoulun kampusalueelle infojärjestelmän taustapalvelu ja hallintapaneeli. Toimeksiantajana toimi LUT:n palvelukoordinaattori. Järjestelmän keskeinen tarkoitus oli tarjota kampuksen käyttäjille ajantasaista tietoa julkisen liikenteen aikatauluista. Tavoitteina olivat erityisesti julkisen liikenteen aikataulujen integrointi, järjestelmän luotettavuus ja tiedon oikeellisuus, sekä uusien asiakaspäätteiden lisäämisen mahdollisuus. Taustapalvelu toteutettiin Node.js-ympäristössä TypeScript-ohjelmointikielellä hyödyntäen Express.js-ohjelmistokehystä. Palvelun tarjoamat rajapinnat toteutettiin RESTful-periaatteita mukaillen HTTP-pohjaisina. Palveluun integroitiin aikataulutiedot DigiTrafficin ja DigiTransitin rajapinnoista sekä säätiedot OpenWeatherMapin rajapinnasta. Hallintapaneeli toteutettiin Vue.js-ohjelmistokehyksellä, ja sen kautta voidaan hallinnoida järjestelmän sijainteja, asiakaspäätteitä ja aikataulunäkymiä. Tietojen tallennukseen käytettiin SQLite-tietokantaa TypeORM-kirjaston avulla. Projektin tuloksena syntyi toimiva infojärjestelmä, joka palvelee päivittäin kampuksen käyttäjiä. Järjestelmän toiminta on osoittautunut luotettavaksi ja vakaaksi tuotantokäytössä. Tiedonhallintaratkaisut kuten kaksitasoinen välimuisti ja varmuuskopioaikataulut ovat mahdollistaneet järjestelmän toiminnan myös ulkoisten rajapintakatkosten aikana. Järjestelmän skaalautuvuus on todennettu käytännössä uusien asiakaspäätteiden lisäämisellä, ja arkkitehtuuri tukee laajentumista myös muihin kaupunkeihin.		
Asiasanat taustapalvelu, rajapintaintegraatio, hallintapaneeli, Node.js, TypeScript, Vue.js, infojärjestelmä		

Abstract

Author(s)	Type of Publication	Published
Jari Kainulainen	Thesis, UAS	2024
	Number of Pages	
	41	
Title of Publication		
Implementation of Information System's Backend Service and Admin Panel		
Degree, Field of Study		
Bachelor of Engineering, Information and Communications Technology		
Organisation of the client (if the thesis work is commissioned by another party)		
Harri Valkeinen, Facility Coordinator, LUT		
Abstract		
The thesis' objective was to implement a backend service and admin panel for an information system at LAB University of Applied Sciences campus area. The project was commissioned by LUT's facility coordinator. The primary purpose of the system was to provide campus users with real-time public transportation schedule information. The main goals were the integration of public transportation schedules, system reliability, data accuracy, and the ability to add new client terminals. The backend service was implemented in Node.js environment using TypeScript programming language and the Express.js framework. The service APIs were implemented following RESTful principles and HTTP-based protocols. The service integrates schedule data from DigiTraffic and DigiTransit APIs, and weather data from the OpenWeatherMap API. The admin panel was implemented using the Vue.js framework, enabling the management of system locations, client terminals, and schedule views. Data storage was implemented using SQLite database with the TypeORM library. The project resulted in a functional information system that serves campus users daily. The information system has proven to be reliable and stable in production use. A two-level caching solution and backup schedules have enabled system operation even during external API outages. The system's scalability has been verified in practice by adding new client terminals, and the architecture supports expansion to other cities as well.		
Keywords		
backend service, API integration, admin panel, Node.js, TypeScript, Vue.js, info system		

Sisällys

1	Johdanto.....	1
2	Ohjelmiston arkkitehtuuri	2
2.1	REST	2
2.1.1	REST:n kuusi periaatetta.....	2
2.1.2	HTTP-pohjainen RESTful-rajapinta.....	3
2.2	MVVM	4
3	Teknologiat ja ympäristö	6
3.1	JavaScript	6
3.1.1	Prototyyppinen perintä.....	6
3.1.2	Monisäikeisyys ja asynkronisuus	7
3.2	TypeScript	9
3.3	Node.js-ajoympäristö.....	10
3.4	Express.js-ohjelmistokehys	12
3.4.1	Middleware-funktiot	13
3.4.2	Reititys	13
3.5	Tunnistautuminen Passport.js-kirjastolla.....	14
3.6	SQLite-tietokanta.....	15
3.7	TypeORM-kirjasto ja olio-relaatio-mallinnus	15
3.8	Vue.js-ohjelmistokehys.....	17
3.9	Linux-palvelin	19
4	Ohjelmiston toteutus	21
4.1	Infojärjestelmän rakenne pääpiirteittäin	21
4.2	Taustapalvelun tarkempi rakennekuvaus.....	22
4.2.1	Tietorakenteet	22
4.2.2	Tiedonhallintakomponentit.....	24
4.2.3	Integraatiokomponentit	27
4.3	Rajapinnat	27
4.1	Toimintaperiaatteet.....	30
4.1.1	Aikataulujen hakeminen ja tallentaminen.....	30
4.1.2	Asiakaspäätteiden kanssa kommunikointi.....	32
4.2	Järjestelmänhallinta.....	33
4.3	Ohjelmiston käyttöönotto	36
5	Yhteenveto ja pohdinta	38
	Lähteet	41

Sanasto:

API	Application Programming Interface, rajapinta ohjelmistojen ja komponenttien väliseen kommunikointiin.
DOM	Document Object Model, ohjelmointirajapinta ja puumalli HTML-dokumentin käsittelyyn.
FIFO	First In First Out, Ensimmäisenä sisään, ensimmäisenä ulos -periaate, käytetään usein tietorakenteiden yhteydessä
GraphQL	Kyselykieli rajapinnoille
HTTP	Hypertext Transfer Protocol, protokolla selainten ja palvelinten väliseen tiedonsiirtoon
IP	Internet Protocol, Internetin tiedonsiirron perusprotokolla
JSON	JavaScript Object Notation, tiedonvälitykseen käytettävä tiedostomuoto ja tekstin rakennemäärittely
MAC	Verkkosovittimen fyysinen osoite
MIME	Multipurpose Internet Mail Extensions, määrittelee tiedostotyytit verkkoympäristössä
MVVM	Model-View-ViewModel, ohjelmistoarkkitehtuuri, joka erottaa datan, logiikan ja näkymän.
ORM	Object-relational mapping, tekniikka oliopohjaisen koodin ja relaatiotietokantojen yhdistämiseen
REST	Representational State Transfer, arkkitehtuurityyli verkkosovelluksille
SaaS	Pilvipalvelumalli ohjelmistojen jakeluun
SPA	Verkkosovellustyyppi, joka toimii yhdellä HTML-sivulla ja sisältää usein vaihtuu dynaamisesti.
SQL	Relaatiotietokantojen kyselykieli
URI	Resurssin yksilöivä merkkijonomuotoinen tunniste

1 Johdanto

Infonäytöt ovat muodostuneet erottamattomaksi osaksi nykyaikaisia oppimisympäristöjä ja kampusalueita. LAB-ammattikorkeakoulun kaltaisissa korkeakouluissa ne palvelevat päivittäin tuhansia opiskelijoita, henkilökuntaa ja vierailijoita, mahdollistaen tehokkaamman ajankäytön ja sujuvamman liikkumisen kampuksen tiloissa. Näytöt voivat myös sisältää esimerkiksi julkisen liikenteen aikataulutietoja, jotka ovat erityisen merkityksellisiä kampusympäristössä, jossa suuri osa opiskelijoista ja henkilökunnasta hyödyntää joukkoliikennettä päivittäiseen liikkumiseen. Reaaliaikaiset aikataulutiedot voivat auttaa kampusyhteisöä suunnittelemaan liikkumistaan tehokkaasti, mahdollisesti vähentäen turhaa odottelua.

Tämän työn toimeksiantaja on LAB-ammattikorkeakoulun tilapalvelut. Toimeksiantajan kanssa käytyjen keskustelujen pohjalta tunnistettiin selkeä tarve kampuksen infojärjestelmän kehittämiseksi erityisesti julkisen liikenteen aikataulujen osalta. Tämän tarpeen pohjalta käynnistettiin kehitysprojekti, jonka tavoitteena on toteuttaa kampukselle moderni infojärjestelmä asiakaspääteineen ja taustapalveluineen.

Opinnäytetyön tavoitteena on toteuttaa infojärjestelmän taustapalvelu sekä hallintapaneeli. Tavoitteena on integroida järjestelmään julkisen liikenteen aikataulutiedot saatavilla olevista rajapinnoista. Infojärjestelmän luotettavuus ja tiedon oikeellisuus ovat myös merkittävässä roolissa, kun kyse on aikataulutiedoista. Järjestelmän tulisi olla myös skaalautuva, eli tulevaisuudessa tulisi olla mahdollista lisätä kampuksille uusia asiakaspäätteitä, joilla näytetään aikataulutietoja eri sijainnin perusteella.

Työ kattaa taustaohjelmiston suunnittelun ja kehitystyön lisäksi myös sen toimintaan saattamisen. Järjestelmäkokonaisuuteen kuuluvan asiakaspäätteen ohjelmiston ja käyttöliittymän toteutuksesta vastaa saman projektin toinen kehittäjä, ja täten se rajautuu tämän opinnäytetyön ulkopuolelle. Projektin kolmas kehittäjä vastaa asiakaspäätteiden toimintaympäristön asennuksista, sekä rajapintojen integraation teknisestä toteutuksesta ja datan prosessoinnista, kun taas tämän opinnäytetyön painopiste on taustapalvelun arkkitehtuuriratkaisuissa ja yleisessä toimintalogiikassa.

2 Ohjelmiston arkkitehtuuri

2.1 REST

Roy Fielding määritteli Representational State Transfer -arkkitehtuurityylin (REST) väitöskirjassaan vuonna 2000. REST-arkkitehtuuri pohjautui 1990-luvun olemassa olevaan web-arkkitehtuuriin, ja sitä voisi kuvailla sen abstrahoinniksi ja formalisoinniksi. REST toimi mallina, jonka avulla voitiin tunnistaa olemassa olevan arkkitehtuurin ongelmia ja arvioida siihen ehdotettuja muutoksia. (Fielding & Taylor 2002, 116, 137-138.)

Fieldingin ja Taylorin (2002, 124-125) mukaan REST-arkkitehtuurin keskeinen ajatus on, että mikä tahansa nimettävissä oleva informaatio voi toimia resurssina. Resurssi voi olla esimerkiksi dokumentti, kuva, hetkellinen palvelu tai kokoelma muita resursseja. Lisäksi REST-arkkitehtuuri perustuu kuuteen keskeiseen rajoitteeseen, joita usein kutsutaan myös periaatteiksi (Fielding 2000, 76).

2.1.1 REST:n kuusi periaatetta

Ensimmäinen REST:n kuudesta periaatteesta on asiakas-palvelin-malli (client-server), jossa käyttöliittymä on erotettu tiedon tallennuksesta. Taustalla olevana ajatuksena on ollut huolien erottelu -periaate, eli käyttöliittymän ja datan erottaminen, sekä tämänkaltaisen erottelun tuomat hyödyt. Komponentit voivat kehittyä ja skaalautua erillään toisistaan. (Fielding 2000, 78.)

Toinen periaate on tilattomuus (stateless), joka tarkoittaa, että jokaisen pyynnön täytyy sisältää kaikki sen käsittelyyn tarvittava tieto. Palvelin ei siis säilytä tietoa aiemmista pyynnöistä, joten tilan seuranta ja hallinta jäävät asiakasohjelman vastuulle. (Fielding 2000, 78–79.)

Kolmas periaate on välimuistin (cache) käyttö, joka mahdollistaa vastausten tallentamisen myöhempää käyttöä varten, tarkoituksena edistää verkon tehokkuutta. Vastaukset voidaan joko erikseen tai oletuksena määritellä välimuistitettaviksi. (Fielding 2000, 79–81.)

Neljäs ja hyvin keskeinen periaate on yhtenäinen rajapinta (uniform interface) komponenttien välillä. Kun kaikki komponentit käyttävät samanlaista rajapintaa, järjestelmästä tulee yksinkertaisempi ja sen toiminta on helpommin ymmärrettävää. Tällä ratkaisulla on kuitenkin kääntöpuolensa: kun kaikki tieto kulkee saman standardoidun rajapinnan läpi, järjestelmä ei ole yhtä tehokas kuin se voisi olla räätälöidyillä ratkaisuilla. Yhtenäisen rajapinnan toteuttamisessa REST-arkkitehtuurissa keskitytään neljään ominaisuuteen:

- resurssien tunnistaminen

- resurssien käsittely niiden esitysmuotojen kautta
- viestien sisällön selkeys ja
- hypermedia sovelluksen tilan hallinnassa. (Fielding 2000, 81–82.)

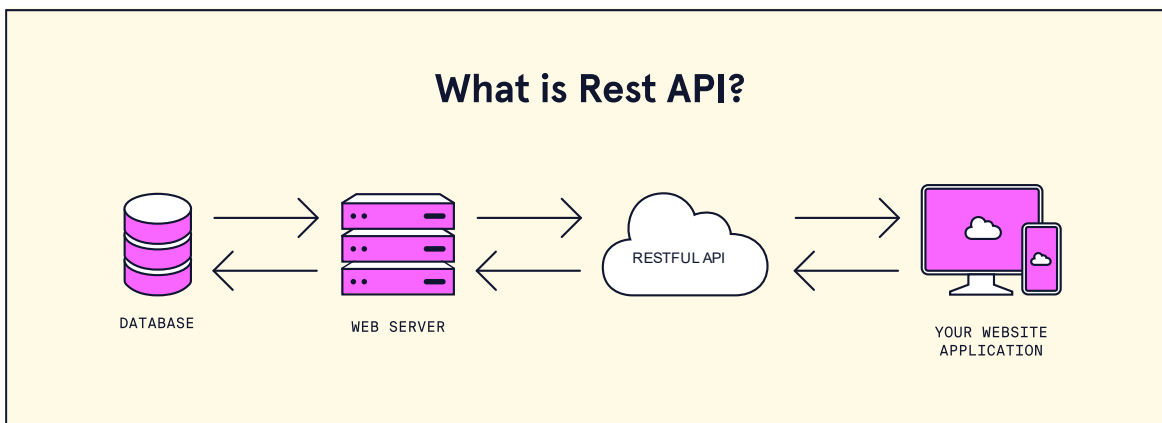
Viides periaate on kerroksittainen järjestelmä (layered system), jonka tarkoitus on pilkkoa järjestelmän arkkitehtuuri hierarkkisiin kerroksiin. Komponenttien tulisi ”nähdä” vain välittömässä vaikutuksessa olevaan kerrokseen, eikä toisiin kerroksiin. Kerroksellisuus pyrkii parantamaan järjestelmän skaalautuvuutta mahdollistamalla kuormantasauksen eri verkkojen ja prosessorien välillä. (Fielding 2000, 82–84.)

Kuudes ja viimeinen, valinnainen periaate, on koodin suoritus tarvittaessa (code on demand). Sen pyrkimys on mahdollistaa asiakasohjelman toiminnallisuuden laajentamisen laa- taamalla ja suorittamalla koodia sovelluskomponenttien tai skriptien muodossa. Tämä yk- sinkertaistaa asiakaspuolen toteutusta vähentämällä esiohjelmointien ominaisuuksien määrää ja parantaa järjestelmän laajennettavuutta. Periaatteen valinnaisuus mahdollistaa sen hyödyntämisen organisaation sisällä samalla kun ulkopuolelta tuleva koodi voidaan es- tää tietoturvasyistä. Haittapuolena on järjestelmän näkyvyyden heikkeneminen. (Fielding 2000, 84–85.)

2.1.2 HTTP-pohjainen RESTful-rajapinta

RESTful termiä käytetään ohjelmistosta, joka seuraa REST-periaatteita. Nykykielessä sitä käytetään usein kuvaamaan HTTP-pohjaisia rajapintoja, jotka noudattavat ainakin osaa REST:n keskeisistä periaatteista. Fielding (2008) kuitenkin kritisoi blogissaan sitä, miten monet REST:ksi kutsutut rajapinnat määrittelevät ennalta vain kiinteitä osoiterakenteita ja toimintoja sen sijaan, että rajapinnan käyttäjä löytäisi tarvittavat toiminnot ja osoitteet dy- naamisesti palvelimen tarjoamista vastauksista osana hypermediasisältöä.

Erilaiset HTTP-pohjaiset RESTful-rajapinnat ovat kuitenkin nykyaikana varsin suosittu ark- kitehtuurimalli web-kehityksessä. Ohjelmistokehittäjille suunnatussa kyselyssä (RapidAPI 2022), lähes 70% kehittäjistä kertoi käyttävänsä REST-arkkitehtuuria tuotannossa. Tyypil- listä RESTful-rajapintaa hyödyntävää web-ohjelmistoa kuvaillaan kuviossa 1.



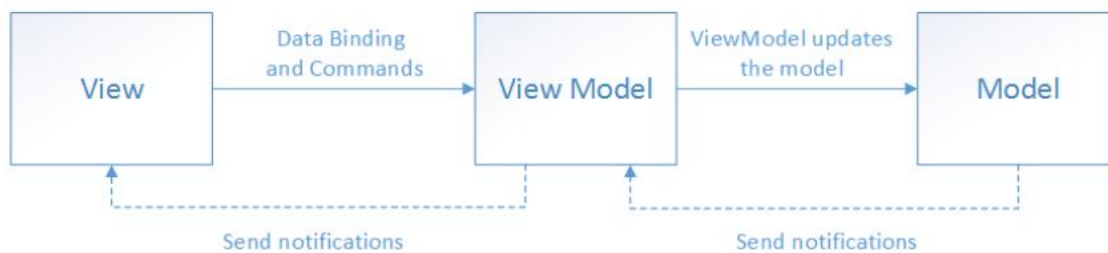
Kuvio 1. RESTful-rajapinta web-ohjelmistossa (Codecademy 2024)

Käytännön toteutuksissa modernin HTTP-pohjaisen RESTful-rajapinnan toteuttamisessa olennaista on selkeiden ja johdonmukaisten polkujen suunnittelu resursseihin. Polkujen tulee olla hierarkkisia ja kuvaavia, ja niissä käytetään tyypillisesti resurssien monikkomuotoja. Esimerkiksi polku `/customers/123/orders/12` kertoo yksiselitteisesti, että käsitellään asiakkaan 123 tilausta 12. Rajapinnan pyynnöissä on määriteltävän HTTP-verbin ja polun lisäksi otsikkotiedot, joissa määritellään hyväksyttävät sisältötyypit MIME-formaatissa, kuten `application/json` tai `text/html`. (Codecademy 2024.)

Vastausten käsittelyssä palvelimen tulee sisällyttää `content-type`-määritys otsikkotietoihin ja palauttaa toimenpiteeseen sopiva statuskoodi. Esimerkiksi onnistunut GET-pyyntö palauttaa koodin 200, uuden resurssin luonti POST-pyynnöllä koodin 201 ja onnistunut poisto DELETE-pyynnöllä koodin 204. Palvelimen palauttaman sisältötyypin tulee vastata asiakkaan `Accept`-kentässä määrittelemiä tyyppejä, minkä tarkoitus on varmistaa yhteensopivan tiedonsiirron ja -käsittelyn osapuolten välillä. (Codecademy 2024.)

2.2 MVVM

MVVM-suunnittelumalli (Model-View-ViewModel) pyrkii erottamaan sovelluksen liiketoimintalogiikan ja esityslogiikan sen käyttöliittymästä (Stonis 2022, 9). Tämän erottelun tarkoitus on tehdä sovelluksesta helpommin testattava, ylläpidettävä ja jatkokehittettävä. MVVM-malli pyrkii tukemaan koodin uudelleenkäytettävyyttä ja kehittäjien ja käyttöliittymäsuunnittelijoiden yhteistyötä sovelluksen eri osien samanaikaisessa kehittämisessä. (Stonis 2024.) Mallin kolme pääkomponenttia ja niiden väliset suhteet on esitetty kuviossa 2.



Kuvio 2. Yleiskuva MVVM-suunnittelumallista (Stonis 2022, 9)

MVVM-mallissa näkymä (View) määrittelee käyttöliittymän rakenteen, asettelun ja ulko-asun. Näkymän koodissa tulisi olla mahdollisimman vähän liiketoimintalogiikkaa. Näkymä voi kuitenkin sisältää käyttöliittymälogiikkaa esimerkiksi sellaisten visuaalisten toimintojen toteuttamiseksi, joita on vaikea toteuttaa muulla tavoin. (Stonis 2024.)

Näkymämalli (ViewModel) toimii sovittimena näkymän ja mallin välillä synkronoimalla niiden tilat (Stonis 2024). Se toteuttaa ominaisuudet ja komennot, joihin näkymä voi sitoutua, ja ilmoittaa näkymälle tilan muutoksista muutosilmoitustapahtumien kautta. Näkymämalli määrittelee käyttöliittymän toiminnallisuuden, mutta näkymä määrittää miten toiminnallisuus esitetään. (Stonis 2022, 11.)

Malli (Model) koostuu ei-visuaalisista luokista, jotka kapseloivat sovelluksen datan. Se edustaa sovelluksen toimialueaailmaa, joka yleensä sisältää datamallin sekä liiketoiminta- ja validointilogiikan. Mallin tulisi olla erillinen näkymämallista, ja muutokset välittyvät viestirajapinnan kautta näkymämallille. Näkymämalli koordinoi näkymän vuorovaikutukset malliluokkien kanssa, ja voi suoraan muokata mallin tilaa tai kutsua mallin metodeja toteuttaessaan käyttöliittymästä tulevia komentoja. (Stonis 2024.)

MVVM eroaa perinteisestä MVC-mallista (Model-View-Controller) erityisesti näkymän ja logiikan välisessä kommunikaatiossa. MVC-mallissa näkymä kommunikoi suoraan käsittelijän kanssa käyttäjän toimintojen ja päivitysten hoitamiseksi, minkä jälkeen käsittelijä huolehtii vuorovaikutuksesta mallin kanssa. MVVM-mallissa näkymä puolestaan kommunikoi näkymämallin kanssa datasidosten avulla sitoutumalla sen tarjoamiin ominaisuuksiin ja komentoihin. MVVM:n datasidokset synkronoivat näkymän ja näkymämallin automaattisesti, kun taas MVC:ssä näkymä tarvitsee käsittelijän toimintaa datan hakemiseen ja käyttöliittymän päivittämiseen. (Krstanović 2023.)

3 Teknologiat ja ympäristö

3.1 JavaScript

JavaScript on ohjelmointikieli, jota tulkitaan ja suoritetaan ajonaikaisesti. Selaimet ovat perinteisesti olleet JavaScriptin pääasiallisia suoritussympäristöjä, mutta nykyisin sitä hyödynnetään myös selaimen ulkopuolella, kuten esimerkiksi palvelimissa Node.js-ajoympäristössä. JavaScript on yksisäikeinen, prototyyppipohjainen, moniparadigmainen ja dynaaminen ohjelmointikieli, joka tukee monipuolisesti erilaisia ohjelmointityylejä kuten imperatiivista, deklarativista sekä olio-ohjelmointityyliä. (MDN 2024a.)

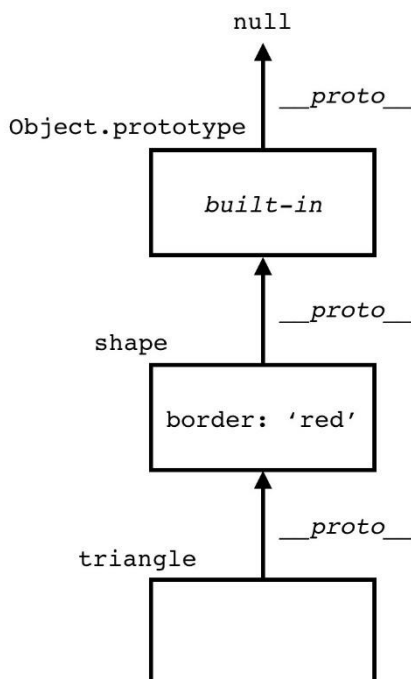
JavaScriptin dynaamisuus näkyy monella tavalla kielen toiminnassa. Ohjelmoija voi luoda uusia olioita suorituksen aikana, käyttää vaihtelevan kokoisia parametrilistoja ja käsitellä funktioita muuttujina. Kieli tarjoaa myös monipuoliset työkalut ohjelman rakenteen tutkimiseen ja muokkaamiseen ajon aikana, kuten olioiden sisällön tarkastelun ja funktioiden lähdekoodin käsittelyn. (MDN 2024a.)

3.1.1 Prototyyppinen perintä

JavaScript muistuttaa syntaksiltaan perinteisiä ohjelmointikieliä kuten C:tä ja Javaa ja siinä hyödynnetään näistä kielistä tuttuja ohjelmointirakenteita, kuten silmukoita ja ehtolauseita. Tapa, jolla JavaScript toteuttaa oliopohjaisuutta poikkeaa kuitenkin luokkapohjaisista olio-ohjelmointitoteutuksista, sillä JavaScript käyttää prototyyppipohjaista mallia perintään. (MDN 2024b.)

Prototyyppisessä oliomallissa olioiden perintä tapahtuu suoraan olioiden välillä, kun olemassa oleva olio toimii toisen olion mallikappaleena eli prototyyppinä. Manuaalisesti tehtynä viite prototyyppiin määritellään olion `__proto__`-ominaisuuteen, mutta nykyisin tähän tehtävään on suositeltavaa käyttää valmiita metodeja kuten `Object.create`, `Object.getPrototypeOf` ja `Object.setPrototypeOf`. Näin voidaan toteuttaa ominaisuuksien ja metodien periytymisen ilman erillisiä luokkamäärittelyjä. (Kantor 2021.)

Kun oliolta pyydetään ominaisuutta tai metodia, jota sillä ei itsellään ole, JavaScript-moottori tarkistaa automaattisesti oliolle määritetyn prototyypin ominaisuudet. Jos pyydetään esimerkiksi ominaisuuden `border`-arvoa `triangle`-oliolta, tarkastelu etenee prototyypin prototyyppiin muodostaen ketjun, joka jatkuu, kunnes ominaisuus löytyy tai prototyyppiketju päättyy ja palautetaan määrittelemätön arvo `undefined` (Kuvio 3). (Haverbeke 2024, 96.)



Kuvio 3. Prototyypiketju (Buoncristiani 2016)

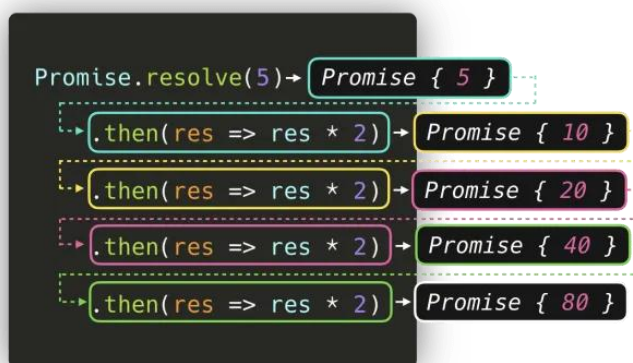
3.1.2 Monisäikeisyys ja asynkronisuus

Vaikka kielen moottori suorittaa tehtäviä yksisäikeisesti, sen ajoympäristöt kuten esimerkiksi selaimet tarjoavat tyypillisesti rajapintoja, jotka mahdollistavat tehtävien käsittelyn myös monisäikeisesti. JavaScript hyödyntää asynkronista ohjelmointimallia, jossa pitkäkestoisien tehtävien suorittaminen, kuten esimerkiksi verkkopyynnöt tai tiedoston käsittely, siirretään ympäristön rajapintojen suoritettaviksi. Lopputulokseen voidaan palata, kun rajapinnoille ulkoistettu tehtävä on suoritettu ja tulos on valmis. (Haverbeke 2024, 173.)

Asynkronisten operaatioiden hallintaan JavaScript tarjoaa erilaisia mekanismeja, joista keskeisimpiä ovat takaisinkutsufunktiot ja lupaukset. Takaisinkutsufunktiot ovat olleet perinteisempi tapa käsitellä asynkronisia operaatioita. Funktiolle määritellään parametriksi toinen funktio, jonka ympäristö kutsuu suoritettavaksi vasta sitten kun alkuperäinen aikaa vievä operaatio on suoritettu. (Haverbeke 2024, 175.) Takaisinkutsufunktiot voivat kuitenkin johtaa monimutkaiseen ja vaikeasti ylläpidettävään koodiin, erityisesti kun useita takaisinkutsufunktioita ketjutetaan peräkkäin (MDN 2024c).

Promise on JavaScript-kielen standardiluokka. Sen sijaan, että käytettäisiin takaisinkutsufunktioita, asynkroniset operaatiot on abstrahoitu olioihin, jotka edustavat tulevaisuudessa saatavaa arvoa. Tämä lähestymistapa pyrkii tekemään ohjelman rakenteesta samankaltaisemman synkronisen ohjelmoinnin kanssa. (Haverbeke 2024, 176–177.)

Promise-luokan keskeinen ominaisuus on then-metodi, jonka avulla voidaan rekisteröidä funktioita suoritettavaksi, kun lupaus ratkaistaan, eli sen arvo tulee saataville. Then-metodi palauttaa uuden Promise-olion, joka mahdollistaa useampien then-kutsujen ketjuttamisen peräkkäin. Tämä ketjutuksen avulla voidaan rakentaa toimintojen sarjoja, joissa kukin vaihe voi joko muokata dataa synkronisesti tai suorittaa uuden asynkronisen operaation (Kuvio 4). (Haverbeke 2024, 177.)



Kuvio 4. Ketjutettuja lupauksia paluuarvoineen (Hallie 2020)

JavaScript-kielen määrittelevän ECMAScript 2016 -standardin myötä kieleen tuotiin async- ja await-syntaksit (Kuva 1), joiden avulla Promise-olioiden käsittely voidaan kirjoittaa näyttämään vielä synkronisemmalta. Async-avainsanalla merkitty funktio palauttaa implisiittisesti Promise-olion, ja await-avainsanan avulla voidaan odottaa niiden ratkaisemista. Tämä lähestymistapa pyrkii poistamaan tarpeen then-ketjutuksille, jonka myötä erityisesti silmukoiden ja virheenkäsittelyn toteuttaminen voi olla suoraviivaisempaa. (Haverbeke 2024, 182–183.)

```

async function fetchData() {
  try {
    const response = await fetch('https://api.example.com/user/1');
    const userData = await response.json();
    console.log(userData);
  } catch (error) {
    console.error('Error:', error);
  }
}
  
```

Kuva 1. Asynkronien funktio async-syntaksilla

Synkronisten ja asynkronisten operaatioiden suoritusjärjestys voi joskus olla hankala hahmottaa. Toimintamalli, jossa asynkroniset operaatiot suoritetaan jonon kautta ja niihin

palataan tulosten valmistuttua, saa aikaan sen, että synkroninen koodi suoritetaan ensin, vaikka se olisi kirjoitettu esimerkiksi takaisinkutsufunktion jälkeen. Asynkronisten operaatioiden suoritusrjestykseen perehdytään tarkemmin Node.js-suoritusympäristön käsitelyssä luvun 3.3 yhteydessä.

3.2 TypeScript

TypeScript on Anders Hejlsbergin suunnittelema ja nykyisin Microsoftin kehittämä ja avoimeen lähdekoodiin perustuva ohjelmointikieli, joka rakentuu JavaScriptin päälle (Poggiali 2024). TypeScript lisää JavaScriptiin staattisen tyyppijärjestelmän. Tämän ansiosta kehittäjät voivat määritellä muuttujien, funktioiden ja olioiden tyypit eksplisiittisesti, mikä voi auttaa tunnistamaan mahdollisia virheitä jo kehitysvaiheessa. TypeScript-kääntäjä voi päätellä tyyppejä automaattisesti koodin perusteella, joten ylimääräisiä tyyppimerkintöjä ei aina tarvita. Rakenteellisen tyyppijärjestelmän vuoksi olioiden yhteensopivuus määräytyy niiden rakenteiden perusteella, eli ankkatyyppityksen-periaatteella. (Microsoft 2024a.)

JavaScriptistä poiketen TypeScriptillä on erillinen käyttäjän hallinnassa oleva kääntäjä (transpiler). Kääntäjällä on kaksi päätehtävää: tyyppitarkistus ja JavaScriptiksi kääntäminen. Nämä prosessit ovat toisistaan riippumattomia, ja TypeScript voi tuottaa suoritettavaa JavaScript-koodia myös tyyppitarkistusvirheiden ilmetessä. Tyypit poistetaan käännosvaiheessa, joten niillä ei tulisi olla vaikutusta suorituskyykyyn ajonaikana. (Poggiali 2024.)

TypeScript tuo mukanaan JavaScriptiin verrattuna uusia kielellisiä ominaisuuksia, joista esimerkkeinä nimiavaruudet, enumeraattorit ja geneeriset tyypit. Konsepteiltaan nämä ominaisuudet ovat osittain samankaltaisia kuin esimerkiksi Javassa ja C#:ssa. (Microsoft 2024a.)

Oliotyyppejä voidaan määritellä kolmella tavalla: anonyymeinä tyypeinä funktioiden parametreissa, interface-määrittelynä tai type alias -määrittelynä. Oliotyypeissä voidaan määritellä ominaisuuksien pakollisuus, kirjoitusoikeudet sekä readonly-määre, joka estää ominaisuuden uudelleenkirjoittamisen käännoksen aikana. (Microsoft 2024b.)

Geneerisyys on ohjelmointikielten ominaisuus, joka mahdollistaa uudelleenkäytettävien ja joustavien komponenttien luomisen. TypeScriptissä geneerisillä tyypeillä voidaan toteuttaa funktioita ja luokkia, joiden tietotyyppi määritellään vasta käytön yhteydessä. Tarkoituksena on pyrkiä parantamaan koodin uudelleenkäytettävyyttä ja vähentää tarvetta kirjoittaa samankaltaista koodia eri tietotyypeille, kuitenkin säilyttäen tyyppiturvallisuuden. Geneeriset tyypit määritellään kulmasulkeilla ja tyyppimuuttujilla, jotka toimivat paikanpitäjinä varsinaisille tyypeille. (Microsoft 2024c.)

Geneeristen tyyppien käyttöä voidaan rajoittaa ja ohjata erilaisten rajoitteiden avulla. Rajoitteita voidaan asettaa esimerkiksi extends-avainsanalla, jolloin geneerinen tyyppi voidaan rajata toteuttamaan tietty rajapinta tai periytymään tietystä luokasta (Kuva 2). Tämä mahdollistaa geneeristen komponenttien turvallisen käytön varmistamalla, että käytetyt tyypit täyttävät tarvittavat vaatimukset. Rajoitteiden avulla voidaan myös hyödyntää tyyppiparametrien ominaisuuksia turvallisesti komponentin toteutuksessa. (Microsoft 2024c.)

```
interface Lengthwise {
  length: number;
}

function loggingIdentity<Type extends Lengthwise>(arg: Type): Type {
  console.log(arg.length); // Now we know it has a .length property, so no more error
  return arg;
}
```

Kuva 2. Geneerinen funktio tyypin rajauksella (Microsoft 2024c)

3.3 Node.js-ajoympäristö

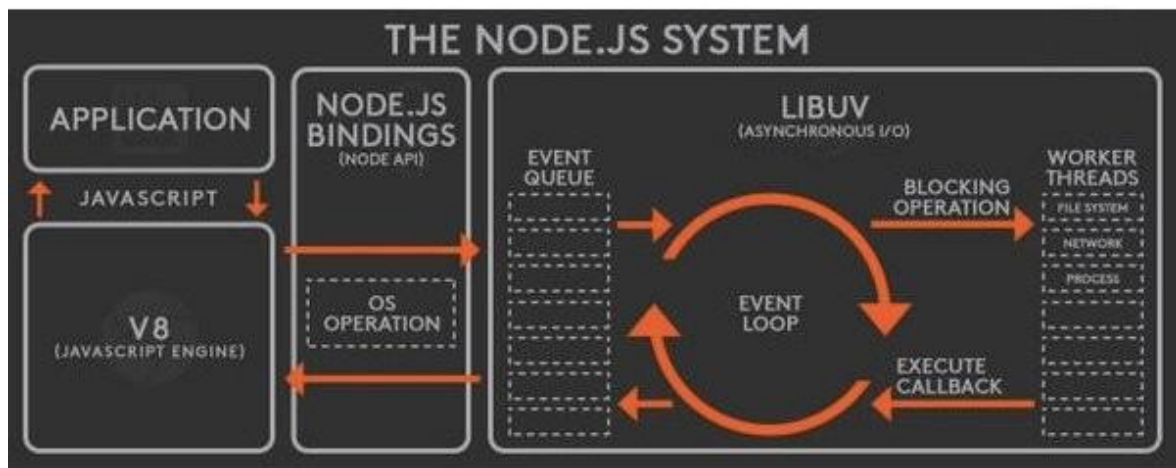
Node.js on avoimen lähdekoodin monialustainen JavaScriptin suoritussympäristö. JavaScriptin suorittamiseen se hyödyntää Googlen Chrome-selaimen tehokkaana pidettyä V8 JavaScript-moottoria. Ajoympäristöllä on kuitenkin omat selaimesta poikkeavat rajapinnat muun muassa tiedostojärjestelmän käsittelyyn ja verkkotoimintoihin. (OpenJS Foundation 2024a.)

Node.js-ympäristön mukana tulee npm, eli Node Package Manager, joka on sen standardoitu laaja pakettinhallintajärjestelmä. Syyskuussa 2022 npm:n rekisterissä oli yli 2,1 miljoonaa pakettia. Järjestelmällä voi asentaa, päivittää ja hallinnoida projektin riippuvuuksia eli esirakennettuja koodikomponentteja. (OpenJS Foundation 2024b.)

Node.js pystyy suorittamaan tsc-kääntäjällä JavaScriptiksi käännettyä TypeScript-koodia sellaisenaan. Node.js:n rinnalla voidaan käyttää myös suoritussympäristöjä (runner), jotka mahdollistavat TypeScript-koodin suorittamisen ilman käyttäjän tekemää erillistä kääntämisvaihetta. Kaksi yleistä suoritussympäristöä ovat ts-node ja tsx. Nämä työkalut mahdollistavat TypeScript-tiedostojen suoran suorittamisen, mutta kumpikaan ei tee tyyppitarkistuksia. Tästä syystä tulisi suorittaa tyyppitarkistus tsc-kääntäjällä ennen koodin ajamista ts-nodella tai tsx:llä. Molemmat työkalut ovat asennettavissa npm-pakettinhallintajärjestelmän kautta kehitysriippuvuuksina. (OpenJS Foundation 2024c.)

Node.js:n keskeytymätön asynkroninen toimintamalli perustuu tapahtumaohjattuun arkkitehtuuriin, jossa sovellus suoritetaan yhdessä säikeessä hyödyntäen lisäksi asynkronista siirräntäoperaatioiden rajapintaa. Tämän ansiosta sovellus ei pysähdy odottamaan aikaa

vievien operaatioiden, kuten tietokantakyselyiden tai tiedostojärjestelmän käsittelyn valmistumista, vaan jatkaa muiden tehtävien suorittamista ja palaa keskeneräisiin operaatioihin vasta niiden valmistuttua (Kuvio 4). Tällainen toimintamalli mahdollistaa järjestelmän resurssien tehokkaan käytön ja tuhansien samanaikaisten yhteyksien ja pyyntöjen käsittelyn ilman monien säikeiden manuaalista hallintaa. (OpenJS Foundation 2024a.)

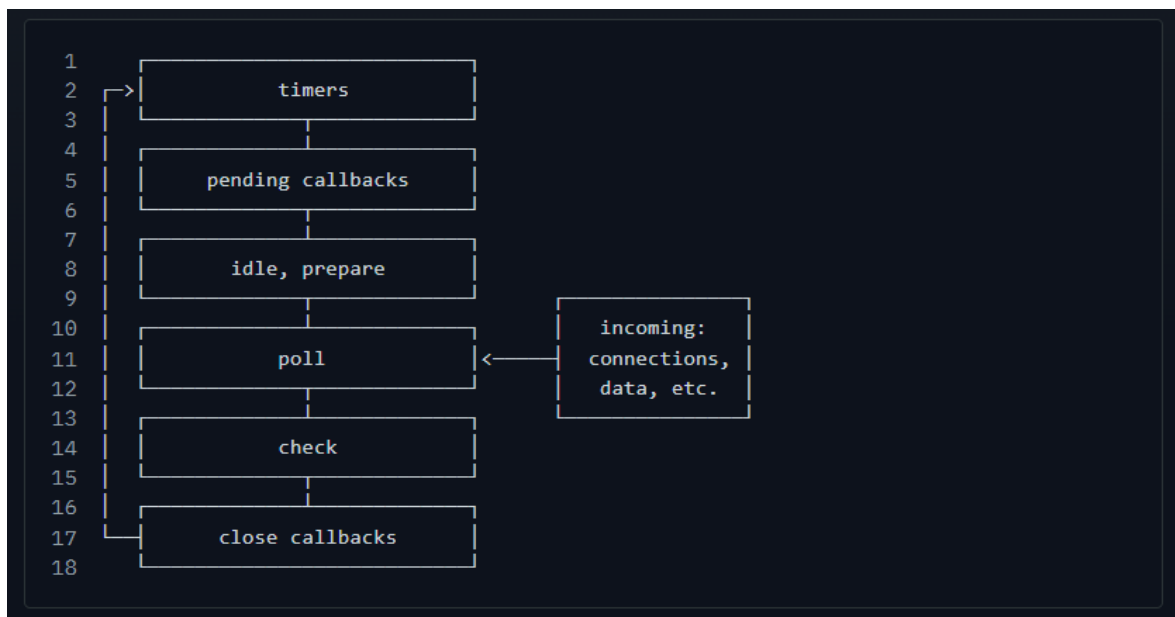


Kuvio 4. Node.js tapahtumaohjattu toimintamalli (Willison 2017)

Asynkroniset operaatiot suoritetaan rinnakkaisesti libuv-kirjaston avulla. Oletusarvoisesti libuv varaa neljä käyttöjärjestelmätason työsäiettä. Näitä säikeitä käytetään vain silloin, kun suoritetaan keskeyttäviä asynkronisia operaatioita, eli operaatioita, jotka estäisivät pääsäikeen suorituksen jatkumisen. Nykyiseltään Node.js hyödyntää kuitenkin pääsääntöisesti keskeytymättömiä asynkronisia operaatioita keskeytymättömien järjestelmäkutsujen välityksellä, esimerkiksi `epoll` Linux-käyttöjärjestelmissä ja `IOCP` Windows-käyttöjärjestelmissä. (Borghi 2023.)

Kun asynkroniset operaatiot valmistuvat, niiden käsittely eroaa operaation tyyppin mukaan. Keskeyttävien operaatioiden tapauksessa libuv-kirjaston työskentelijäsäie asettaa operaation tuloksen tapahtumasilmukan makrotehtäväjonoon, minkä jälkeen takaisinkutsu suoritetaan pääsäikeessä. Keskeytymättömissä operaatioissa libuv sen sijaan suorittaa järjestelmäkutsun ja tarkkailee operaation valmistumista. Node.js:n versiosta 20.3.0 lähtien tämä on toteutettu `io_uring`-rajapinnan avulla, joka käsittelee operaatioiden lähettämisen ja valmistumisen jonopohjaisesti. Operaatioiden valmistuessa libuv huolehtii takaisinkutsujen suorittamisesta pääsäikeessä. (Borghi 2023.)

Node.js:n tapahtumasilmukka koostuu useista eri vaiheista (Kuvio 5), joissa jokaisessa käsitellään tietyn tyyppisiä takaisinkutsuja omissa FIFO-periaatteella (First In First Out) toimivissa jonoissaan. Kun jokin operaatio valmistuu, sen takaisinkutsu lisätään sopivaan jonoon odottamaan suoritusta. Tämä monivaiheinen prosessi voi johtaa yllättävältä tuntuvaan suoritusrjestykseen, esimerkiksi `setImmediate`-funktiolla ajastettu koodi voi suoriutua ennen `setTimeout`-funktiolla ajastettua koodia, vaikka ne olisi kirjoitettu päinvastaisessa järjestyksessä. Tämä johtuu siitä, että `setImmediate` suoritetaan heti `poll`-vaiheen päätyttyä `check`-vaiheessa, kun taas `setTimeout` odottaa ajastimen täyttymistä `timers`-vaiheessa. (OpenJS Foundation 2024d.)



Kuvio 5. Tapahtumasilmukan vaiheet (OpenJS Foundation 2024d)

3.4 Express.js-ohjelmistokehys

Express.js on Node.js-ajoympäristöön kehitetty web-ohjelmistokehys, joka tunnetaan erityisesti sen joustavasta ja minimalistisesta lähestymistavastaan web-sovellusten kehittämiseen. Kehys ei aseta tiukkoja vaatimuksia sovelluksen rakenteelle tai käytettäville teknologioille. Express.js tarjoaa kehittäjille vapauden valita esimerkiksi haluamansa ORM-kirjasto tai mallinnusmoottori. (Github 2024.)

Express.js:n keskeisiin ominaisuuksiin kuuluvat tehokas reititys, erilaiset HTTP-apurit, sekä korkea suoritussyky. Kehys on suunniteltu erityisesti HTTP-palvelimien työkaluksi, ja sitä käytetään esimerkiksi SPA-sovellusten taustapalveluina, sekä muun muassa verkkosivustojen ja HTTP-rajapintojen toteuttamiseen. Express.js:n asentaminen tapahtuu npm-paketinhallintajärjestelmän kautta. (Github 2024.)

3.4.1 Middleware-funktiot

Express.js-ohjelmistokehyksessä sovellus toteutetaan väliohjelmistoarkkitehtuurilla, jossa middleware-funktiot muodostavat käsittelyketjun. Express-sovelluksen toiminnallisuus perustuu näiden funktioiden sarjaan, joilla jokaisella on pääsy pyyntöolioon, vastausolioon ja seuraavaan middleware-funktioon. Middleware-funktiot voivat suorittaa koodia, tehdä muutoksia pyyntö- ja vastausolioihin, päättää pyynnön käsittelyn tai kutsua seuraavaa middleware-funktiota pinossa. Jos middleware-funktio ei päättää pyyntö-vastaus-sykliä, sen täytyy kutsua next-funktiota siirtääkseen kontrollin seuraavalle middleware-funktiolle. (OpenJS Foundation 2024e.)

Express.js käyttää viittä eri middleware-tyyppiä: sovellustason, reititintason, virheenkäsittely-, sisäänrakennetut ja kolmannen osapuolen middleware-funktiot. Sovellustason ja reititintason middleware-funktioita voidaan käyttää valinnaisen liitospolun kanssa. Esimerkiksi `/user/:id`-polkuun liitetty middleware-funktio suoritetaan vain kyseiseen polkuun kohdistuvissa HTTP-pyyntöissä. Virheenkäsittely-middleware on erikoistunut käsittelemään sovelluksessa ilmeneviä virhetilanteita. Express.js tarjoaa myös valmiita sisäänrakennettuja middleware-funktioita esimerkiksi staattisten tiedostojen käsittelyyn ja JSON-pyyntöjen jäsentämiseen. Middleware-funktioita voidaan ketjuttaa yhteen, jolloin ne muodostavat tiettyyn liitoskohtaan sijoittuvan aliketjun. (OpenJS Foundation 2024e.)

3.4.2 Reititys

Express.js-ohjelmistokehyksessä reititys määrittelee, kuinka sovelluksen päätepisteet, eli URI:t vastaavat asiakaspyyntöihin. Reititys toteutetaan Express-sovelluksen metodien avulla, jotka vastaavat HTTP-metodeja, kuten `app.get()` GET-pyyntöjen ja `app.post()` POST-pyyntöjen käsittelyyn. Nämä reititysmetodit määrittelevät takaisinkutsufunktion, jota kutsutaan sovelluksen vastaanottaessa pyynnön määriteltyyn reittiin ja HTTP-metodiin. (OpenJS Foundation.)

Reittien polut voidaan määritellä merkkijonoina, merkkijonomalleina tai säännöllisinä lausekkeina, ja ne voivat sisältää reititysparametreja. Reititysparametrit ovat nimettyjä osia URL-osoitteessa, joiden arvot tallennetaan `req.params`-olioon. Express.js tukee myös modulaarista reititystä Router-luokan avulla, mikä mahdollistaa erillisten, liitettävien reitityskäsittelijöiden luomisen. (OpenJS Foundation.)

Reitityskäsittelijät voivat sisältää useita takaisinkutsufunktioita, jotka toimivat väliohjelmiston tavoin pyynnön käsittelyssä. Reitityskäsittelijät voidaan toteuttaa yksittäisenä funktiona,

funktioaulukkona tai näiden yhdistelmänä. (OpenJS Foundation.) Kuvan 3 koodiesimerkki havainnollistaa reitin määrittelyä, reititysparametriä, sekä modulaarista reititystä.

```
const express = require('express')
const router = express.Router()

// Määritellään takaisinkutsufunktio
const logRequest = (req, res, next) => {
  console.log('Pyyntö vastaanotettu:', req.params.id)
  next()
}

// Reitti, joka käyttää reititysparametria :id ja määriteltyä takaisinkutsufunktiota
router.get('/users/:id', logRequest, (req, res) => {
  res.send(`Käyttäjän ${req.params.id} tiedot`)
})

module.exports = router
```

Kuva 3. Express.js esimerkki

3.5 Tunnistautuminen Passport.js-kirjastolla

Passport.js on Node.js-ympäristöön Express.js-yhteensopivaksi kehitetty todennuksen väliohjelmisto, jonka tehtävä on pyyntöjen todentaminen. Kirjasto on modulaarinen ja sen toiminnallisuus perustuu satoihin erillisiin liitännäisiin, joita kutsutaan strategioiksi. Esimerkkejä strategioista ovat käyttäjätunnus-salasana-todennus, OAuth-pohjainen todennus esimerkiksi Facebookin tai Twitterin kautta, sekä OpenID-pohjainen todennus muun muassa Googlen välityksellä. Joustavuuden vuoksi Passport.js ei itsessään määritä reittejä tai tietokantaskeemaa. Kirjasto tarjoaa yksinkertaisen rajapinnan middleware-funktioiden muodossa: kehittäjä antaa Passport.js:lle pyynnön todennettavaksi, ja kirjasto tarjoaa toiminnot todennuksen onnistumisen tai epäonnistumisen käsittelyyn. (Hanson 2024a.)

Passport.js voi ylläpitää pysyviä kirjautumisistuntoja käyttäjän kirjautumistilan säilyttämiseksi useiden HTTP-pyyntöjen välillä. Istuntojen käyttöönotto vaatii express-session-npm-paketin sekä passport.initialize- ja passport.session-funktioiden käyttöä. Passport.js käyttää HTTP-evästä tunnistaakseen mihin istuntoon pyyntö kuuluu. Tyypillisessä toteutuksessa istuntoon tallennetaan vain käyttäjän tunniste, jonka avulla varsinaiset käyttäjätiedot voidaan hakea tietokannasta tarvittaessa. (Hanson 2024b.)

Esimerkkinä käyttäjätunnus-salasana-todennuksen toteuttamisessa LocalStrategy-komponentti konfiguroidaan antamalla sille vahvistusfunktio argumenttina. Tässä funktiossa määritellään, miten käyttäjä haetaan tietokannasta ja miten salasana varmistetaan. Tyypillisessä toteutuksessa funktio hakee ensin käyttäjän tietokannasta käyttäjätunnuksen perusteella. Jos käyttäjä löytyy, haetaan tietokannasta myös käyttäjän suolattu salasana ja suola-arvo. Annettu salasana suolataan ja hajautetaan samoilla parametreilla kuin alkuperäinen

salasana, minkä jälkeen hajautusarvoja verrataan keskenään. Jos käyttäjää ei löydy tai hajautusarvot eivät täsmää, funktio palauttaa epätosi-totuusarvon, ja mahdollisen virheviestin. Onnistuneessa todennuksessa funktio palauttaa käyttäjäolion, jonka pohjalta Passport.js tallentaa käyttäjän tunnisteen istuntoon `serializeUser`-funktion avulla. (Hanson 2024b)

3.6 SQLite-tietokanta

SQLite on kevyt, monialustainen ja vahvaan luotettavuuteen panostava SQL-tietokantamoottori, joka on toteutettu C-kielellä. SQLiten tiedostomuoto on vakaa, alustariippumaton ja taaksepäin yhteensopiva, minkä kehittäjät lupaavat säilyvän vuoteen 2050 asti. Lähdekoodi on julkisessa omistuksessa ja vapaasti kaikkien käytettävissä mihin tahansa tarkoitukseen. (SQLite Consortium 2024a.)

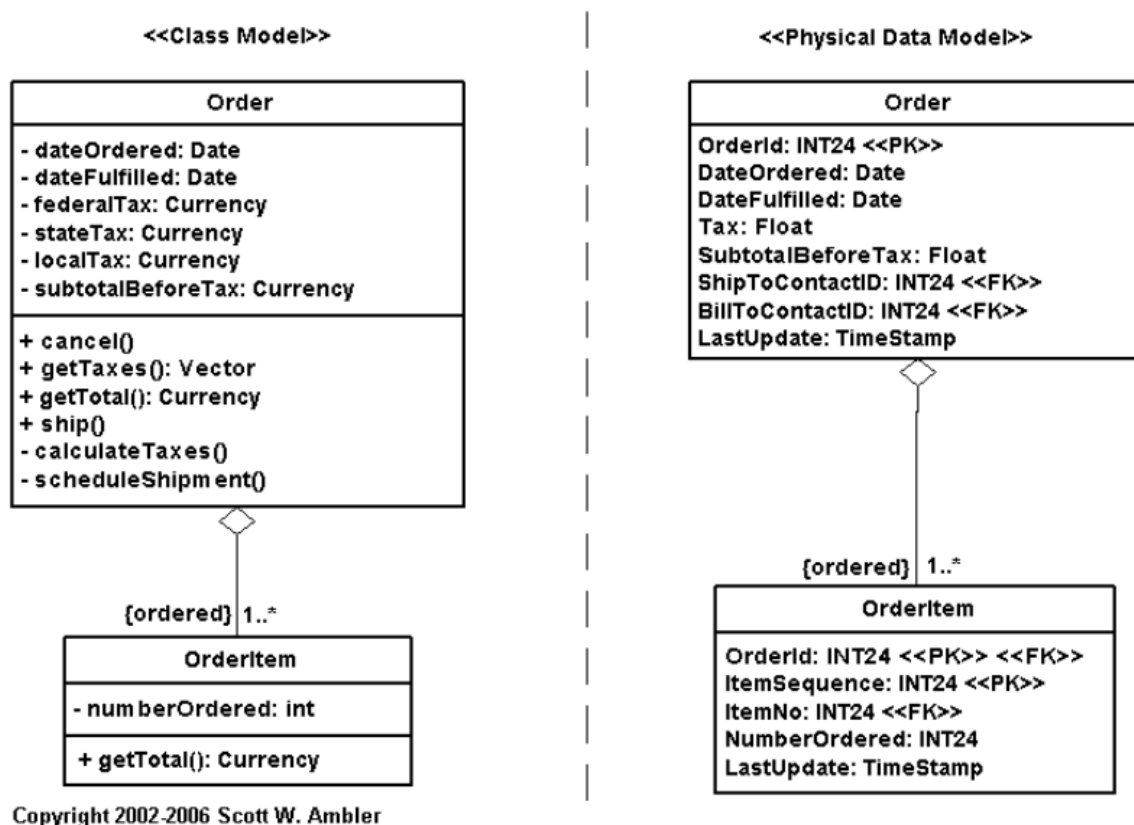
SQLite:ssä on tiettyjä rajoituksia SQL-ominaisuuksien toteutuksessa. Tietokantataulujen muokkaukseen liittyen `ALTER TABLE` -toiminnoista tuetaan vain taulujen ja sarakkeiden uudelleennimeämistä sekä sarakkeiden lisäämistä ja poistamista. Herättimistä SQLite tukee vain rivikohtaisia `FOR EACH ROW` -toteutuksia, mutta ei lausekohtaisia `FOR EACH STATEMENT` -herättimiä. `VIEW`-näkymät ovat vain luettavia, eikä niihin voi kohdistaa poisto-, lisäys- tai päivitysoperaatioita. Käyttöoikeuksien hallintaan liittyviä `GRANT`- ja `REVOKE`-komentoja ei ole toteutettu, sillä SQLite toimii tavallisena tiedostona, jolloin käyttöoikeudet määräytyvät käyttöjärjestelmän tiedosto-oikeuksien mukaan (SQLite Consortium 2024b.)

SQLite soveltuu käyttökohteisiin, joissa tietokantaan kohdistuu samanaikaisesti useita lukuoperaatioita mutta vähän kirjoitusoperaatioita. SQLite tukee rajoittamatonta määrää samanaikaisia lukijoita, mutta sallii vain yhden kirjoittajan kerrallaan. Kirjoitustapahtumat asetuvat jonoon, ja yksittäinen kirjoituslukitus kestää tyypillisesti muutamia millisekunteja. SQLite toimii työpöytäsovellusten tiedostomuotona, kuten versionhallintajärjestelmissä ja talousanalyysityökaluissa, sekä soveltuu pienen ja keskisuuren liikenteen verkkosivustoille, joilla on alle 100 000 käyntiä päivässä. SQLitea käytetään myös tietoanalyysiin, välimuistina yritystason datalle sekä palvelinpuolen tietokantana määrätyissä käyttötapauksissa. (SQLite Consortium 2024c.)

3.7 TypeORM-kirjasto ja olio-relaatio-mallinnus

TypeORM on Node.js-ympäristössä toimiva ORM-kirjasto, joka mahdollistaa tietokannan käsittelyn olio-ohjelmoinnin keinoin. Se tukee useita eri tietokantoja, kuten MySQL, PostgreSQL ja SQLite, ja sitä voidaan käyttää sekä JavaScript- että TypeScript-ohjelmointikielessä. (TypeORM 2024.)

Olio-relaatio-mallinnus on prosessi, jossa oliomallin luokat ja niiden väliset suhteet mallinetaan relaatiotietokannan tauluiksi ja niiden välisiksi suhteiksi (Kuvio 6). Yksinkertaisimmillaan yhden luokan ominaisuus voidaan kartoittaa yhteen tietokantataulun sarakkeeseen. Kartoitus monimutkaistuu, kun käsitellään olioiden välisiä suhteita, sillä relaatiotietokannoissa suhteet toteutetaan vierasavaimilla, kun taas olio-ohjelmoinnissa ne toteutetaan viittauksina toisiin olioihin. Tietokantakartoituksessa joudutaan myös usein lisäämään olioihin ylimääräisiä "varjotietoja", kuten pääavaimia ja rinnakkaisuuden hallintaan liittyviä aikaleimoja, joita tarvitaan tietokannan toiminnallisuuden toteuttamiseksi. (Ambler 2023.)



Kuvio 6. Olio-relaatio-mallintamisesimerkki (Ambler 2023)

TypeORM tukee sekä Active Record- että Data Mapper -suunnittelumalleja tietokannan käsittelyssä. Active Record -mallissa kyselyt ja tietokantaoperaatiot määritellään suoraan oliomallin luokissa, ja oliot itse vastaavat tiedon tallentamisesta, poistamisesta ja lataamisesta. Data Mapper -lähestymistavassa tietokantakyselyt määritellään erillisissä repositorioluokissa, jolloin oliomallin luokat keskittyvät vain tiedon määrittelyyn ja mahdollisiin yksinkertaisiin metodeihin. Data Mapper -mallin etuna on parempi ylläpidettävyys erityisesti suuremmissa sovelluksissa, kun taas Active Record -malli voi pitää pienemmissä sovelluksissa rakenteen yksinkertaisempaan, koska kaikki toiminnallisuus on määritelty suoraan

malliluokissa. (Active Record vs Data Mapper 2024.). Suunnittelumallien eroavaisuuksia on havainnollistettu Kuvassa 4.

// Active Records	// Data Mapper
<pre> @Entity() export class User extends BaseEntity { @PrimaryGeneratedColumn() id: number @Column() firstName: string @Column() lastName: string @Column() isActive: boolean static findByName(firstName: string, lastName: string) { return this.createQueryBuilder("user") .where("user.firstName = :firstName", { firstName }) .andWhere("user.lastName = :lastName", { lastName }) .getMany() } } // const user = new User() user.firstName = "Timber" user.lastName = "Saw" user.isActive = true await user.save() const timber = await User.findOneBy({ firstName: "Timber", lastName: "Saw" }); // or const timber = await User.findByName("Timber", "Saw") </pre>	<pre> @Entity() export class User { @PrimaryGeneratedColumn() id: number @Column() firstName: string @Column() lastName: string @Column() isActive: boolean } // const userRepository = dataSource.getRepository(User) const user = new User() user.firstName = "Timber" user.lastName = "Saw" user.isActive = true await userRepository.save(user) const timber = await userRepository.findOneBy({ firstName: "Timber", lastName: "Saw", }) </pre>

Kuva 4. Esimerkki Data Mapper ja Active Records eroista

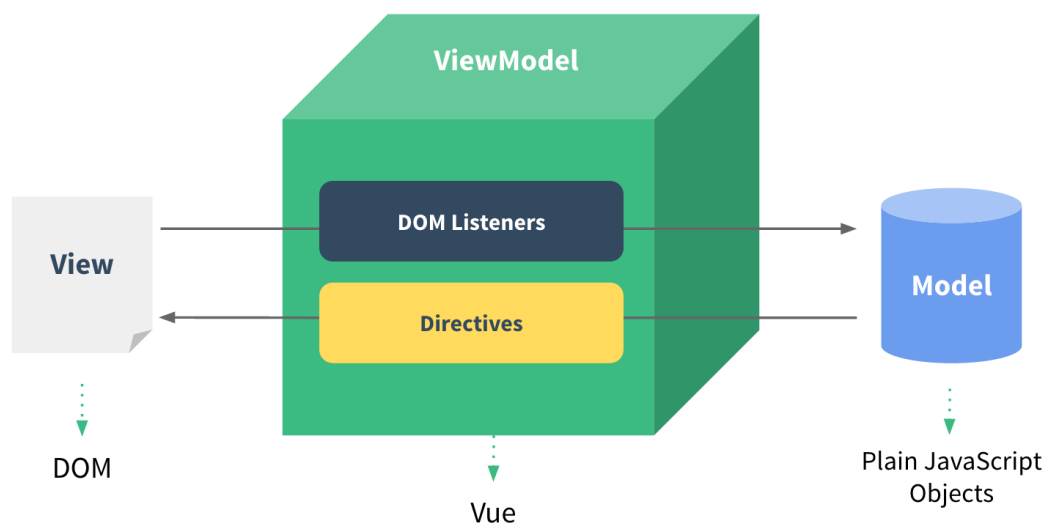
TypeORM käyttää TypeScript-kielen ominaisuutta nimeltä dekoraattorit, jotka ovat tapa lisätä merkintöjä luokkiin ja niiden osiin. Dekoraattori on yksinkertaisesti funktio, jota kutsutaan ajon aikana, ja se merkitään @-symbolilla. TypeORM:ssä dekoraattoreita käytetään määrittelemään, miten luokat ja niiden ominaisuudet liittyvät tietokantaan. Esimerkiksi @Entity()-dekoraattori kertoo luokan vastaavan tietokantataulua ja @Column() määrittelee luokan ominaisuuden vastaavan taulun saraketta. Dekoraattorien käyttö TypeScriptissä vaatii experimentalDecorators-asetuksen käyttöönoton kääntäjän asetuksissa. (Microsoft 2024d.)

3.8 Vue.js-ohjelmistokehys

Vue.js on JavaScript-ohjelmistokehys käyttöliittymien rakentamiseen. Se rakentuu HTML:n, CSS:n ja JavaScriptin päälle tarjoten deklarativisen, komponenttipohjaisen ohjelmointimalin, jonka avulla voidaan kehittää monimutkaisiakin käyttöliittymiä. Vue:n ydinominaisuuksiin kuuluu deklarativinen renderöinti, joka laajentaa perinteistä HTML:ää mallisyntaksilla, jolla HTML-tuloste voidaan kuvata JavaScriptin tilan perusteella. (Vue.js c.)

Vue:n toinen keskeinen ominaisuus on reaktiivisuus, jonka ansiosta ohjelmistokehitys seuraa automaattisesti JavaScriptin tilan muutoksia ja päivittää DOM:ia tehokkaasti muutosten tapahtuessa. Vue on suunniteltu joustavaksi ja asteittain käyttöönotettavaksi, minkä ansiosta sitä voidaan hyödyntää erilaisissa käyttötapauksissa yksinkertaisesta HTML:n tehostamisesta aina monimutkaisiin yksisivuisiin sovelluksiin ja palvelinpuolen renderöintiin. (Vue.js c.)

Vue.js toimii pohjimmiltaan MVVM-suunnittelumallin periaatteiden mukaisesti, keskittyen erityisesti näkymämalliin (Kuvio 7). Vue.js yhdistää näkymän ja mallin kaksisuuntaisella datasisidoksella, missä jokainen Vue-instanssi toimii näkymämallina. Vue.js käyttää DOM-pohjaista mallinnusta, jossa Vue-instanssi käy luomisvaiheessa rekursiivisesti läpi kaikki juurielementin alaiset DOM-solmut ja asettaa tarvittavat datasisidokset. Tämän jälkeen näkymä muuttuu reaktiivisesti datan muuttuessa. (Vue.js a.)



Kuvio 7. Vue.js ja MVVM-suunnittelumalli

Direktiivit ovat Vuen omia HTML-attribuutteja, jotka alkavat "v"-etuliitteellä ja kertovat Vue.js:lle, miten DOM-elementtiä tulee käsitellä. Esimerkiksi v-text-direktiivi ohjeistaa Vue.js:ää pitämään elementin tekstisisällön synkronoituna mallin datan kanssa (Kuva 5) Vue-instanssin välityksellä. (Vue.js.) Direktiivit mahdollistavat erilaisia DOM-elementtien käsittelytapoja, esimerkiksi v-bind-direktiivillä muokataan elementtien attribuutteja, v-for-direktiivillä kloonataan elementtejä taulukon perusteella ja v-on-direktiivillä liitetään tapahtuma-kuuntelijoita (Vue.js b).

```

<script setup>
const title = ref('Otsikko')
const message = ref('Hei maailma!')

const handleClick = () => {
  message.value = 'Tekstiä klikattu!'
}
</script>

<template>
  <h1>{{ title }}</h1>
  <div v-text="message" v-on:click="handleClick"></div>
</template>

```

Kuva 5. Esimerkki Vue.js:n direktiivien käytämisestä

3.9 Linux-palvelin

Linux-palvelimet soveltuvat ympäristöihin, joissa sovellukset palvelevat tuhansia käyttäjiä. Niillä voidaan suorittaa useita sovelluksia samanaikaisesti yhdellä palvelimella. Linux-palvelimille on ominaista vähäinen käyttökatkoaika, mikä tekee niistä käyttökelpoisia SaaS-työkalujen ja reaaliaikaisten sovellusten alustana. Komentorivipohjainen käyttö tekee Linux-palvelimista kevytrakenteisia, ja ne soveltuvat sekä fyysisiin että pilvipalvelinympäristöihin. Linux-palvelimilla voidaan hallita säilytystiloja, kontteja ja ohjelmistovarastoja, kuten Dockeria ja MySQL:ää, sekä optimoida järjestelmän suorituskykyä. (Red Hat 2022.)

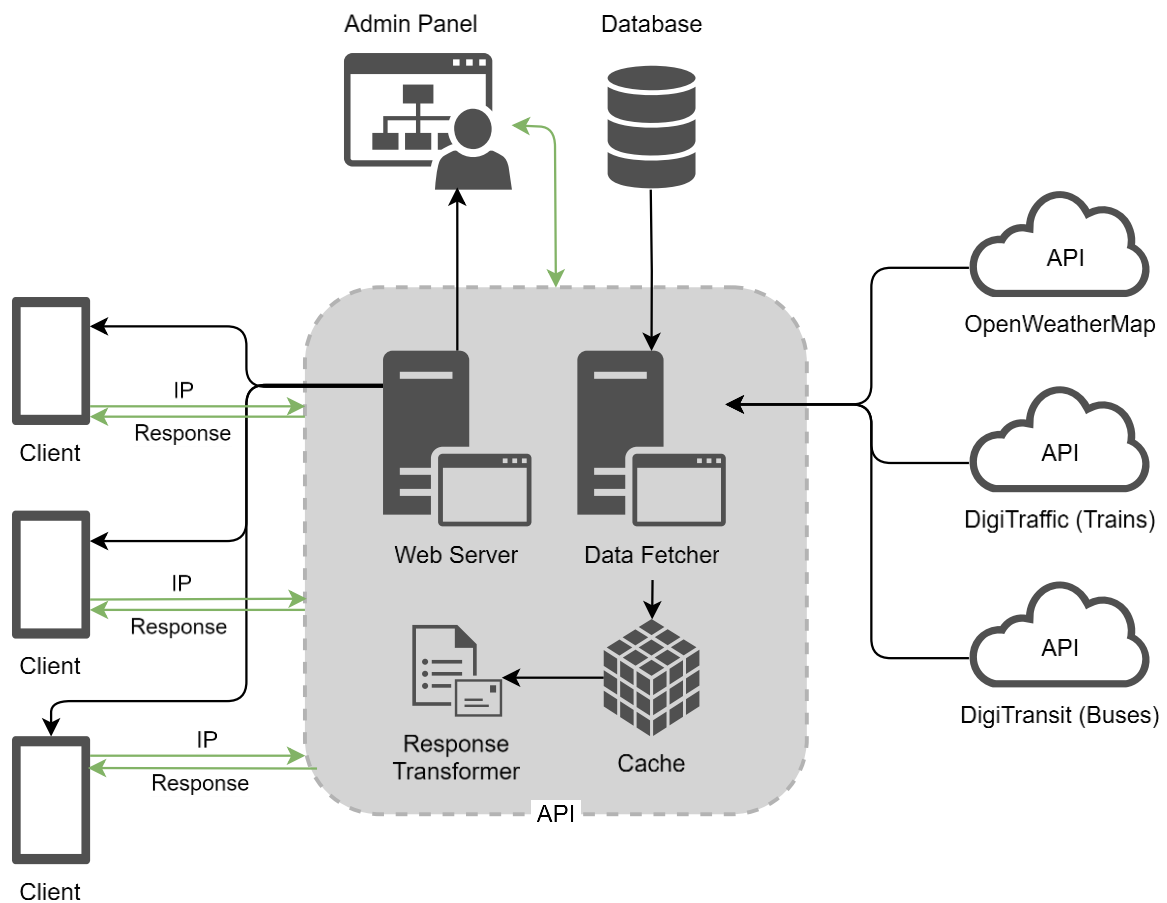
Systemd on nykyaikainen järjestelmä- ja palveluidenhallintaohjelma Linuxille. Se käynnistyy ensimmäisenä prosessina ja vastaa muun järjestelmän käynnistämisestä. Systemd:n tehokkuus perustuu sen kykyyn käynnistää palveluita rinnakkain ja tarpeen mukaan, käyttäen socket- ja D-Bus-aktivointia. Systemd:n toimintaa hallitaan ja valvotaan systemctl-komentorivityökalulla, jolla onnistuu niin palveluiden käynnistäminen, pysäyttäminen kuin järjestelmän tilan tarkkailu. (Debian 2024.)

Systemd:n toiminta perustuu yksikkötiedostoihin (Kuva 6), joiden avulla määritellään miten eri palvelut toimivat ja käynnistyvät. Järjestelmän mukana tulevat yksikkötiedostot löytyvät Debianin tapauksessa `/lib/systemd/system-hakemistosta`, kun taas järjestelmänvalvojan omat määrittelyt tehdään `/etc/systemd/system/-hakemistoon`. Kun halutaan palvelun käynnistyvän automaattisesti järjestelmän mukana, luodaan ensin yksikkötiedosto ja otetaan se sitten käyttöön systemctl enable -komennolla. (Debian 2024.)

4 Ohjelmiston toteutus

4.1 Infojärjestelmän rakenne pääpiirteittäin

Infojärjestelmän toteutus koostuu Noden Express-ohjelmistokehyksellä toteutetusta mono-liittisestä taustapalvelusta, sekä asiakaspäätteiden ja hallintapaneelin asiakaspuolen ohjelmistoista. Taustapalvelu sisältää useita eri komponentteja (Kuvio 8). Taustapalvelun sisäinen web-palvelin tarjoilee hallintapaneelin ja asiakaspäätteen käyttöliittymäohjelmistot.



Kuvio 8. Infojärjestelmän rakenne

Tuotantoympäristössä taustaohjelmisto toimii Ubuntu-käyttöjärjestelmällä varustetulla palvelimella systemd-palveluna. Palvelu käynnistyy automaattisesti palvelimen käynnistyessä. Systemd:n komentorivityökalulla journalctl voidaan selata lokitietoja. Samalla palvelimella käytetään myös SQLite-tiedostotietokantaa tietojen tallennukseen.

Järjestelmään on integroitu mukaan kolme eri ulkoista rajapintaa. Suomen valtion omistama Liikenteenohjausyhtiö Fintraffic tarjoaa rajapinnan juna-aikataulutiedoille. DigiTransit on HSL:n, Fintrafficin sekä TVV LMJ Oy:n tarjoama palvelualusta ja rajapinta muun muassa linja-autoaikatauluille. Aikatauluihin liittyvissä ohjelmistoissa tyypillisesti ilmoitetaan myös

säättietoja, jonka vuoksi järjestelmään toteutettiin integraatio OpenWeatherMapin rajapintaan.

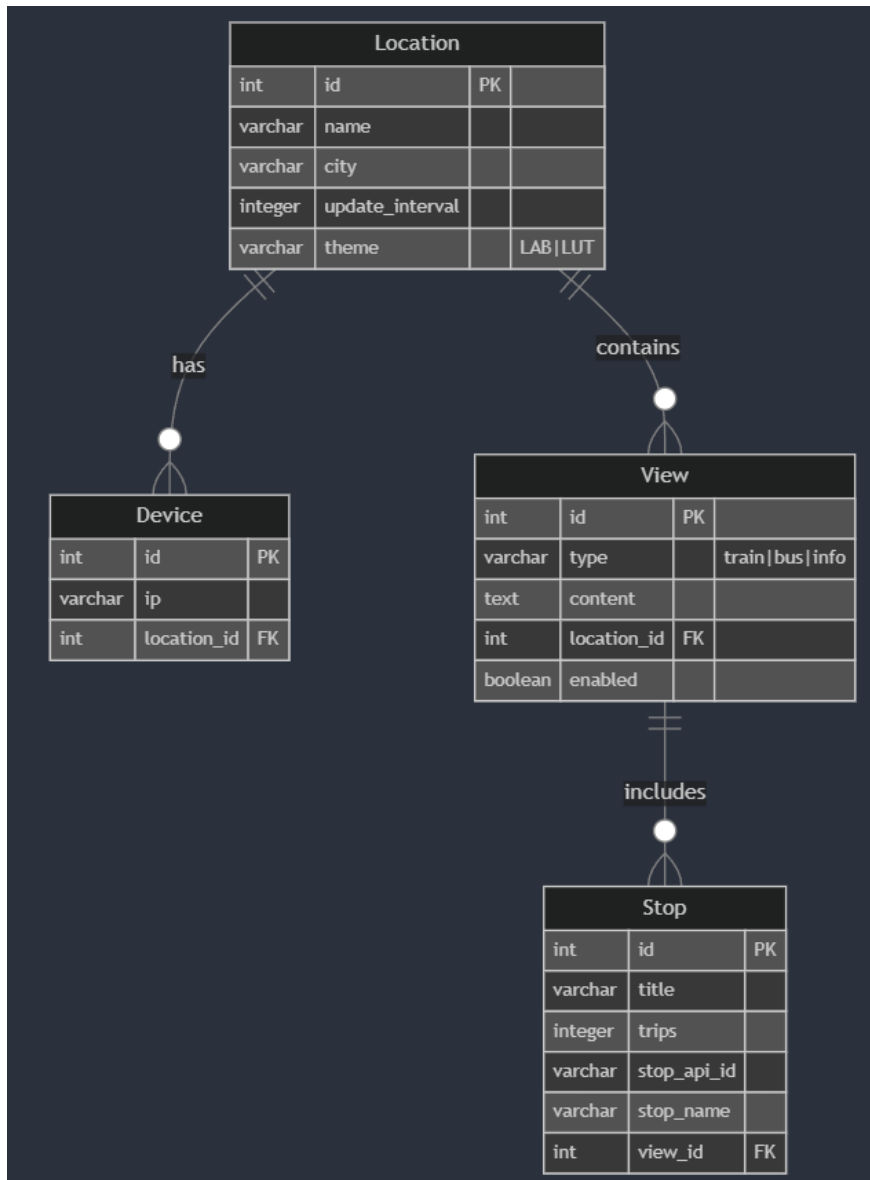
4.2 Taustapalvelun tarkempi rakennekuvaus

Taustapalvelun ydinkomponentit koostuvat tiedonhakukomponenteista, välimuistiratkaisusta sekä tietokannan käsittelyä varten luodusta palveluluokasta. Luokkien toteutuksissa hyödynnettiin TypeScriptin tyyppijärjestelmää ja geneerisiä rakenteita koodin uudelleenkäytettävyyden parantamiseksi.

Komponenttien välinen kommunikointi noudattaa funktionaalista työnjakoa, jossa tiedonhakukomponentit, välimuisti ja tietokantapalvelu toimivat itsenäisinä kokonaisuuksina. Tällä rakenteella pyrittiin joustavampaan taustapalvelun jatkokehitykseen.

4.2.1 Tietorakenteet

Informaatiojärjestelmän tietokantarakenteet koostuvat neljästä päätaulusta: Location, Device, View ja Stop. Tietokantarakenteet suunniteltiin tukemaan järjestelmän keskeisiä vaatimuksia sijaintipohjaisesta tietojen näyttämisestä ja joustavasta laajennettavuudesta. Rakenne määrittelee samalla järjestelmän tietomallin ja toimii sovelluskehitystä ohjaavana teknisenä määrittelynä, jonka pohjalta toteutettiin sekä tietokantakerros että sovelluslogiikka. Taulujen väliset relaatiot ja rakenteet on esitelty kuviossa 9.



Kuvio 9. Infojärjestelmän tietokantaskeema

Location-taulu on järjestelmän perusta, ja se sisältää sijainteihin liittyvät määrittelyt. Jokaisella sijainnilla on yksilöllinen nimi, kaupunkitieto, näkymien päivitysväli sekä teeman nimi. Kaupunkitietoa käytetään säätietorajapinnan hakujen kohdistamiseen. Jokaiseen sijaintiin liittyy laitteita Device-taulusta ja yksi tai useampi View-taulun mukainen näkymä.

Device-taulu kuvastaa asiakaspäätteitä. Jokainen pääte yksilöidään IP-osoitteen perusteella. Järjestelmä on suunniteltu toimimaan suljetussa verkossa, jossa IP-osoitteet voidaan varata MAC-osoitteiden perusteella. Laitteet yhdistetään sijaintitietoihin location_id-viiteavaimen avulla.

View-taulu määrittelee asiakaspääteen eri näkymien tiedot. Näkymät voivat olla tyypiltään joko juna- tai linja-autoaikatauluja, eli arvoiltaan train tai bus. Järjestelmään oli alun perin suunniteltu myös yleinen informaationäkymätyyppi, jota ei toteutettu sen pidemmälle tässä

projektissa. Tauluun jätettiin kuitenkin content-sarake infonäkymän sisällölle, sekä näkymän tyypin arvo info. Näkymät linkittyvät sijainteihin location_id-viiteavaimen avulla. Asiakastapaamisissa ehdotettiin ominaisuutta, jonka avulla voidaan piilottaa näkymiä tilapäisesti, joten enabled-sarakkeen avulla ilmaistaan sitä, tulisiko näkymää näyttää.

Stop-taulu sisältää pysäkki- tai asematiedot, jotka näytetään eri näkymissä. Pysäkeillä on asiakaspääteille näkyvä otsikko ja näytettävien vuorojen määrä. Rajapinnan pyyntöjä varten tarvitaan pysäkkitunniste stop_api_id, sekä rajapinnan määrittelemä pysäkin nimi stop_name. Pysäkit linkittyvät näkymiin view_id-viiteavaimen kautta.

Järjestelmänhallitsijan tiedoille on luotu yksikertainen tietokantarakenne. Admin-taulu sisältää käyttäjätunnuksen sekä suolatun salasanan. Pääavaimena toimii juokseva id-numero.

TypeORM selkeytti teknistä toteutusta. Relaatiot ja mallit olivat selkeä määritellä, ja niiden käsittely oli tyyppiturvallista TypeScriptin tyyppijärjestelmän ansiosta. TypeORM:n dekooraattori-pohjainen syntaksi teki tietorakenteiden määrittelyistä suoraviivaista, kuten kuva 7 havainnollistaa View-mallin osalta.

```
export type ViewType = 'train' | 'bus' | 'info';

@Entity()
export class View {
  @PrimaryGeneratedColumn()
  id: number;

  @Column({ type: 'varchar' })
  type: ViewType;

  @Column({ type: 'text', nullable: true })
  content: string | null;

  @ManyToOne(() => Location, (location) => location.id)
  @JoinColumn({ name: 'location_id' })
  location: Relation<Location>;

  @OneToMany(() => Stop, (stop) => stop.view, { cascade: true })
  stops: Relation<Stop[]>;

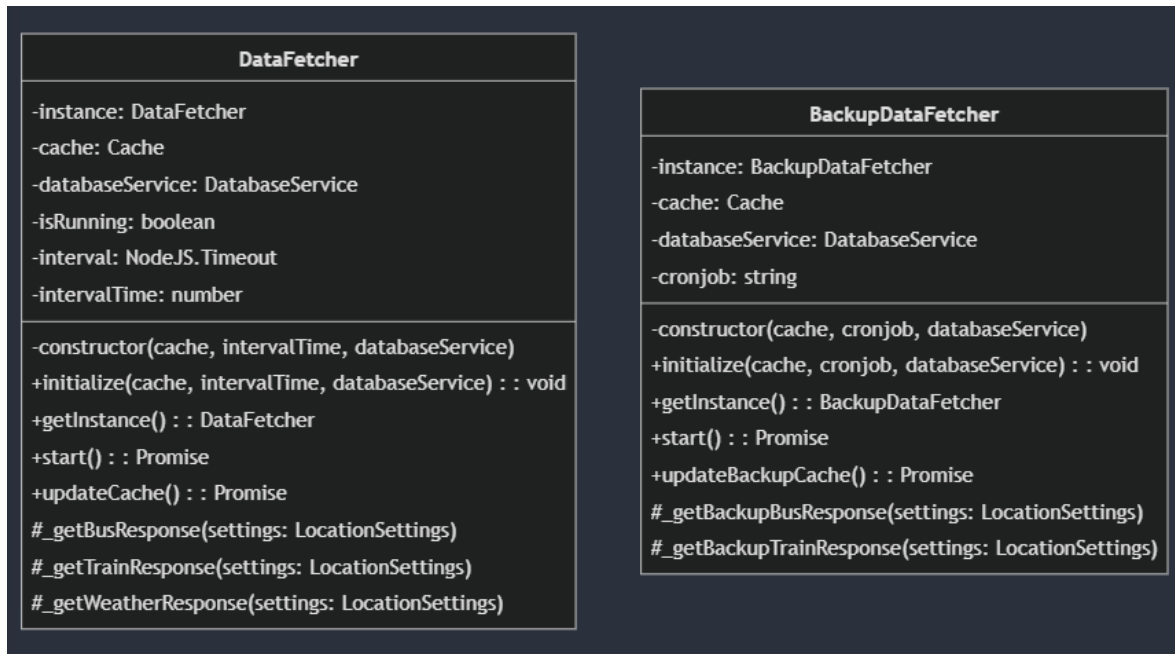
  @Column({ type: 'boolean', default: false })
  enabled: boolean;
}
```

Kuva 7. View-malli TypeORM:llä

4.2.2 Tiedonhallintakomponentit

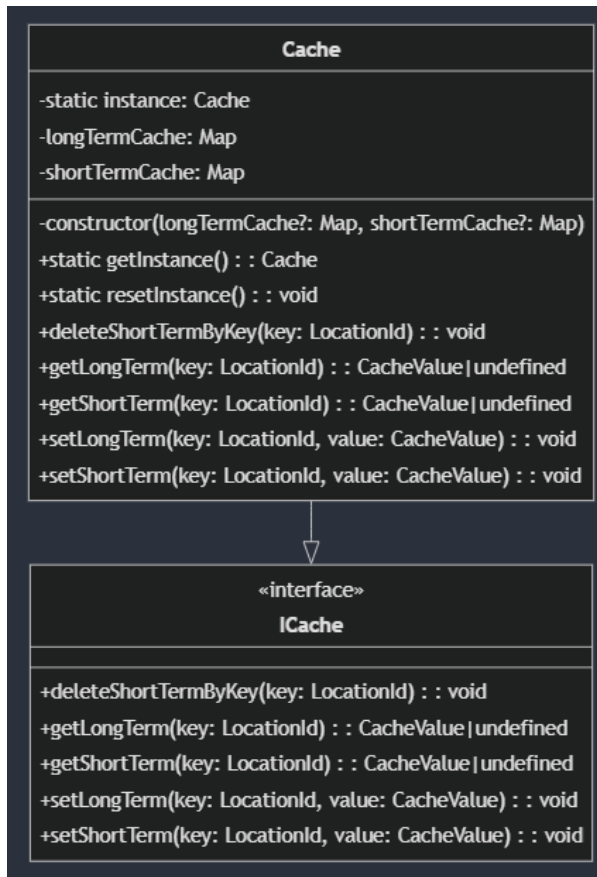
Data Fetcher- ja Backup Data Fetcher -komponentit (Kuvio 10) noutavat ulkoisista kolmannen osapuolen rajapinnoista tietokantaan määriteltyjen sijaintien perusteella juna- ja linja-autoaikataulut sekä säätiedot. Data Fetcher hakee tietoja määritellyin aikavälein ja tallentaa vastaukset välimuistiin muuttumattomina. Hakuväli on konfiguroitavissa sovelluksen env-tiedostossa. Backup Data Fetcher puolestaan hakee kerran vuorokaudessa koko

vuorokauden aikataulutiedot varmuuskopiona, ja sen ajastus määritellään cron-syntaksilla myös env-tiedostossa. Molemmat komponentit on toteutettu Singleton-suunnittelumallin mukaisesti, minkä tarkoitus on varmistaa, että järjestelmässä on vain yksi aktiivinen instanssi kumpaakin tiedonhakukomponenttia.



Kuvio 10. DataFetcher ja BackupDataFetcher luokat

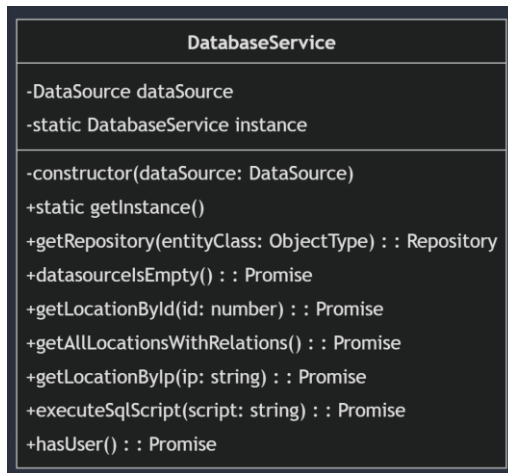
Cache-luokka (Kuvio 11) edustaa välimuistiratkaisua, joka on suunniteltu julkisten rajapintapyyntöjen vastauksien tallentamiseen ja hakemiseen. Luokan ydinajatuksena on tarjota kaksi eri tavalla nimettyä ajonaikaista välimuistia: lyhytaikainen ja pitkäaikainen. Singleton-toteutukseen päädyttiin varmistaakseen tiedon yhtenäisyys koko sovelluksessa.



Kuvio 11. Välimuisti-luokka

Teknisesti luokka hyödyntää TypeScriptin Map-tietorakennetta tallennukseen ja se toteuttaa erikseen luodun ICache-rajapinnan, joka määrittelee välimuistin perustoiminnallisuuden ja tietotyypit. Tyypin `LocationId` toimii välimuistin avaimena ja välimuistin tallennusrakenteen määrittelevä `CacheValue` tallennettavana arvona. Luokan toteutus sisältää metodit molempien välimuistien hallintaan. Lisäksi luotiin erillinen `resetInstance`-metodi mahdollisen yksikkötestauksen helpottamiseksi. Tämän kaksitasoisen välimuistiratkaisun pyrkimyksenä on tarjota asiakaspäätteille haettu tieto taustapalvelun omasta välimuistista, jotta välttyttiin ylimääräisiltä pyynnöiltä julkisille rajapinnoille. Lisäksi sen tarkoitus on tehostaa soveluksen toimintavarmuutta.

Database Service luokkaan (Kuvio 12) abstrahoitii yleisimpiä tietokantatoimintoja. Luokka tarjoaa rajapinnan TypeORM:n Repository-objektien hakemiseen sekä useita metodeja tietokantakyselyihin, kuten sijaintitietojen hakemisen IP-osoitteen perusteella ja tietokannan tyhjyyden tarkistamisen. Luokan vastuulla on myös SQL-skriptien suorittaminen ja käyttäjätietojen olemassaolon varmistaminen, mikä mahdollistaa tietokannan alustamisen ja ylläpidon automatisoidusti.



Kuvio 12. Database Service -luokka

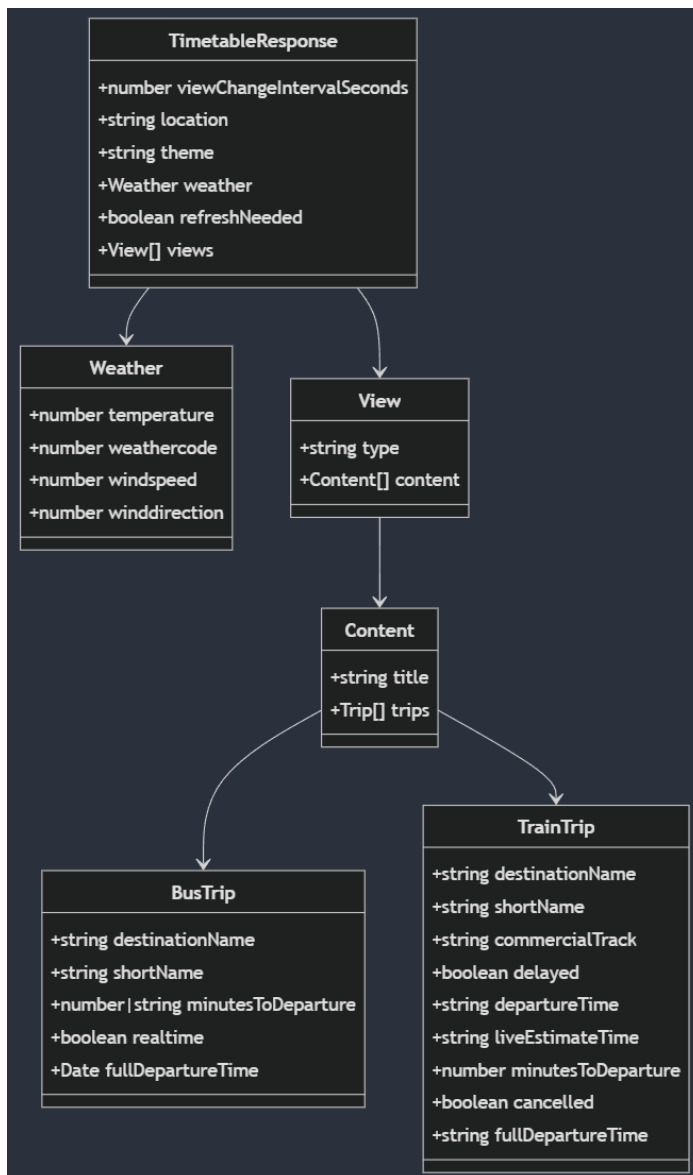
4.2.3 Integraatiokomponentit

BusApi ja TrainApi -komponentit rakennettiin hakemaan aikataulutietoja eri rajapinnoilta. BusApi käyttää Digitransitin GraphQL-rajapintaa linja-autoaikataulujen hakemiseen, kun taas TrainApi hyödyntää Digitrafficin GraphQL-rajapintaa junaliikenteen aikatauluille. Molemmista komponenteista päämetodit `fetchBusData` ja `fetchTrainData` ottavat parametreiksi pysäkkien tai asemien tunnisteet ja haluttujen aikataulurivien määrän. Pyyntöjen muodostamisesta vastaavat erilliset `makeJsonQuery`-apufunktiot, jotka kokoavat GraphQL-kyselyt. Komponentit sisältävät myös virheenkäsittelylogiikkaa ja metodit vuorokauden kattavien vara-aikataulujen hakemiseen.

Aikataulutietojen muokkauksesta vastaa erillinen muunnoskomponentti, joka käsittelee rajapinnoilta saadun datan yhtenäiseen muotoon. Komponentin `transformBusData`-metodi muuntaa bussiaikataulut ja `processTrainData`-metodi juna-aikataulut sovelluksen sisäiseen tietorakenteeseen. Molemmat metodit laskevat lähtöajat minuutteina nykyhetkestä ja järjestävät vuorot aikajärjestykseen. `MakeFinalJson`-metodi kokoaa muunnetut aikataulut, sää-tiedot ja sijaintiasetukset yhtenäiseksi JSON-rakenteeksi. Komponentti lisää tarvittaessa päivämäärärivit aikatauluihin vuorokauden vaihtuessa. Komponentti sisältää myös logiikan päivitystarpeen muokkaamisesta palautukseen.

4.3 Rajapinnat

Järjestelmän keskeinen rajapinta on `/timetables`-reitti, joka toimii pääasiallisena tietopisteenä asiakaspäätteiden ja palvelimen välillä. Reitti vastaanottaa HTTP GET -pyyntöjä ja palauttaa JSON-muotoisen vastauksen (Kuvio 13), joka sisältää tiedot julkisen liikenteen aikatauluista, säätilasta sekä asiakaspäätteen muista asetuksista.



Kuvio 13. Luokkakaavio timetables-rajapinnan JSON-vastauksesta

Tietokantaan tallennettujen tietojen muokkaamista varten on luotu RESTful-periaatteita mukaileva HTTP-pohjainen rajapinta, jossa jokainen tietokantarakenne toimii resurssina. Jokaiselle tietorakenteelle on omat geneerisistä middleware-funktioista luodut reittinsä niiden noutamiseen, poistamiseen, sekä lisäämiseen että päivittämiseen. Reitit vaativat tunnistautumisen. Rajapintaa esitellään Location-resurssin osalta taulukossa 1.

Metodi	Reitti	Kuvaus	Vastaus
GET	/locations	Palauttaa kaikki sijainnit. Kyselyparametri relations määrittää haetaanko liittyvät entiteetit. Parametrin arvo <i>true</i> sisällyttää laitteet, näkymät ja pysäkit, <i>false</i> (oletus) palauttaa vain sijainnit. Tulokset järjestetään ID:n mukaan nousevasti.	200 OK ja JSON-taulukko sijainneista
POST	/locations	Lisää uuden sijainnin tai päivittää olemassaolevia. Vastaa yhden tai useamman sijainnin tiedot JSON-muodossa.	200 OK ja tallennetun/päivitetyt resurssin tiedot
DELETE	/locations/:id	Poistaa sijainnin annetun ID:n perusteella.	204 No Content jos poisto onnistui, 404 Not Found jos sijaintia ei löytynyt

Taulukko 1. Locations-resurssin rajapintadokumentaatio

TypeScriptin geneeristen luokkien ja Expressin middleware-funktioiden avulla useiden mallien toteutuksessa vähennettiin koodin toistoa. Yhtenäiset geneeriset funktiot vastasivat yleisestä toimintalogiikasta, minkä pohjalta tyyppikohtalaiset toteutukset luotiin. Geneerisen funktion rakennetta esitetään kuvassa 7.

```
/**
 * Generic middleware function to fetch entities with or without relations.
 * @param req
 * @param res
 * @param entityClass
 * @param options
 */
export async function responseToGet<T extends ObjectLiteral>({
  req: Request,
  res: Response,
  entityClass: ObjectType<T>,
  options?: { relations?: string[]; order?: object }
}): Promise<void> {
  const databaseService = await DatabaseService.getInstance();
  const repository = databaseService.getRepository(entityClass);
  const entities = await repository.find(options);

  res.json(entities);
}
```

Kuva 7. Geneerinen middleware-funktio GET-pyntöön vastaamiseen

Järjestelmään toteutettiin HTTP-pohjainen rajapinta käyttäjähallintaa ja tunnistautumista varten Passport.js-kirjaston avulla. Se mahdollistaa käyttäjien rekisteröitymisen, kirjautumisen sekä salasanan vaihdon. Tunnistautuminen perustuu istuntoihin, joissa käyttäjätiedot

tarkistetaan ja istunto ylläpidetään palvelimella. Salasanat talletetaan tietokantaan tiivisteinä bcrypt-kirjaston avulla.

Järjestelmänhallitsijan rajapinta sisältää sekä julkisia että isAuthenticated-middlewarella suojattuja reittejä. Suojatut reitit varmistavat, että vain kirjautuneet käyttäjät pääsevät käsiksi resursseihin. Järjestelmä sisältää toiminnallisuudet käyttäjän tilan tarkistamiseen, ensimmäisen pääkäyttäjän luomiseen sekä hallintakäyttäjien tietojen hakemiseen ja päivittämiseen.

Kirjautuminen ja tunnistautuminen toteutettiin varsin kevyenä ratkaisuna web-palveluksi. Asiakas ei kokenut, että olisi ollut tarvetta useammalle järjestelmäkäyttäjätunnukselle. Myös tietoturvan osalta asiakkaan kanssa pohdittiin, että järjestelmä ei sisällä arkaluonteista materiaalia ja toimii vain yksityisessä ja tarkkaan valvotussa verkossa.

Välimuistin tyhjentämiselle luotiin oma rajapintansa, johon välitetään HTTP-pyyntö POST-metodin avulla tunnistautuneena. Kyseisen reitin tarvetta ja toimintaa käsitellään enemmän luvussa 4.1.

4.1 Toimintaperiaatteet

Järjestelmän arkkitehtuuri perustuu kaksitasoiseen välimuistiratkaisuun. Taustapalvelu vastaa tietojen hakemisesta ulkoisista rajapinnoista sekä niiden prosessoinnista. Välimuistiratkaisun avulla optimoidaan rajapintakutsujen määrää ja parannetaan järjestelmän suorituskykyä.

Toiminta jakautuu kahteen erilliseen sykliin: aikataulutietojen hakemiseen rajapinnoista ja tietojen jakeluun asiakaspääteille. Taustapalvelu päivittää tietoja itsenäisesti määriteltujen aikavälien mukaisesti. Asiakaspäätteet puolestaan noutavat ajantasaiset tiedot välimuistista oman päivityssyklinsä mukaisesti.

4.1.1 Aikataulujen hakeminen ja tallentaminen

Järjestelmän taustapalvelu hakee käynnistyksen yhteydessä sää- ja aikataulutiedot ulkoisista rajapinnoista sekä sijaintiin liittyvät tiedot tietokannasta. Nämä tiedot tallennetaan välimuistiin alkuperäisessä muodossaan jatkokäsittelyä varten.

Välimuistin pitkäaikainen tietorakenne on suunniteltu 24 tunnin aikaikkunalle, sillä seuraavan vuorokauden aikataulutiedot tulevat aina saataville aamuyön tunteina. Nämä ovat kuitenkin ennustettuja lähtö- ja saapumisaikoja. Järjestelmän ensisijainen tietolähde koostuu lyhyemmän lähes reaaliaikaisesti muuttuvista aika-arvioista, jotka taustapalvelu hakee

oletusarvoisesti 30 sekunnin välein. Nämä hakusykli toistuvat itsenäisesti riippumatta asiakaspäätteiden pyynnöistä (Kuvio 14).



Kuvio 14. Lyhytaikainen tiedonhakusykli

Käytetyt julkiset rajapinnat eivät sisältäneet muutostentunnistusmekanismia tai versiointia, jonka avulla täysi tietopyyntö olisi voitu suorittaa vain uuden tiedon ollessa saatavilla. Aikataulujen rajapinnat olisivat tarjonneet täysin reaaliaikaisen sijainninseurannan, mutta järjestelmän suunnittelussa priorisoitiin arvioituja saapumisaikoja reaaliaikaisen sijaintiseurannan sijaan, sillä tämän todettiin palvelevan paremmin käyttötarkoitusta.

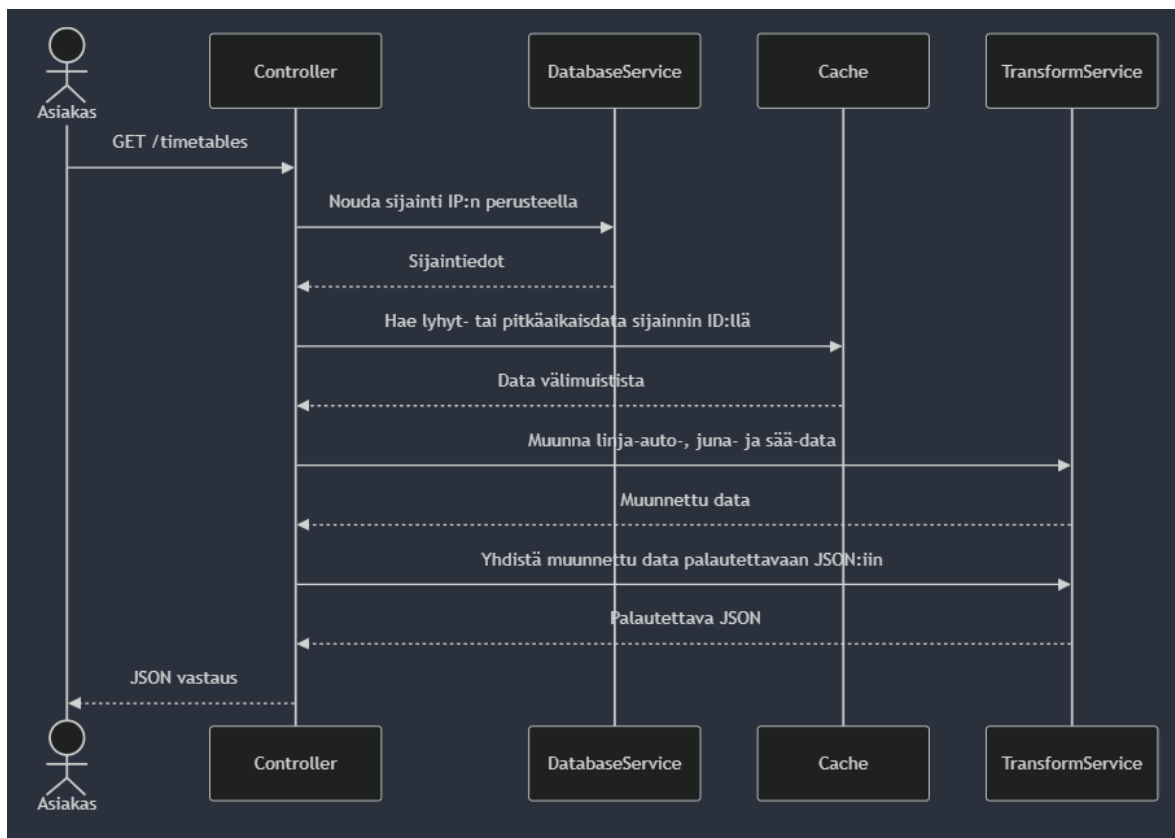
Rajapintojen ohjeistuksissa ilmaistiin, että kohtuullinen käyttö on sallittua ilman erillistä maksua. Asiaa ei määritelty tarkemmin. Paikallisen välimuistiratkaisun avulla julkisten rajapintakutsujen määrä pyritään pitämään hallittavana ja kohtuullisena. Tämä arkkitehtuuriratkaisu mahdollistaa sen, että asiakaspäätteiden määrän kasvu ei suoraan korreloi julkisten

rajapintakutsujen määrän kanssa. Rajapintakutsujen määrä skaalautuu vain erillisten sijaintien mukaan.

Välimuistiratkaisun toinen keskeinen peruste oli pyrkimys toimintavarmuuden takaamiseen. Kerran vuorokaudessa haettavat aikataulut tallennetaan välimuistiin, josta niitä voidaan hyödyntää tilanteissa, joissa rajapintayhteydet ovat tilapäisesti poikki.

4.1.2 Asiakaspäätteiden kanssa kommunikointi

Asiakaspäätteille suunnattu /timetables-rajapinta hyödyntää IP-osoitetta sijainnintunnistuksessa. IP-osoitteet ja niihin liitetyt sijainnit määritellään hallintapaneelista. Reittikäsittelijä määrittää pyyntöjen IP-osoitteiden perusteella vastauksiin sisällytettävät tiedot ja hakee ne välimuistista. Jos IP-osoitetta ei ole ilmoitettu, palautetaan sijaintilistan ensimmäisen sijainnin tiedot oletusarvona. Reittikäsittelijän toimintaa ja muunnosprosessia on havainnollistettu kuviossa 15.



Kuvio 15. Asiakaspäätteen rajapinnan toimintamalli

Jos päivitettyjä tietoja ei ole saatavilla, käytetään saumattomasti pitkäaikaisvälimuistiin tallennettuja tietoja. Ennen asiakaspäätteille välittämistä muunnoskomponentti prosessoi välimuistista noudetun datan asiakaspäätteille sopivaan muotoon.

Kehitystyön yhteydessä huomattiin, että satunnaisesti toistuvissa tilanteissa asiakaspääteen tietojen päivittäminen hallintapaneelin kautta ei täysin toiminut ilman asiakaspääteen selaimen manuaalista sivun päivittämistä. Tämän vuoksi asiakaspäätteille palautettavaan JSON-vastaukseen lisättiin totuusarvoinen `refreshNeeded`-kenttä, jonka arvoksi asetetaan `true` vain silloin, kun välimuistin päivittämiseen liittyvää ominaisuutta käytetään. Lisäksi luotiin `RefreshManager`-komponentti, joka pitää ajonaikana IP-osoitteen perusteella huolen, että asiakaspäätteille lähetetään vain yhden kerran viesti päivityksentarpeesta.

Pitkäaikaiseen välimuistiin saattoi myös jäädä vanhentuneet tiedot, jos muutoksia tehtiin esimerkiksi pysäkkeihin ja vuorojen määriin. Erillinen hallintapaneelin välityksellä käytettävän reitti välimuistin tyhjentämiseen suorittaa pitkäaikaisvälimuistin tietojen uudelleenhakemisen välittömästi. Toiminto päätettiin erottaa erilliseksi komennoksi, jota voidaan käyttää manuaalisesti. Tällä pyrittiin välttämään sitä, että tyhjentämistä ei tehtäisi automaattisesti jokaisen tallennetun muutoksen yhteydessä, mikä saattaisi johtaa tarpeettomiin ja ylimäärisiin laajoihin rajapintakutsuihin.

4.2 Järjestelmänhallinta

Järjestelmänhallinta tapahtuu hallintapaneelin välityksellä. Hallintapaneelilla on oma URL-polkunsa, mutta sama taustapalvelun web-palvelin tarjoilee sekä hallintapaneelin että asiakaspääteiden frontend-ohjelmistot. Kun palvelu ensimmäistä kertaa käynnistetään, pyydetään hallintapaneelin sivulla luomaan järjestelmäkäyttäjän tunnus. Hallintapaneelissa on erilliset näkymät laitteille, sijainneille, ohjeille sekä järjestelmän muuhun hallintaan.

Ulkoasun värit ja fontit valittiin LAB-ammattikorkeakoulun brand-oppaan ohjeistusten mukaisesti. Samaa opasta käytettiin myös asiakaspääteen käyttöliittymän suunnittelussa. Suunnittelun periaate oli pyrkiä yksinkertaisuuteen ja selkeyteen. Hallintapaneelia olettaen käytetään harvakseltaan, joten käyttöliittymän tulee olla mahdollisimman yksiselitteinen ja selkeäluukuinen.

Teknisen toteutuksen osalta projektissa päädyttiin käyttämään Vue.js-ohjelmistokehystä sekä asiakaspääteessä että hallintapaneelissa. Vue.js-ohjelmistokehystä on usein kuvailtu kevyeksi ja tehokkaaksi, mutta monipuoliseksi, joten se oli sopiva ja yhtenäinen valinta myös hallintapaneeliin. Uudelleenkäytettävät komponentit, reaktiivinen datamalli ja Vuex-tilahallinta osoittautuivat kehitystyötä helpottaviksi valinnoiksi hallintapaneelin osalta.

Laitteiden näkymässä (Kuva 8) voidaan lisätä ja poistaa laitteita. Laitteille voidaan määrittellä IP-osoite ja tieto sijainnista. Saatavilla olevat sijainnit haetaan valmiiksi alasvetovalikkoon.

Laitteet Sijainnit Ohjeet Hallinta

Laite

Poista

IP 123.123.123.123

Sijainti Lahden kampus

Laite

Poista

IP 1

Sijainti Lahden kampus

Laite

Poista

IP 111.222.333.444

Sijainti Lappeenrannan kampus

Lisää laite

Kuva 8. Hallintapaneelin käyttöliittymä Laitteet-näkymästä

Sijaintien kuvaamisessa päädyttiin visuaaliseen ratkaisuun, joissa jokainen rakenne pitää sisällään siihen liitoksissa olevat alirakenteet (Kuva 9). Asiakkaan kanssa käytyjen keskustelujen ja käyttöönottokoulutuksen perusteella tämä ratkaisu oli nopeasti ymmärrettävä.

Laitteet
Sijainnit
Ohjeet
Hallinta

Sijainti

Poista

Nimi

Lahden kampus

Näkymien vaihtumisaika (sek)

5

Kaupunki

Lahti

Teema:

LAB

Piilota lisätiedot

Näkymä

Poista

Näkymän tyyppi

Linja-auto

☒ Päällä

Pysäkki

Poista

Otsikko

To Travel Centre

Pysäkkitunnus

103578

Matkoja

4

Pysäkki

Poista

Otsikko

To Mikkula

Pysäkkitunnus

103579

Matkoja

4

Lisää pysäkki

Näkymä

Poista

Näkymän tyyppi

Juna

☒ Päällä

Pysäkki

Poista

Otsikko

Departures

Pysäkkitunnus

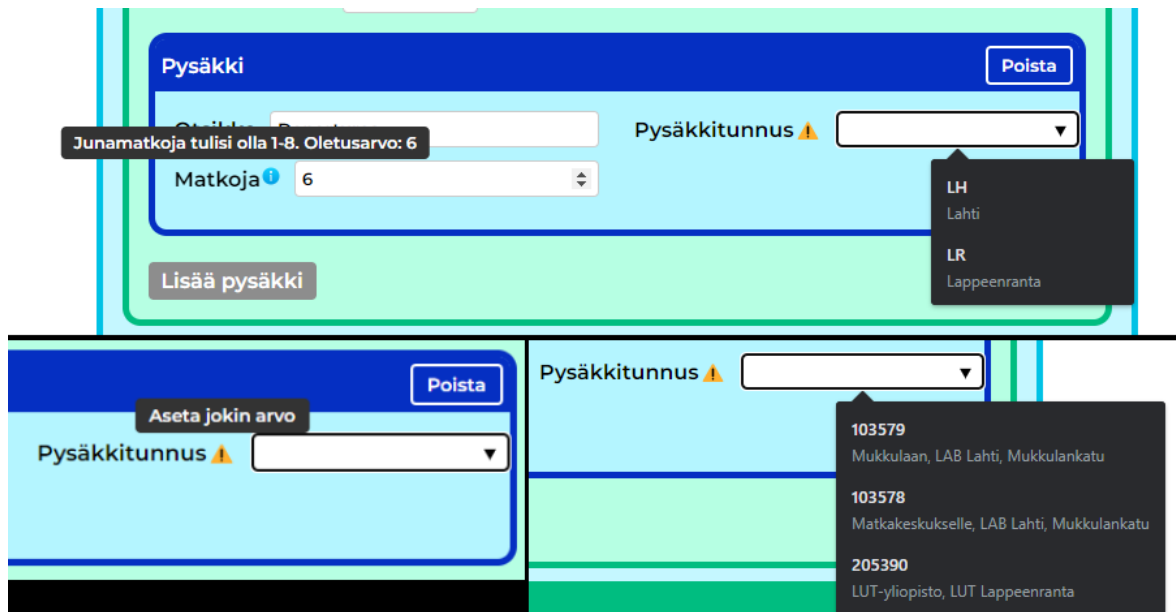
LH

Matkoja

6

Kuva 9. Hallintapaneelin käyttöliittymä Sijainnit-näkymästä

Sekä rakenteellisen että visuaalisen informaation eheyden säilyttämisen vuoksi luotiin visuaalisia vihjeitä ja ohjeistusviestejä suoraan käyttöliittymään. Vihjeet ja ohjeet muuntuvat kontekstin, kuten linja-auto- tai junatietojen mukaan. Näitä apuohjeita vaadituista arvoista ja sopivista arvojen rajoista esitellään kuvassa 10.



Kuva 10. Kuvakollaasi käyttöliittymän eri dynaamisista ohjeistuksista

Toteutusvaiheessa rajapintojen tarjoajilta haettiin valmiiksi asiakkaalle olennaisimmat pysäkkitunnukset. Järjestelmään voi lisätä manuaalisesti DigiTrafficin ja DigiTransitin tarjoamia pysäkkejä ja asemia myös muista kaupungeista. Pysäkkitunnusten noutaminen ulkopuolisista palveluista on kuitenkin ajoittain monimutkaista. Tämän vuoksi asiakkaalle laadittiin erillinen ohjeistus myös muiden pysäkkitietojen hakemiseen ja käyttöönottoon.

4.3 Ohjelmiston käyttöönotto

Ohjelmiston käyttöönotto toteutettiin yhteistyössä asiakkaan IT-palveluiden kanssa. Tauspalvelu asennettiin asiakkaan tarjoamalle virtualisoidulle Ubuntu-palvelimelle. Asennusprosessi sujui mutkattomasti esivalmistetun bash-skriptiautomaation avulla, joka huolehti tarvittavien riippuvuuksien asentamisesta, tietokannan alustamisesta ja systemd-palvelumääritysten tekemisestä.

Käyttöönoton yhteydessä asiakkaan kanssa käytiin läpi tarvittavien rajapinta-avainten hakuprosessi DigiTrafficin, DigiTransitin ja OpenWeatherMapin palveluista. Rajapinta-avaimet tallennettiin järjestelmään ympäristömuuttujina tuotantopalvelimelle. Samassa yhteydessä määriteltiin myös järjestelmän muut keskeiset asetukset, kuten tiedonhakusykli ja varmuuskopioaikataulujen ajastukset.

Asiakas vastasi päätteiden fyysisistä asennuksista ja verkkomäärittelyksistä. Päätteiden IP-osoitteet varattiin kiinteästi MAC-osoitteiden perusteella, mikä oli edellytys järjestelmän sijainnintunnistuksen toiminnalle. Asennusten jälkeen järjestelmän toiminta testattiin yhdessä asiakkaan kanssa.

Järjestelmän pitkäaikaisen vakauden varmistamiseksi palvelin asetettiin käynnistymään automaattisesti uudelleen viikonloppuisin. Tämä ennakkoiva toimenpide otettiin käyttöön, koska taustapalvelun muistinkäyttö osoitti lievää kasvua pidemmissä käyttöjaksoissa. Säännöllisellä uudelleenkäynnistyksellä pyrittiin varmistamaan järjestelmän resurssien optimaalisen käytön ja ehkäisee mahdollisia muistinhallintaan liittyviä ongelmia.

5 Yhteenveto ja pohdinta

Opinnäytetyössä kehitettiin infojärjestelmän taustapalvelun ja hallintapaneelin ohjelmistot LAB-ammattikorkeakoulun tilapalveluille. Palveluun integroitiin juna- ja linja-autoaikataulut sekä säätiedot DigiTrafficin, DigiTransitin ja OpenWeatherMapin rajapinnoista. Järjestelmän hallinta toteutettiin web-pohjaisella hallintapaneelilla, jonka kautta voidaan hallinnoida sijainteja, asiakaspäätteitä ja aikataulunäkymiä.

Projektin ensisijainen tavoite julkisen liikenteen aikataulujen integroinnista saavutettiin onnistuneesti. Järjestelmä noutaa, prosessoi ja välittää aikataulutiedot asiakaspäätteille yhteisessä muodossa. Hallintapaneelin kautta voidaan määritellä joustavasti eri sijainteihin liittyvät pysäkit ja näytettävät vuorot.

Toinen keskeinen tavoite järjestelmän luotettavuudesta ja tiedon oikeellisuudesta toteutui suunnitellusti. Kaksitasoinen välimuistiratkaisu ja varmuuskopioaikataulut ovat osoittautuneet toimiviksi ratkaisuksiksi. Asiakaspalautteen perusteella järjestelmä on toiminut vakaasti tuotantokäytössä. Järjestelmä selvisi myös useita tunteja kestäneestä DigiTransit-rajapinnan katkoksesta ilman näkyviä häiriöitä asiakaspäätteillä.

Aikataulutietojen todettiin olevan ajantasaisia ja vastaavan rajapinnan tarkkuutta. Palvelun päivityssykli aiheuttaa luonnollisesti viivettä. Taulukossa 2 on vertailtu linja-autovuorojen todellisia ohitusaikoja sarakkeessa Lähtö ja järjestelmästä poistumishetkiä sarakkeessa JuBu. Testeissä havaittiin, että 20 sekunnin päivityssyklillä vuorotiedot poistuvat infojärjestelmästä keskimäärin 39,7 sekuntia sen jälkeen, kun linja-auto on ohittanut pysäkin. Tämän viiveen ennakoivaa korjaamista ei pidetty tarpeellisena, sillä kampuksella sijaitsevat infonäytöt palvelevat tarkoitustaan myös nykyisellä tarkkuudella. Lisäksi DigiTransitin omat aikatauluennusteet ovat vastaavalla tavalla jäljessä reaaliaikaisesta sijainnista.

JuBu	Lähtö	Erotus (s)
18.15.27	18.14.59	28
18.15.27	18.15.08	21
18.20.56	18.20.04	52
18.24.11	18.23.32	37
18.29.11	18.28.44	33
18.34.11	18.33.38	49
18.35.26	18.35.08	20
18.35.54	18.35.14	40
18.37.12	18.36.41	53
18.39.11	18.38.29	40
18.43.58	18.43.15	43
18.48.56	18.47.56	60
	Keskiarvo (s):	39,67

Taulukko 2. Mittauksia lähtöaikojen eroavaisuuksista

Kolmas tavoite skaalautuvuudesta täyttyi sekä teknisen toteutuksen että käytännön kokemusten perusteella. Asiakas on lisännyt järjestelmään uusia päätteitä alkuperäisen käyttöönoton jälkeen, ja suunnitteilla on vielä mahdollisesti lisää asiakaspäätteitä. Järjestelmän arkkitehtuuri tukee laajentumista, sillä julkisten rajapintojen kuormitus kasvaa vain uusien sijaintien, ei yksittäisten asiakaspäätteiden mukaan. Testien perusteella järjestelmään voidaan lisätä myös muiden kaupunkien aikataulutietoja.

Kehitystyön aikana ilmeni muutamia haasteita, kuten asiakaspäätteiden päivitysten ajoittainen viivästyminen ja välimuistin hallintaan liittyvät kysymykset. Nämä ratkaistiin lisäämällä taustapalvelun vastauksiin päivitystarpeen ilmaisun ja erillinen välimuistin hallintakomponentti. Ratkaisut osoittautuivat toimiviksi.

Ohjelmistoarkkitehtuurissa vastualueiden selkeämpään erotteluun olisi voitu kiinnittää enemmän huomiota. Projektin loppupuolella tehdyt muutokset heijastuivat usein moniin taustapalvelun osiin, mikä teki muutosten hallinnasta haastavampaa. Vähemmän sidoksissa oleva rakenne olisi voinut tehdä taustapalvelun kehittämisestä ja ylläpidosta suoraviivaisempaa.

Myös testiautomaation puute alkoi näkyä selkeämmin projektin kasvaessa. Vaikka erilaiset npm-skriptit auttoivat kehitystyön sujuvuudessa, yksinkertaiset API-tason integraatiotestit keskeisille reiteille olisivat voineet nopeuttaa regressiotestausta muutosten yhteydessä.

Jatkokehityksen kannalta järjestelmään jäi kiinnostava potentiaali yleistyyppisenä tietojärjestelmänä. Arkkitehtuuriin sisällytetty pohja infotyyppisille näkymille tarjoaa

mahdollisuuksia laajentaa järjestelmän käyttötarkoitusta. Tämän ominaisuuden hyödyntäminen ja jatkokehitys olisi mielenkiintoinen tutkimuskohde.

Kokonaisuutena opinnäytetyö saavutti sille asetetut tavoitteet. Infojärjestelmä palvelee päivittäin kampuksen käyttäjiä tarjoten ajantasaista tietoa julkisen liikenteen aikatauluista. Valitut teknologiat ja arkkitehtuuriratkaisut ovat tukeneet järjestelmän luotettavaa toimintaa.

Lähteet

- Ambler, S. 2023. Mapping Objects to Relational Databases: O/R Mapping. Viitattu 5.11.2024. Saatavissa <https://agiledata.org/essays/mappingobjects.html>
- Borghi, C. 2023. Inside Node.js: Exploring Asynchronous I/O. Viitattu 31.10.2024. Saatavissa <https://dev.to/ocodista/inside-nodejs-exploring-asynchronous-io-4bg1>
- Buoncristiani, G. 2016. JS Prototype chain mechanism. Viitattu 31.10.2024. Saatavissa <https://giamir.com/js-prototype-chain-mechanism>
- Codecademy 2024. What is REST?. Viitattu 7.11.2024. Saatavissa <https://www.codecademy.com/article/what-is-rest>
- Debian 2024. systemd - Debian Wiki. Viitattu 8.11.2024. Saatavissa <https://wiki.debian.org/systemd>
- Fielding, R. 2000. Architectural Styles and the Design of Network-based Software Architectures. Irvine: University of California.
- Fielding, R. 2008. REST APIs must be hypertext-driven. Blogikirjoitus. Viitattu 6.11.2024. Saatavissa <https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven>
- Fielding, R. & Taylor, R. 2002. Principled Design of the Modern Web Architecture. Teoksessa ACM Transactions on Internet Technology, Vol. 2, No. 2. New York: Association for Computing Machinery, 115-150.
- Github 2024. expressjs/express: Fast, unopinionated, minimalist web framework for node. Viitattu 5.11.2024. Saatavissa <https://github.com/expressjs/express>
- Hallie, L. 2020. JavaScript Visualized: Promises & Async/Await. Viitattu 30.10.2024. Saatavissa <https://medium.com/@lydiahallie/javascript-visualized-promises-async-await-a3f1aad8a943>
- Hanson, J. 2024a. jaredhanson/passport. Viitattu 5.11.2024. Saatavissa <https://github.com/jaredhanson/passport>
- Hanson, J. 2024b. Overview. Viitattu 5.11.2024. Saatavissa <https://www.passportjs.org/docs/>
- Haverbeke, M. 2024. Eloquent JavaScript. 4. painos. San Francisco: No Starch Press.
- Kantor, I. 2021. Prototype methods, objects without __proto__. Viitattu 31.10.2024. Saatavissa <https://id.javascript.info/prototype-methods>

- Krstanović, M. 2023. Understanding the Differences: MVC vs. MVVM. Blogikirjoitus. Viitattu 5.11.2024. Saatavissa <https://brightmarbles.io/blog/differences-between-mvc-and-mvvm/>
- MDN 2024a. JavaScript. Viitattu 30.10.2024. Saatavissa <https://developer.mozilla.org/en-US/docs/Web/JavaScript>
- MDN 2024b. Inheritance and the prototype chain. Viitattu 12.11.2024. Saatavissa [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance and the prototype chain](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain)
- MDN 2024c. Introducing asynchronous JavaScript. Viitattu 30.10.2024. Saatavissa <https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Introducing>
- Microsoft 2024a. TypeScript for JavaScript Programmers. Viitattu 30.10.2024. Saatavissa <https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html>
- Microsoft 2024b. Object Types. Viitattu 30.10.2024. Saatavissa <https://www.typescriptlang.org/docs/handbook/2/objects.html>
- Microsoft 2024c. Generics. Viitattu 30.10.2024. Saatavissa <https://www.typescriptlang.org/docs/handbook/2/generics.html>
- Microsoft 2024d. Decorators. Viitattu 5.11.2024. Saatavissa <https://www.typescriptlang.org/docs/handbook/decorators.html>
- OpenJS Foundation 2024a. Introduction to Node.js. Viitattu 31.10.2024. Saatavissa <https://nodejs.org/en/learn/getting-started/introduction-to-nodejs>
- OpenJS Foundation 2024b. An introduction to the npm package manager. Viitattu 31.10.2024. Saatavissa <https://nodejs.org/en/learn/getting-started/an-introduction-to-the-npm-package-manager>
- OpenJS Foundation 2024c. Running TypeScript with a runner. Viitattu 5.11.2024. Saatavissa <https://nodejs.org/en/learn/typescript/run>
- OpenJS Foundation 2024d. The Node.js Event Loop. Viitattu 31.10.2024. Saatavissa <https://nodejs.org/en/learn/asynchronous-work/event-loop-timers-and-nexttick>
- OpenJS Foundation 2024e. Using Express middleware. Viitattu 5.11.2024. Saatavissa <https://expressjs.com/en/guide/using-middleware.html>
- OpenJS Foundation 2024f. Express routing. Viitattu 5.11.2024. Saatavissa <https://expressjs.com/en/guide/routing.html>

Poggiali, S. 2024. TypeScript Introduction. Viitattu 30.10.2024. Saatavissa <https://gibbok.github.io/typescript-book/book/typescript-introduction/>

RapidAPI 2022. State of APIs 2022 | Rapid Developer Survey Results. Viitattu 7.11.2024. Saatavissa <https://stateofapis.com/>

Red Hat 2022. What is a Linux server? Viitattu 5.11.2024. Saatavissa <https://www.redhat.com/en/topics/linux/linux-server>

SQLite Consortium 2024a. SQLite Home Page. Viitattu 5.11.2024. Saatavissa <https://www.sqlite.org/>

SQLite Consortium 2024b. SQL Features That SQLite Does Not Implement. Viitattu 5.11.2024. Saatavissa <https://www.sqlite.org/omitted.html>

SQLite Consortium 2024c. Appropriate Uses For SQLite. Viitattu 5.11.2024. Saatavissa <https://www.sqlite.org/whentouse.html>

Stonis, M. 2022. Enterprise Application Patterns using .NET MAUI. 2. painos. Redmond: Microsoft Developer Division, .NET, and Visual Studio product teams.

Stonis, M. 2024. Model-View-ViewModel - .NET. Viitattu 5.11.2024. Saatavissa <https://learn.microsoft.com/en-us/dotnet/architecture/maui/mvvm>

TypeORM 2024a. TypeORM. Viitattu 5.11.2024. Saatavissa <https://typeorm.io/>

TypeORM 2024b. Active Record vs Data Mapper. Viitattu 5.11.2024. Saatavissa <https://typeorm.io/active-record-data-mapper#what-is-the-active-record-pattern>

Vermeersch, R. 2013. Deploying Node.js with systemd. Viitattu 5.11.2024. Saatavissa <https://rocketeer.be/articles/deploying-node-js-with-systemd/>

Vue.js a. Getting Started - vue.js. Viitattu 8.11.2024. Saatavissa <https://012.vuejs.org/guide/>

Vue.js b. Built-in Directives. Viitattu 8.11.2024. Saatavissa <https://vuejs.org/api/built-in-directives.html>

Vue.js c. Introduction. Viitattu 6.11.2024. Saatavissa <https://vuejs.org/guide/introduction.html>

Willison, G. 2017. NodeJS: the event driven, server-side runtime bringing "JavaScript everywhere". Viitattu 31.10.2024. Saatavissa <https://www.linkedin.com/pulse/nodejs-event-driven-server-side-runtime-bringing-grant-willison>