Kokoushuoneiden Varausjärjestelmän Arkkitehtuurisuunnitelma ja Tekninen Toteutusraportti

Tiivistelmä

Tämä dokumentti muodostaa kattavan teknisen määrittelyn ja arkkitehtuurisuunnitelman kokoushuoneiden varausjärjestelmän ("Meeting Room Booking System") toteuttamiseksi. Raportti on laadittu vastaamaan modernin ohjelmistokehityksen vaatimuksia, joissa korostuvat paitsi tekninen suorituskyky ja koodin ylläpidettävyys, myös tekoälyavusteisten kehitystyökalujen integrointi osaksi ohjelmistotuotantoprosessia. Analyysi pohjautuu annettuun ennakkotehtävään, jossa simuloidaan tilannetta, jossa kehittäjä toimii tekoälyn tuottaman koodin arkkitehtina ja laadunvarmistajana. Raportti kattaa liiketoimintalogiikan, tietomallinnuksen, rajapintasuunnittelun (API) sekä laadunvarmistuksen strategiat, syventyen erityisesti aikavälien hallintaan (temporal data management) ja samanaikaisuuden hallintaan (concurrency control) REST-arkkitehtuurissa.

1. Johdanto ja Toimeksiannon Analyysi

Ohjelmistoteollisuus elää parhaillaan yhtä historiansa merkittävintä murrosvaihetta. Generatiivisen tekoälyn (AI) nousu on muuttanut ohjelmistokehittäjän roolia perinteisestä koodin kirjoittajasta kohti arkkitehtuurista suunnittelijaa, koodin katselmoijaa ja järjestelmäintegraattoria. Tämän toimeksiannon ytimessä ei ole ainoastaan toimivan varausjärjestelmän tuottaminen, vaan nimenomaan tämän uudenlaisen työnkulun hallinta, jossa tekoäly toimii "junior-tason parikoodaajana" ja ihminen ottaa vastuun kokonaisuuden eheydestä, tietoturvasta ja skaalautuvuudesta.[1]

Tässä projektissa toteutettava kokoushuoneiden varausjärjestelmä on luonteeltaan MVP-tason (Minimum Viable Product) sovellus, mutta sen tekninen perusta on rakennettava kestämään tuotantokäytön vaatimukset. Vaikka sovelluksen laajuus on rajattu perustoimintoihin – varausten luontiin, perumiseen ja listaukseen – pinnan alla piilee monimutkaisia tietoteknisiä haasteita. Näitä ovat muun muassa aikavälien päällekkäisyyksien matemaattinen todentaminen, tietokantatransaktioiden eristäminen ja HTTP-protokollan semantiikan oikeaoppinen soveltaminen virhetilanteissa.

Raportin rakenne etenee loogisesti liiketoimintatarpeiden tunnistamisesta tekniseen syväanalyysiin. Ensin määritellään käyttäjätarinat ja hyväksymiskriteerit, jotka toimivat kehityksen ohjenuorana. Tämän jälkeen pureudutaan tietomallinnukseen, jossa ratkaistaan temporaalisen datan tallennukseen liittyvät ongelmat SQLite-tietokannassa. Arkkitehtuuriosuudessa perustellaan valittu teknologiapino (Node.js, TypeScript, Express) ja suunnittelumallit (Layered Architecture, Repository Pattern). Lopuksi käsitellään

API-rajapinnan suunnittelu RESTful-periaatteiden mukaisesti sekä tekoälyavusteisen prosessin dokumentointi ja hallinta.

1.1 Toimeksiannon Reunaehdot ja Oletukset

Tehtävänanto asettaa tietyt tekniset ja prosessuaaliset reunaehdot, jotka ohjaavat kaikkia suunnitteluratkaisuja. Keskeisin tekninen rajoite on muistinvaraisen (in-memory) tietokannan käyttö, mikä eliminoi tarpeen raskaalle infrastruktuurille, mutta vaatii erityistä huomiota datan pysyvyyden simuloinnissa sovelluksen elinkaaren aikana.[1]
Koska toimeksiannossa ei ole käytettävissä "asiakasta", jolta voisi tarkentaa vaatimuksia, tämä raportti tekee seuraavat arkkitehtuuriset oletukset, jotka on johdettu alan parhaista käytännöistä:

1. **Aikavyöhykkeettömyys:** Järjestelmä käsittelee kaikkia aikoja UTC-muodossa (Coordinated Universal Time). Tämä on kriittistä, jotta vältytään kesäaikaan (DST) ja aikavyöhykemuunnoksiin liittyviltä virheiltä palvelinpuolella.[2]
2. **Identiteetin hallinta:** Vaikka täyttä autentikaatiojärjestelmää ei vaadita, tietomalliin sisällytetään userId-kenttä, jotta varaukset voidaan sitoa käyttäjiin. Tämä mahdollistaa myöhemmin ominaisuuden, jossa vain varauksen tekijä voi muokata tai poistaa varauksensa.
3. **Resurssien staattisuus:** Huoneet oletetaan toistaiseksi staattisiksi resursseiksi, jotka alustetaan järjestelmän käynnistyessä. Tämä yksinkertaistaa tietokantarakennetta MVP-vaiheessa.

2. Liiketoimintavaatimukset ja Käyttäjäkokemus

Järjestelmän ensisijainen tavoite on ratkaista fyysisten resurssien allokointiongelma ajallisessa ulottuvuudessa. Tämä on klassinen "resource scheduling" -ongelma, jossa resurssi (huone) on niukka hyödyke, ja kilpailevat prosessit (käyttäjät) yrittävät varata sitä eksklusiiviseen käyttöön. Liiketoimintalogiikan ytimessä ovat invariantit, eli säännöt, joiden on oltava totta kaikissa järjestelmän tiloissa. Tärkein näistä on päällekkäisyyden kielto: yhdelläkään ajanhetkellä t samalla huoneella R ei voi olla voimassa kuin korkeintaan yksi varaus B.

2.1 Käyttäjätarinat (User Stories)

Käyttäjätarinat on muotoiltu noudattaen INVEST-kriteeristöä (Independent, Negotiable, Valuable, Estimable, Small, Testable), mikä varmistaa niiden soveltuvuuden ketterään kehitykseen.[4]

US-1: Huoneen Varaaminen

- **Tarina:** "Työntekijänä haluan varata neuvotteluhuoneen tietylle aikavälille, jotta voin järjestää tapaamisen ilman keskeytyksiä tai tilanpuutetta."
- **Liiketoiminta-arvo:** Mahdollistaa organisaation sujuvan toiminnan ja resurssien tehokkaan käytön.
- **Hyväksymiskriteerit:**
    - Varaus onnistuu, mikäli tila on vapaa koko pyydetyn ajan.

    - Varaus hylätään automaattisesti, jos se menee päällekkäin (edes osittain) olemassa olevan varauksen kanssa.
    - Varaus hylätään, jos aloitusaika on menneisyydessä ($t_{start} < t_{now}$).
    - Varaus hylätään, jos lopetusaika on ennen aloitusaikaa tai sama kuin aloitusaika ($t_{end} \le t_{start}$).
    - Onnistuneesta varauksesta palautuu vahvistus, joka sisältää varauksen yksilöllisen tunnisteen (UUID), huoneen tiedot ja aikavälin.[5]

US-2: Varausten Listaus

- **Tarina:** "Työntekijänä haluan nähdä listan tietyn huoneen tulevista varauksista, jotta voin löytää vapaan ajan omalle kokoukselleni."
- **Liiketoiminta-arvo:** Vähentää turhia varausyrityksiä ja auttaa käyttäjää suunnittelemaan ajankäyttöään.
- **Hyväksymiskriteerit:**
    - Järjestelmä palauttaa listan varauksista, jotka on tehty valittuun huoneeseen.
    - Lista on oletuksena järjestetty kronologisesti nousevaan järjestykseen (t_{start} mukaan).
    - Menneet varaukset voidaan suodattaa pois tai sisällyttää valinnan mukaan (oletuksena näytetään tulevat).
    - Vastaus sisältää varaajan tiedot läpinäkyvyyden lisäämiseksi.[1]

US-3: Varauksen Peruutus

- **Tarina:** "Työntekijänä haluan perua tekemäni varauksen, jotta huone vapautuu muiden kollegoiden käyttöön tilaisuuden peruuntuessa."
- **Liiketoiminta-arvo:** Optimoi resurssien käyttöastetta vapauttamalla käyttämättömät resurssit.
- **Hyväksymiskriteerit:**
    - Varaus voidaan poistaa sen yksilöllisellä tunnisteella.
    - Poisto on idempotentti operaatio: jos varaus on jo poistettu, järjestelmä reagoi johdonmukaisesti ilman virhetilaa.
    - Poiston jälkeen kyseinen aikaväli on välittömästi varattavissa uudelleen.[6]

2.2 Virhetilanteiden Liiketoimintalogiikka

Virhetilanteet eivät ole vain teknisiä poikkeuksia, vaan osa liiketoimintaprosessia. Esimerkiksi "Double Booking" -tilanne on liiketoiminnallisesti kriittinen konflikti. Järjestelmän on viestittävä selkeästi, miksi toimenpide epäonnistui. Pelkkä "Error"-ilmoitus ei riitä; käyttäjän on tiedettävä, onko kyseessä tekninen vika, virheellinen syöte vai resurssikonflikti. Tämä ohjaa API:n HTTP-statuskoodien valintaa myöhemmässä vaiheessa.[7]

3. Tietomallinnus ja Temporaalinen Logiikka

Tietokantasuunnittelu on kriittinen vaihe varausjärjestelmän toteutuksessa. Vaikka käytössä on "yksinkertainen" in-memory SQLite-tietokanta, datan eheyden varmistaminen vaatii tarkkaa

suunnittelua, erityisesti aikakäsitteiden osalta. SQLite eroaa monista muista relaatiotietokannoista (kuten PostgreSQL) siinä, ettei sillä ole natiivia DATETIME tai TIMESTAMP -tietotyyppiä, vaan se luottaa ns. tyyppiaffiniteettiin (type affinity).[3]

3.1 Aikojen Tallennusformaatti: ISO8601 vs. Unix Epoch

SQLite mahdollistaa aikojen tallentamisen joko tekstinä (ISO8601), reaalilukuina (Julian day) tai kokonaislukuina (Unix timestamp). Valinta näiden välillä on tehtävä tietoisesti:

1. **TEXT (ISO8601):** Esimerkiksi "2026-01-30T12:00:00Z".
    - *Edut:* Ihmisluettava, debugattava, leksikaalinen järjestys vastaa kronologista järjestystä, tukee aikavyöhykkeitä (Z-suffix).
    - *Haitat:* Vie enemmän tilaa (tavuja), vertailu on merkkijono-operaatio.
2. **INTEGER (Unix Epoch):** Esimerkiksi 1769774400000.
    - *Edut:* Tehokas numeerinen vertailu, pieni tallennustila.
    - *Haitat:* Ei ihmisluettava ilman muunnosta, vaikeampi debugata suoraan kannasta.

Ratkaisu: Tässä järjestelmässä käytetään **TEXT (ISO8601 UTC)** -formaattia. Vaikka numeerinen vertailu on marginaalisesti nopeampaa, ISO8601-formaatin tuoma luettavuus ja virheettömyys (erityisesti tekoälyn generoimaa koodia tarkasteltaessa) painavat vaakakupissa enemmän. SQLite:n sisäänrakennetut funktiot kuten datetime() ja julianday() toimivat saumattomasti ISO8601-merkkijonojen kanssa. Lisäksi JSON-rajapinnat käyttävät luonnostaan ISO-merkkijonoja, joten muunnoskerros (serialization/deserialization overhead) jää pois.[3]

3.2 Tietokantaskeema (Schema)

Tietokantataulujen rakenne suunnitellaan relaatiomallin mukaisesti. Keskeiset entiteetit ovat Room ja Booking.

Taulu: Rooms

Kenttä	Tyyppi	Rajoite	Kuvaus
id	TEXT	PRIMARY KEY	Huoneen yksilöllinen tunniste (esim. "neukkari-1").
name	TEXT	NOT NULL	Huoneen ihmisluettava nimi.
capacity	INTEGER	CHECK (capacity > 0)	Huoneen henkilökapasiteetti.

Taulu: Bookings

Kenttä	Tyyppi	Rajoite	Kuvaus
id	TEXT	PRIMARY KEY	Varauksen UUID.
roomId	TEXT	NOT NULL, FOREIGN KEY	Viittaus Rooms-tauluun.
user	TEXT	NOT NULL	Varaajan tunniste.
startTime	TEXT	NOT NULL	Varauksen alku

			(ISO8601).
endTime	TEXT	NOT NULL	Varauksen loppu (ISO8601).
createdAt	TEXT	DEFAULT CURRENT_TIMESTAMP	Audit-tieto luontihetkestä.

Indeksointi on suorituskyvyn kannalta kriittistä. Bookings-tauluun luodaan yhdistelmäindeksi (roomId, startTime), joka nopeuttaa merkittävästi päällekkäisyystarkistuksia ja varausten listausta.[9]

3.3 Päällekkäisyyksien Tunnistaminen (Overlap Logic)

Yksi yleisimmistä virheistä varausjärjestelmissä on virheellinen päällekkäisyyslogiikka. Usein tarkistetaan naiivisti, onko uusi aloitusaika olemassa olevan varauksen sisällä. Tämä jättää huomiotta tilanteen, jossa uusi varaus peittää kokonaan olemassa olevan varauksen (ns. "engulfing scenario").
Matemaattisesti täydellinen ehto kahden aikavälin $

4. Tekninen Arkkitehtuuri ja Teknologiavalinnat

Sovelluksen arkkitehtuuri on suunniteltu modulaariseksi, testattavaksi ja selkeäksi. Valinnat perustuvat Node.js-ekosysteemin moderneihin standardeihin.

4.1 Teknologiapino (Tech Stack)

- **Runtime: Node.js (LTS).** Node.js:n tapahtumapohjainen (event-driven) ja ei-estävä I/O-malli sopii erinomaisesti korkean suorituskyvyn verkkorajapintoihin.
- **Kieli: TypeScript.** TypeScriptin tuoma staattinen tyypitys on välttämätöntä, kun halutaan varmistaa koodin laatu ja estää undefined-virheet, jotka ovat yleisiä dynaamisissa kielissä. TypeScript toimii myös dokumentaationa: tyyppimäärittelyt kertovat suoraan, millaista dataa järjestelmässä liikkuu.[12]
- **Framework: Express.js.** Express on minimalistinen ja joustava kehys, joka tarjoaa tarvittavat työkalut reititykseen ja middleware-käsittelyyn ilman pakotettua "opinionated"-rakennetta.[13]
- **Database Driver: better-sqlite3.** Toisin kuin monet muut SQLite-ajurit, better-sqlite3 on synkroninen. Tämä saattaa kuulostaa Node.js:n filosofian vastaiselta, mutta SQLite-operaatiot ovat muistinvaraisessa (ja usein myös levykäytössä) niin nopeita, että asynkronisuuden tuoma "overhead" on suurempi hyötyyn nähden. Synkroninen API yksinkertaistaa koodia merkittävästi ja on suorituskyvyltään ylivoimainen.[14]

4.2 Kerrosarkkitehtuuri (Layered Architecture)

Sovellus noudattaa kerrosarkkitehtuuria, joka erottaa vastuut (Separation of Concerns).

1. **Presentation Layer (Controllers):** Vastaa HTTP-pyyntöjen vastaanottamisesta, parametrien parsinnasta ja HTTP-vastausten lähettämisestä. Tämä kerros ei sisällä

liiketoimintalogiikkaa.

2. **Service Layer (Services):** Sovelluksen sydän. Täällä tapahtuu logiikan suoritus, kuten päällekkäisyyksien tarkistus, validointi ja päätökset siitä, hyväksytäänkö varaus. Service-kerros on tietokantariippumaton.
3. **Data Access Layer (Repositories):** Abstrahoi tietokantakutsut. Repository tarjoaa metodit kuten findOverlappingBookings tai createBooking. Tämä eristää SQL-lauseet yhteen paikkaan, mikä helpottaa testausta ja mahdollista tietokannan vaihtoa tulevaisuudessa.[15]
4. **Domain Models / DTOs:** Määrittelee datarakenteet, joita liikutellaan kerrosten välillä. Käytämme DTO (Data Transfer Object) -mallia erottamaan API:n ulkoisen muodon sisäisestä tietokantamuodosta.

4.3 Validointi ja Zod

Tekoälyn tuottama koodi on usein liian luottavaista syötteiden suhteen. Turvallisuuden takaamiseksi käytämme **Zod**-kirjastoa ajonaikaiseen validointiin. Zod mahdollistaa skeemojen määrittelyn, joista voidaan automaattisesti johtaa TypeScript-tyypit.
Esimerkki validointilogiikasta:

- startTime on oltava validi ISO8601-merkkijono.
- endTime on oltava validi ISO8601-merkkijono.
- user ei saa olla tyhjä merkkijono.
- Mukautettu validointi (refinement): new Date(startTime) < new Date(endTime).[16]

5. API-Rajapinnan Suunnittelu (REST)

Rajapinta noudattaa REST (Representational State Transfer) -arkkitehtuurityyliä. Se on resurssikeskeinen, tilaton ja hyödyntää HTTP-protokollan semantiikkaa.

5.1 Resurssien Nimeäminen

Resurssit nimetään monikkomuotoisilla substantiiveilla. URL-poluissa vältetään verbejä (kuten /createBooking), koska HTTP-metodi kertoo toiminnon luonteen.[18]

- /api/v1/bookings - Kokoelma varauksia.
- /api/v1/rooms/:roomId/bookings - Tietyn huoneen varaukset (sisäkkäinen resurssi).

5.2 Päätepisteet ja Toiminnot

Seuraavassa taulukossa on määritelty rajapinnan päätepisteet:

Metodi	Polku	Kuvaus	Onnistunut Vastaus	Virhevastaus
GET	/api/v1/rooms/:id/bookings	Listaa huoneen varaukset.	200 OK (JSON Array)	404 Not Found (Huonetta ei ole)
POST	/api/v1/bookings	Luo uuden varauksen.	201 Created (Luotu resurssi)	400 Bad Request (Validointi), 409 Conflict (Päällekkäisyys)

DELETE	/api/v1/bookings/:id	Poistaa varauksen.	204 No Content	404 Not Found

5.3 Statuskoodit ja Virheenkäsittely

Oikeiden statuskoodien käyttö on REST-rajapinnan käytettävyyden kulmakivi.

- **200 OK:** Pyyntö onnistui, palauttaa dataa.
- **201 Created:** Resurssi luotiin onnistuneesti. Vastaus sisältää luodun resurssin ja Location-otsakkeen.
- **204 No Content:** Operaatio onnistui, mutta palautettavaa dataa ei ole (käytetään DELETE-operaatiossa).[20]
- **400 Bad Request:** Asiakkaan lähettämä data on virheellistä (esim. Zod-validointivirhe, aika menneisyydessä).
- **404 Not Found:** Pyydettyä resurssia (varausta tai huonetta) ei löydy.
- **409 Conflict:** Pyyntö on ristiriidassa palvelimen tilan kanssa. Tätä käytetään nimenomaan päällekkäisten varausten indikoimiseen. Tämä on semanttisesti tarkempi kuin 400, koska syöte on teknisesti oikein, mutta tila estää operaation.[21]

Virhevastaukset palautetaan standardoidussa JSON-muodossa (Problem Details for HTTP APIs, RFC 7807), jotta asiakasohjelma voi käsitellä ne johdonmukaisesti:

JSON

```
{
  "status": 409,
  "error": "Conflict",
  "message": "Valitulle ajankohdalle on jo olemassa varaus.",
  "timestamp": "2026-01-30T10:00:00Z"
}
```

6. Algoritmiikka ja Datan Eheys

Tietojärjestelmän luotettavuus mitataan sen kyvyllä säilyttää datan eheys poikkeustilanteissa ja kuormituksen alla.

6.1 Samanaikaisuuden Hallinta (Concurrency Control)

Vaikka Node.js on yksisäikeinen (single-threaded event loop), tietokantaoperaatiot voivat aiheuttaa kilpailutilanteita (race conditions), jos useampi pyyntö käsitellään samanaikaisesti. Kuvitellaan tilanne, jossa kaksi käyttäjää yrittää varata saman huoneen samalle ajalle millisekuntien erolla:

1. Pyyntö A tarkistaa: "Onko tilaa?" -> Kyllä.
2. Pyyntö B tarkistaa: "Onko tilaa?" -> Kyllä (koska A ei ole vielä kirjoittanut varausta).

3. Pyyntö A kirjoittaa varauksen.
4. Pyyntö B kirjoittaa varauksen.
5. Lopputulos: Kaksi päällekkäistä varausta (Double Booking).

Ratkaisu on suorittaa tarkistus ja kirjoitus **atomisena transaktiona** tai hyödyntää tietokannan rajoitteita. SQLite better-sqlite3 -kirjastolla toimii synkronisesti ja lukitsee tietokantatiedoston kirjoituksen ajaksi, mikä in-memory-moodissa ja yksisäikeisessä Node-ympäristössä eliminoi suurimman osan sovellustason kilpailutilanteista. Kuitenkin, paras käytäntö on kääriä "tarkista-ja-lisää" -logiikka transaktioon (db.transaction()), joka takaa, että operaatiosarja on jakamaton.[24]

Lisäksi voidaan käyttää SQL-tason TRIGGER-komentoa tai CHECK-rajoitetta, joka estää fyysisesti päällekkäisen rivin lisäämisen, toimien viimeisenä lukkona sovelluslogiikan pettäessä.[24]

6.2 Idempotenttius

DELETE-operaation idempotenttius on tärkeää verkon epävarmuuden vuoksi. Jos asiakas lähettää poistopyynnön, mutta ei saa vastausta verkkovirheen vuoksi, hän saattaa yrittää uudelleen. Toisen pyynnön ei pitäisi aiheuttaa virhettä, vaikka resurssi olisi jo poistettu ensimmäisellä kerralla. API:n tulee käsitellä tilanne siten, että lopputila on toivottu (resurssi poistettu), riippumatta siitä, kuinka monta kertaa pyyntö toistetaan.[18]

7. Laadunvarmistus ja Tekoälyavusteinen Prosessi

Toimeksiannon erityispiirre on tekoälyn käyttö "parikoodaajana". Tämä muuttaa laadunvarmistuksen prosessia.

7.1 Tekoälyn Rooli ja Riskit

Tekoäly (kuten GitHub Copilot tai Claude) on erinomainen tuottamaan "boilerplate"-koodia, SQL-kyselyitä ja peruslogiikkaa. Sillä on kuitenkin taipumus "hallusinoida" kirjastojen ominaisuuksia tai tuottaa naiivia koodia, joka ei huomioi reunaehtoja (kuten aikavälien leikkauksia).

Kehittäjän on toimittava portinvartijana:

1. **Kriittinen tarkastelu:** Tekoälyn tuottama SQL-kysely on validoitava. Ymmärtääkö malli BETWEEN-operaattorin inklusiivisuuden?
2. **Tietoturva-auditointi:** Eihän koodi sisällä SQL-injektiohaavoittuvuuksia (esim. merkkijonojen liimaamista suoraan kyselyyn)? better-sqlite3 tukee parametrisoituja kyselyitä (stmt.run(param1, param2)), joita on ehdottomasti käytettävä.
3. **Dokumentaatio:** PROMPTIT.md -tiedostoon kirjataan, miten tekoälyä on ohjattu. Hyvä prompti on iteratiivinen: "Luo funktio varausten tarkistukseen" -> "Korjaa funktio huomioimaan tilanne, jossa uusi varaus peittää vanhan".[1]

7.2 Testausstrategia

Laadunvarmistus perustuu automaattisiin testeihin:

- **Yksikkötestit (Unit Tests):** Testataan liiketoimintalogiikka eristettynä. Esimerkiksi:

tunnistaako isOverlapping-funktio kaikki neljä konfliktityyppiä?

- **Integraatiotestit:** Testataan API:n ja tietokannan yhteistoiminta. Käytetään supertest-kirjastoa HTTP-pyyntöjen simulointiin in-memory-tietokantaa vasten. Tämä on nopeaa ja luotettavaa.[26]

8. Johtopäätökset

Tämä raportti on määritellyt kokoushuoneiden varausjärjestelmän arkkitehtuurin, joka on teknisesti robusti, noudattaa alan standardeja ja huomioi toimeksiannon erityisvaatimukset. Valittu teknologiapino (Node.js + TypeScript + SQLite) tarjoaa ihanteellisen tasapainon suorituskyvyn, kehitysnopeuden ja tyyppiturvallisuuden välillä.
Keskeisin oivallus on, että yksinkertaiseltakin vaikuttava varausjärjestelmä vaatii syvällistä ymmärrystä temporaalisesta datasta ja samanaikaisuuden hallinnasta. Tekoälyavusteisessa kehityksessä ihmisen rooli korostuu nimenomaan näiden vaativien arkkitehtuuristen ja loogisten ongelmien ratkaisijana, tekoälyn toimiessa tehokkaana toteuttajana. Tämä hybridimalli, jossa yhdistyvät koneen nopeus ja ihmisen ymmärrys, edustaa modernin ohjelmistokehityksen kärkeä.

Liite: Tekninen Rajapintakuvaus (OpenAPI-formaatin mukailtu tiivistelmä)

YAML

```
openapi: 3.0.0
info:
  title: Meeting Room Booking API
  version: 1.0.0
paths:
  /bookings:
    post:
      summary: Create a new booking
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              properties:
                roomId:
```

```
          type: string
        user:
          type: string
        startTime:
          type: string
          format: date-time
        endTime:
          type: string
          format: date-time
  responses:
    '201':
      description: Booking created
    '409':
      description: Conflict - Room already booked
```

Lähdeartikkelit

1. Date range set operations - Stack Overflow, avattu tammikuuta 30, 2026, https://stackoverflow.com/questions/18725426/date-range-set-operations
2. Handling Timestamps in SQLite, avattu tammikuuta 30, 2026, https://blog.sqlite.ai/handling-timestamps-in-sqlite
3. User stories – How to write them and examples - Adobe for Business, avattu tammikuuta 30, 2026, https://business.adobe.com/blog/basics/user-stories-overview
4. How to Write User Stories for Backend APIs - Karaleise.com, avattu tammikuuta 30, 2026, https://karaleise.com/how-to-write-user-stories-for-backend-apis/
5. How we built a room-booking system - DWP Digital, avattu tammikuuta 30, 2026, https://dwpdigital.blog.gov.uk/2017/09/07/how-we-built-a-room-booking-system/
6. Best Practices for API Error Handling - Postman Blog, avattu tammikuuta 30, 2026, https://blog.postman.com/best-practices-for-api-error-handling/
7. [sqlite] Best Practice: Storing Dates, avattu tammikuuta 30, 2026, https://sqlite-users.sqlite.narkive.com/af9RSJwH/sqlite-best-practice-storing-dates
8. Effective indexing and querying of overlapping date/time ranges, avattu tammikuuta 30, 2026, https://dba.stackexchange.com/questions/107092/effective-indexing-and-querying-of-overlapping-date-time-ranges
9. pattpjy/express-ts-sqlite-template - GitHub, avattu tammikuuta 30, 2026, https://github.com/pattpjy/express-ts-sqlite-template
10. Express.js + TypeScript Boilerplate for 2024: Backend Development! : r/node - Reddit, avattu tammikuuta 30, 2026, https://www.reddit.com/r/node/comments/1hyipak/expressjs_typescript_boilerplate_for_2024_backend/

11. Understanding Better-SQLite3: The Fastest SQLite Library for Node.js - DEV Community, avattu tammikuuta 30, 2026, https://dev.to/lovestaco/understanding-better-sqlite3-the-fastest-sqlite-library-for-nodejs-4n8
12. SQLite with a Repository Pattern - Dapper Example | Carl Paton - GitHub Pages, avattu tammikuuta 30, 2026, https://carlpaton.github.io/2017/10/sqlite-with-repository-pattern/
13. Validating Request Data in Express.js using Zod and TypeScript: A Comprehensive Guide, avattu tammikuuta 30, 2026, https://dev.to/osalumense/validating-request-data-in-expressjs-using-zod-a-comprehensive-guide-3a0j
14. Validating API Requests with Zod in Express | by Galtzabari - Medium, avattu tammikuuta 30, 2026, https://medium.com/@galtzabari1/validating-api-requests-with-zod-in-express-7df915199182
15. RESTful API Design Guide: Principles & Best Practices - Strapi, avattu tammikuuta 30, 2026, https://strapi.io/blog/restful-api-design-guide-principles-best-practices
16. How to Choose the Right REST API Naming Conventions | Zuplo Learning Center, avattu tammikuuta 30, 2026, https://zuplo.com/learning-center/how-to-choose-the-right-rest-api-naming-conventions
17. Web API Design Best Practices - Azure Architecture Center | Microsoft Learn, avattu tammikuuta 30, 2026, https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design
18. HTTP Status Codes: All 63 explained - including FAQ & Video - Umbraco, avattu tammikuuta 30, 2026, https://umbraco.com/knowledge-base/http-status-codes/
19. How to fix 409 conflict issue in MS Bookings API - Microsoft Learn, avattu tammikuuta 30, 2026, https://learn.microsoft.com/en-gb/answers/questions/5491714/how-to-fix-409-conflict-issue-in-ms-bookings-api
20. Status Codes in the REST API - Developer, Atlassian, avattu tammikuuta 30, 2026, https://developer.atlassian.com/cloud/trello/guides/rest-api/status-codes/
21. How to check for overlapping intervals when entering data in SQLite3? - Stack Overflow, avattu tammikuuta 30, 2026, https://stackoverflow.com/questions/44337526/how-to-check-for-overlapping-intervals-when-entering-data-in-sqlite3
22. Best practices for using SQLite3 + Node.js - javascript - Stack Overflow, avattu tammikuuta 30, 2026, https://stackoverflow.com/questions/18899828/best-practices-for-using-sqlite3-node-js
23. Node — Boost your code coverage with in-memory Database | by Oded Levy | Medium, avattu tammikuuta 30, 2026, https://medium.com/@odedlevy02/node-how-to-add-per-module-integration-tests-b59c31b47183