Utökad Parkeringsapp: Klient-Server-arkitektur med Databaslagring

Mål

Vidareutveckla den tidigare CLI-baserade parkeringsapplikationen till en klient-server-arkitektur med databaslagring.

Komponenter

1. Klient (CLI-applikation)

- Modifiera den befintliga CLI-applikationen för att använda HTTP-kommunikation med servern.
- Ersätt befintliga repositories med HTTP-klientrepositories.

2. Server

- Implementera en Shelf-server för att hantera HTTP-förfrågningar.
- Skapa route-hanterare för CRUD-operationer på alla entitetstyper (Person, Vehicle, ParkingSpace, Parking).
- Implementera server-side repositories med valfri persistent datalagring.

2.5 Förslag på persistent datalagring

- fil - Du kan skapa din helt egen lösning där du läser/skriver till/från fil i t.ex. JSON eller CSV-format. Oironiskt är kanske detta att rekommendera för dig som aldrig arbetat med databaser förut.
- sqlite - för dig som har tidigare erfarenhet med SQL/SQLite. Se paketet sqlite3 på pub.dev. Notera, jag som kursledare har begränsad erfarenhet med att skriva SQL. Jag föredrar att arbeta mer ORM:s om jag har en SQL-databas.
- hive - Min favorit, en av de mest smidigaste och använda lösningarna för lokal datalagring till dart/flutter, men däremot underhålls inte projektet och en del kodexempel saknas i dokumentationen.
- Supabase - Väldigt stort överlapp med firebase som vi kommer använda senare men just Supabase har jag i skrivande stund begränsad erfarenhet. Jag skriver in det som ett alternativ för jag hoppas trixa lite med det själv under kursens gång men kan inte garantera det.
- din preferens.

3. Datamodeller

- Utöka befintliga datamodeller för att stödja serialisering/deserialisering.
- Utöka befintliga datamodeller för att stödja kraven som ditt val av persistent datalagring har.

Detaljerade krav

Klientsidan (CLI-applikation)

- Uppdatera alla repository-klasser för att använda HTTP-anrop:
    - Implementera create, getAll, getById, update och delete-metoder med http-paketet.
    - Använd async/await och Future för alla nätverksoperationer.
- Modifiera CLI-gränssnittet för att hantera asynkrona operationer:
    - Implementera felhantering för nätverksfel och serverfel.
- Implementera JSON-serialisering/deserialisering för alla datamodeller:
    - Lägg till toJson() och fromJson()-metoder till varje modellklass.

Serversidan

- Sätt upp en Shelf-server med lämpliga routes för varje entitetstyp:
    - /persons, /vehicles, /parkingspaces, /parkings
    - För specifika dokument: /persons/, /vehicles/, etc.
- Implementera route-hanterare för CRUD-operationer:
    - GET (alla och efter ID)
    - POST (skapa)
    - PUT (uppdatera)
    - DELETE
- Skapa repositories för varje entitetstyp:
    - Implementera CRUD-operationer som lagrar i ditt val av persistent datalagring
- Sätt upp felhantering och lämpliga HTTP-statuskoder för svar.

Datamodeller

- Implementera toJson() och fromJson()-metoder för alla entitetsklasser för att stödja serialisering.

Implementeringssteg

1. Sätt upp Shelf-servern med grundläggande routing.
2. Implementera route-hanterare på servern och koppla dem till de befintliga repositories från uppgift 1.
3. Uppdatera klientsidans repositories för att använda HTTP-anrop.
4. Utöka modellklasser till att stödja serialisering/deserialisering till/från JSON.
5. Modifiera CLI-gränssnittet för att hantera asynkrona operationer.
6. Implementera felhantering på både klient- och serversidan.
7. Testa hela systemet för att säkerställa korrekt klient-server-kommunikation.
8. När allt fungerar, kan du byta ut dina repositories från uppgift 1 till nya som garanterar persistent datalagring.

Tabell över routes och metoder

Route	HTTP-metod	Beskrivning	Repository-metod
/persons	GET	Hämta alla personer	getAll()
/persons	POST	Skapa ny person	create()
/persons/	GET	Hämta specifik person	getById()
/persons/	PUT	Uppdatera specifik person	update()
/persons/	DELETE	Ta bort specifik person	delete()
/vehicles	GET	Hämta alla fordon	getAll()
/vehicles	POST	Skapa nytt fordon	create()
/vehicles/	GET	Hämta specifikt fordon	getById()
/vehicles/	PUT	Uppdatera specifikt fordon	update()
/vehicles/	DELETE	Ta bort specifikt fordon	delete()
/parkingspaces	GET	Hämta alla parkeringsplatser	getAll()
/parkingspaces	POST	Skapa ny parkeringsplats	create()
/parkingspaces/	GET	Hämta specifik parkeringsplats	getById()
/parkingspaces/	PUT	Uppdatera parkeringsplats	update()
/parkingspaces/	DELETE	Ta bort parkeringsplats	delete()
/parkings	GET	Hämta alla parkeringar	getAll()
/parkings	POST	Skapa ny parkering	create()
/parkings/	GET	Hämta specifik parkering	getById()
/parkings/	PUT	Uppdatera specifik parkering	update()
/parkings/	DELETE	Ta bort specifik parkering	delete()

Uppdaterade krav för repository-klasser

Varje repository-klass (både på klient- och serversidan) bör nu innehålla följande metoder:

- Future create(T item)
- Future<List> getAll()
- Future<T?> getById(string id)
- Future update(string id, T item)
- Future delete(string id)

På klientsidan ska dessa metoder göra HTTP-anrop till servern. På serversidan ska de först använda de lokala datastrukturerna från uppgift 1, och senare bytas ut mot operationer mot din valda datalagring.

Viktiga punkter att tänka på

- Börja med att implementera server och klient med de befintliga repositories från uppgift 1. Detta ger er en fungerande grund att bygga vidare på.
- När grundfunktionaliteten är på plats och testad, kan ni börja integrera persistent datalagring på serversidan.
- Se till att alla metoder i repository-klasserna är asynkrona och returnerar Future.
- Testa kontinuerligt under utvecklingen för att säkerställa att varje steg fungerar innan ni går vidare till nästa.
- Hantera fel på både klient- och serversidan. Använd lämpliga HTTP-statuskoder i serversvaren och visa användbara felmeddelanden i CLI:t.

Länkar

- https://pub.dev/packages/http
- https://pub.dev/packages/shelf
- https://pub.dev/packages/shelf_router

VG-kriterier (gör minst 2)

- Systemet saknar uppenbara begränsningar
- Applikationen har täckande felhantering på både servern och klienten (nätverksfel/felaktiga id:n osv.)
- Shelf-serverns handlers testas med enhetstester i den teststruktur som genereras när server-projektet skapades med ´dart create´.
- Studenten kommunicerar med handledare ett förslag på förbättring till applikationen som godkänns som en vg-kvalificerande förbättring och implementerar sedan denna.
- Studenten undviker att lagra duplicerat data i databasen genom att nyttja någon form av relationer/referenser.
- Studenten undviker duplicerad kod genom smart nyttjande av abstrakta klasser/generics.