Saad Josef

Dokumentation – AirBnb Naturnära & Hundvänlig

Projektbeskrivning

Detta projekt är en **fullstack TypeScript-applikation** inspirerad av AirBnb-konceptet, med fokus på **naturnära och hundvänliga boenden**.
Systemet möjliggör hantering av användare, boenden och bokningar genom en **MERN-stack**-baserad lösning:

- **MongoDB** – NoSQL-databas för datahantering
- **Express.js** – Backendramverk för API-endpoints
- **React + Vite** – Frontendutveckling för snabb rendering och utvecklingsmiljö
- **Node.js** – Servermiljö

Applikationen erbjuder ett **RESTful API** för datakommunikation mellan frontend och backend. Frontend-gränssnittet är byggt med **React**, **TailwindCSS** och **ShadCN UI** och kommunicerar med backend via **HTTP-förfrågningar** med **Axios**.

För prototyp bestämde att ta bort favoriter sidan, upplevelse sidan och startsidan. Det ska istället direkt visa alla boenden när man kommer in på sidan, sedan har man möjlighet att filtrera som vanligt, det kommer inte finnas en söksida utan det kommer uppdateras direkt efter filter på startsidan. Det kommer finnas en profil sida där det finns bokningshistorik med alla bokningar man gjort och man kommer ha möjlighet att visa detaljerad information om bokningen på egen sida, man kommer också kunna logga ut. Detalj sidan är densamma, register och login samma sak, betalningssidan densamma med små css ändringar.

- Teknisk Översikt

Backend

Komponent	Funktion
Node.js	**Kör servern och hanterar backendlogik**
Express.js	**Skapar API-endpoints och middleware**
MongoDB	**Lagrar data för användare, boenden och bokningar**
Mongoose	**ODM (Object Data Modeling) för MongoDB**
JWT (jsonwebtoken)	**Autentisering och auktorisering**
bcrypt	**Kryptering av lösenord**

Saad Josef

TypeScript	**Typkontroll och ökad kodsäkerhet**
dotenv	**Hantering av miljövariabler**
cors	**Säker kommunikation mellan klient och server**
express-async-handler	**Effektiv felhantering i asynkrona funktioner**
nodemon / ts-node	**Automatisk serveromstart vid utveckling**
Postman	**Testning av API-endpoints**

- **Backend Setup – Kommandon**

 # TypeScript & grundkonfiguration

- npx tsc --init
- npm init -y

 # Server & utvecklingsverktyg

- npm i express mongoose dotenv cors
- npm i -D nodemon ts-node @types/express

 # Autentisering & säkerhet

- npm i jsonwebtoken bcrypt bcrypt-ts
- npm i -D @types/jsonwebtoken @types/bcrypt

 # Felhantering

- npm i express-async-handler

Frontend

Komponent	Funktion
Typescript	**Typkonmtroll & ökad kodsäkerhet**
React + Vite	**Frontendramverk & utvecklingsmiljö**
React Router	**Navigering mellan sidor**
TailwindCSS	**CSS-ramverk för snabb UI-styling**
ShadCN	**UI-komponentbibliotek**
Axios	**HTTP-förfrågningar till API**

Redux	**Global state-hantering**
Redux Toolkit & Persist	**Effektiv state management & persistens**
React-leaflet karta	**Integrerar kartor från Leaflet**
React-icons	**Integrerar ikoner**

- **Frontend Setup – Kommandon**

 # Frontend-installation

- npm create vite@latest
- npm i react react-dom react-router-dom

 # Styling & UI

- npm i -D tailwindcss
- npx tailwindcss init -p
- npm i shadcn-ui

 # State management & API

- npm i axios @reduxjs/toolkit react-redux redux-persist

API-Design & Funktionalitet

Boenden (Housing)

Metod	Endpoint	Beskrivning
GET	**/api/housing**	**Lista alla boenden (med filter & query-parametrar)**
GET	**/api/housing/:id**	**Hämta ett specifikt boende**
POST	**/api/housing**	**Lägg till nytt boende *(kräver autentisering)***
PUT/PATCH	**/api/housing/:id**	**Uppdatera befintligt boende *(admin/member)***
DELETE	**/api/housing/:id**	**Ta bort boende *(admin/member)***

Bokningar (Booking)

Metod	Endpoint	Beskrivning
POST	**/api/bookings**	**Skapa en ny bokning**
GET	**/api/bookings**	**Hämta användarens bokningshistorik**

GET	**/api/bookings/:id**	**Hämta specifik bokning**

Automatisk logik:

- Vid bokning av hela datumintervallet göms annonsen.
- Vid delbokning(inte hela datumet som satt) uppdateras de tillgängliga datumen automatiskt.

Användare (User)

Metod	Endpoint	Beskrivning
POST	**/api/users/register**	**Registrera ny användare** ***(returnerar JWT)***
POST	**/api/users/login**	**Logga in användare** ***(returnerar JWT)***
GET	**/api/users**	**Hämta alla användare** ***(endast admin)***
GET	**/api/users/:id**	**Hämta en användare** ***(endast admin)***
PUT	**/api/users/:id/role**	**Uppdatera användarroll**
DELETE	**/api/users/:id**	**Ta bort användare**

Autentisering & Säkerhet

- **JWT (JSON Web Token)** – används för autentisering och rollbaserad åtkomstkontroll.
- **bcrypt** – används för att hasha lösenord innan lagring i databasen.

Middleware:

Fil	Syfte

authMiddleware.ts	**Validerar JWT och användarroller**
errorMiddleware.ts	**Hanterar fel och skickar standardiserade svar till klienten**

Databasmodeller (Mongoose)

- User – innehåller användardata, roll och autentiseringsinformation
- Housing – hanterar boendeinformation, bilder, datum och tillgänglighet
- Booking – hanterar bokningar, datumintervall och referenser till boende/användare

Implementerade Funktioner – Backend

- TypeScript-konfiguration (tsconfig)
- package.json- konfiguration för nodemon
- Filter- och sökfunktion för boenden
- JWT-autentisering och skyddade rutter
- Rollbaserad åtkomst (admin/member)
- CRUD-operationer för användare, boenden och bokningar
- Automatisk hantering av bokningsdatum
- Kryptering av lösenord
- Miljöhantering via .env
- Git-ignore & versionshantering

Planering & Utvecklingsprocess - Backend

1. Installation av dependencies
2. Konfiguration av tsconfig.json, package.json och .env
3. Strukturering av app, server, typer och konstanter
4. Implementering av modeller, kontrollers och routes
5. Autentisering och felhantering
6. Testning av API-endpoints i **Postman**
7. Kodgranskning och dokumentation

Implementerade Funktioner – Frontend

Här är de nyckelfunktioner som byggdes:

- Interaktivt UI: React + Vite skapar en snabb, komponentbaserat gränssnitt som är lätt att återanvända och anpassa.
- Navigation: React Router hanterar sömlösa övergångar mellan sidor som Home, Housing Detail och Bookings.
- Styling och komponenter: TailwindCSS och ShadCN ger responsiv design med knappar, kalendrar och former.
- State management: Redux Toolkit och Persist hanterar autentisering, boendelistor och bokningar effektivt.

- API och kartor: Axios integrerar backend-anrop, medan react-leaflet visar kartor och react-icons förbättrar visuella element.
- Arkitektur: Komponentbaserad struktur med separata mappar för Pages, Components, Store, Util, Assets, Types, Hooks, Layout och Lib säkerställer organiserad kod.

Planering & Utvecklingsprocess - Frontend

Planerade frontend-utvecklingen steg för steg för att säkerställa en skalbar och effektiv process. Här är översikten:

- Byggt interaktivt UI med React: Fokuserat på användarupplevelse genom komponentbaserad design, med hjälp av moderna verktyg för att skapa en responsiv app.
- Steg-för-steg-implementering:
    - Sätt upp projektet: Använd React + Vite för snabb utveckling och byggprocess. Konfigurera TailwindCSS för styling och ShadCN för komponenter som knappar och kalendrar.
    - Navigation och routing: Implementera React Router för att hantera övergångar mellan sidor.
    - State management: Sätt upp Redux Toolkit för att förenkla hantering av appens tillstånd, och Redux Persist för att spara data över sessioner.
    - API-integration: Använd Axios för HTTP-förfrågningar till backend, med interceptors för säkerhet.
    - Kartor och visuella element: Integrera react-leaflet för geografiska data och react-icons för ikoner.
    - Typning och struktur: Använd TypeScript genomgående, och organisera koden i mappar som Pages, Components, Store, Util, Assets, Types, Hooks, Layout och Lib för enkel underhåll.

Denna process hjälpte att arbeta iterativt, med tester och justeringar för bättre prestanda och användbarhet.

Utmaningar

En utmaning jag stötte på var konfigurationen av tsconfig.json-filen. Jag ville använda ESM-import istället för require, men fick varningar om att det inte var korrekt. Lösningen var att lägga till "esModuleInterop: true" i konfigurationen. Eftersom det var första gången jag använde TypeScript i backend, behövde jag också konfigurera de nödvändiga dependencies för TS, vilket krävde extra tid och felsökning.

En annan utmaning jag stötte på var implementeringen av kartan från react-leaflet i frontend. Först fungerade hämtningen och visningen av platser bra, men nästa dag fick jag felmeddelanden om att tjänsten var blockerad, att jag inte använde rätt header och att platsen inte kunde hittas. Efter att ha läst igenom felmeddelandena och tjänstens regler på Nominatim (karttjänsten), lade jag till en User-Agent-header

("MyHousingApp") och en e-postadress i vite.config.ts för att skapa en direkt anslutning till deras URL. Det löste problemet tillfälligt, men efter 1-2 timmar blockerades mitt IP igen eftersom jag gjorde för många förfrågningar varje gång jag öppnade projektet och gick in på ett boende.
Jag insåg då att felet låg i att jag hämtade och renderade data för ofta. Lösningen var att flytta hämtningen till backend: varje gång ett boende skapas ska koordinaterna fetchas och sparas en gång, istället för att göra upprepade förfrågningar i frontend. Sedan hämtar frontend bara den sparade informationen, vilket undviker blockeringar och förbättrar prestandan.

Exempel som bevisar typescript:

Backend:
TypeScript används i projektet för att skapa tydliga och säkra datamodeller, typa API-requestar och strukturen på MongoDB-dokument.

- Interfaces (t.ex. IHousing) definierar hur ett boende ska se ut i databasen.

- Typer som CreateHousingBody och HousingQuery säkerställer att req.body och req.query innehåller rätt data vid API-anrop.

- TypeScript genererar fel tidigt, innan koden körs, om exempelvis ett fält saknas eller har fel datatyp.

- Controllers använder generics i Express (Request<{}, {}, CreateHousingBody>) för att göra servern stabil och förhindra fel från inkommande JSON.

- mongoose.model<IHousing> kopplar ihop MongoDB-modellen med TypeScript-interfacet så att du får IntelliSense och typkontroll även vid databasanrop.

Kort sagt:
TypeScript gör hela API:et mer robust, tydligt och förhindrar data valideringsfel innan de når databasen.

Frontend:

1. Props är typade (HouseProps)
   → Gör att komponenten inte kan få fel data.

2. house kommer från en definierad typ (Housing)
   → Förhindrar buggar som “undefined property” i UI:t.

3. images, coords, boolean-flaggar och number-värden är strikt typade
   → t.ex. house.petFriendly måste vara boolean
   → house.images måste vara en sträng-array

→ house.totalPrice måste vara ett nummer

4. Autocompletion & compile-time-fel
   När man skriver house. i VS Code får du:
   - title
   - description
   - images
   - nearActivities
   - petFriendly
   - osv.

5. Säkra API-svar och dataflöde
   Eftersom backend också är typad matchar datan frontendens Housing-interface automatiskt.

TypeScript har använts för att skapa tydliga och säkra datamodeller i både frontend och backend. I React-komponenterna typas props med interfaces, t.ex. HouseProps, vilket säkerställer att varje komponent får korrekt data. Backendens modeller och API-svar definieras i .d.ts-filer (t.ex. Housing och CreateHousingBody), vilket ger full typkontroll hela vägen från databasen till UI:t. Detta förhindrar typfel, ger bättre autocompletion och gör koden betydligt mer robust och lättare att underhålla.