Greecebnb App for house renting and booking – MongoDB

Advanced Data Management, May 2024

Ruben Rodrigues

Department of Electrical and Computer Engineering

University of Thessaly, Volos

Abstract – This report presents the development of Greecebnb, a web application designed to be similar to the well-known Airbnb, focusing on booking and renting houses for short periods of time. The project was undertaken as part of the Advanced Data Management course to the university of Thessaly (Electrical and Computer Engineering Department), with a particular emphasis on utilizing a NoSQL database. The chosen database was MongoDB, specifically hosted on Atlas, for its flexibility, scalability and ease of use.

This report covers various aspects of the project, including the rationale behind the choice of the application, the selection of MongoDB, configuration details, technologies employed, and the implementation process. It provides insights into the development journey, highlighting the challenges encountered, solutions devised, and the overall architecture of the application.

The report serves as a comprehensive documentation of the Greecebnb project, shedding light on the intricacies of building a web application with a NoSQL database backend. Through this endeavor, fundamental principles of advanced data management, database design, and web development were explored and applied in a practical context.

Table of Contents

Introduction and Features

The Greecebnb project is an elaborate web application designed for house booking and renting, offering an innovative platform to people looking for temporary housing. Greecebnb, which focuses on user interaction and convenience, provides a variety of options to improve the rental experience.

Project Overview

Greecebnb is aimed to simplify the process of identifying and booking rental houses, responding to the various demands of consumers seeking temporary housing options. By combining convenience, accessibility, and usefulness, the platform aims to provide a seamless user experience.

User Registration and Authentication

Users are provided with the ability to register and log in to the Greecebnb platform, facilitating personalized interactions and access to exclusive features. Leveraging authentication APIs from Google and GitHub, the registration process is simplified, enabling users to seamlessly connect their existing accounts with Greecebnb.

House Listing

The basis of Greecebnb is its wide variety of house listings and categories, which are carefully managed to fit the different interests and needs of its customers. When users join in, they are presented with a full database of available rental properties, including detailed descriptions, multimedia content, and filtering options to narrow their search.

Booking

Greecebnb enables users to easily book their selected accommodations. Users may easily select their chosen rental properties and complete the reservation procedure thanks to the straightforward booking functionality.

Favorites

Understanding the importance of user preferences and convenience, Greecebnb has included a favorites management tool. Users can bookmark their preferred rental properties for future reference and to speed up the booking process.

Advertise your Properties

In addition to exploring existing rental listings, Greecebnb allows users to become hosts by creating their own listings, allowing users to rent out their properties inside the Greecebnb platform.

Trip Cancelation and House Deletion

Greecebnb ensures flexibility by providing trip cancellation alternatives and allowing hosts to properly manage their listings. Users can cancel their listings as needed. Hosts, on the other hand, have the ability to delete their rental listings, ensuring that users receive accurate and up-to-date property availability information.

Database Selection

MongoDB – Atlas

The integration of MongoDB Atlas as the core NoSQL database solution for Greecebnb offers a myriad of advantages, placing automation at the forefront of its functionality and reliability.

Provisioning & Configuration:

MongoDB Atlas simplifies the setup process with intuitive steps, guiding users through provisioning and configuration seamlessly. This alleviates concerns about infrastructure setup, particularly for users who may not have extensive experience in database administration.

Patching & Upgrades:

With automatic patching and single-click upgrades, MongoDB Atlas ensures that the database remains up to date with the latest security patches and feature enhancements. This eliminates the need for manual intervention in patch management, minimizing downtime and ensuring optimal performance.

Monitoring & Alerts:

MongoDB Atlas provides instant visibility into database and hardware metrics, enabling proactive monitoring and alerting for potential issues. By staying ahead of performance bottlenecks and potential failures, Greecebnb can maintain optimal user experience and system reliability.

Disaster Recovery:

The fully managed backup service offered by MongoDB Atlas ensures continuous, consistent backups and point-in-time recovery. Custom retention policies further enhance disaster recovery capabilities, providing peace of mind in the event of unforeseen data loss or corruption.

Flexibility & Support:

MongoDB Atlas combines the capabilities of both relational and NoSQL databases, offering a versatile platform for diverse application requirements. The platform is backed by strong community support and optional expert support services, ensuring that Greecebnb receives timely assistance and guidance as needed.

Security:

MongoDB Atlas prioritizes data security with robust access controls, network isolation, encryption of data in transit and at rest, and optional encryption of the underlying filesystem. These security measures ensure that sensitive data within Greecebnb remains protected from unauthorized access and data breaches.

Scalability:

MongoDB Atlas supports seamless scalability with the click of a button, allowing Greecebnb to scale up or out to accommodate growing data volumes and user traffic. Automatic sharding enables horizontal scale-out for databases, ensuring optimal performance and resource utilization without application downtime.

Availability:

Designed for exceptional uptime, MongoDB Atlas offers transparent recovery from instance failures and automatic failover. Data replication across availability zones and self-healing replica sets ensure continuous availability of Greecebnb's database, minimizing the impact of potential disruptions.

Performance:

MongoDB Atlas delivers high throughput and low latency for demanding workloads, eliminating the need for separate caching tiers. Consistent and predictable performance ensures optimal application performance and a superior price-performance ratio compared to traditional database solutions.

Cloudinary:

In addition to MongoDB (Atlas) as the NoSQL database, the app incorporates Cloudinary for efficient storage and management of images associated with rental property listings. Cloudinary offers a comprehensive cloud-based solution for media management, providing Greecebnb with a reliable platform to store, optimize, and deliver images seamlessly.

Database Setup

To create and set up the MongoDB database for Greecebnb, we utilize Prisma Object-Relational Mapping (ORM) to streamline the connection and interaction between the code and the database. This eliminates the need to manually write SQL commands, simplifying the development process.

Steps

Initialize Prisma:

Open the terminal and execute the following command to initialize Prisma within your project directory:

"npx prisma init"

This command creates a new Prisma folder within your project directory.

Inside the newly created Prisma folder, you'll find a schema.prisma file. Edit this file and configure the datasource for MongoDB:

```
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "mongodb"
  url      = env("DATABASE_URL")
}
```

Create MongoDB Database:

Open the MongoDB web application and sign in or register for an account if you haven't already. Once logged in, create a new database by following the instructions provided in the MongoDB dashboard. Get Connection String: After creating the database, MongoDB will provide you with a connection string. This string includes the necessary information to connect to your MongoDB database, such as the hostname, username, password, and database name.

```
DATABASE_URL="mongodb+srv://ruben:ruben@cluster0.tfqq107.mongodb.net/test
"
```

Designing the DB model and implementing it in the NoSQL DB system

After setting up the database configuration with Prisma and MongoDB Atlas, the next step in developing is to define the data models that will represent the various entities within the application. In the provided Prisma schema, four main models are defined: User, Account, Listing, and Reservation. Let's delve into each model and discuss their significance within the context of the Greecebnb application:

```
model User {
  id              String @id @default(auto()) @map("_id") @db.ObjectId
  name            String?
  email           String?   @unique
  emailVerified   DateTime?
  image           String?
  hashedPassword  String?
  createdAt       DateTime @default(now())
  updatedAt       DateTime @updatedAt
  favoriteIds     String[] @db.ObjectId

  accounts Account[]
  listings Listing[]
  reservations Reservation[]
}

model Account {
  id String @id @default(auto()) @map("_id") @db.ObjectId
  userId             String   @db.ObjectId
  type               String
  provider           String
  providerAccountId  String
  refresh_token      String?  @db.String
  access_token       String?  @db.String
  expires_at         Int?
  token_type         String?
  scope              String?
  id_token           String?  @db.String
  session_state      String?

  user User @relation(fields: [userId], references: [id], onDelete:
Cascade)

  @@unique([provider, providerAccountId])
}

model Listing {
  id             String @id @default(auto()) @map("_id") @db.ObjectId
  title String
  description String
  imageSrc String
  createdAt DateTime @default(now())
  category  String
  roomCount Int
  bathroomCount Int
  guestCount Int
  locationValue String
  userId String @db.ObjectId
  price Int

  user User @relation(fields: [userId], references: [id], onDelete:
Cascade)
  reservations Reservation[]
}

model Reservation {
  id String @id @default(auto()) @map("_id") @db.ObjectId
  userId String @db.ObjectId
  listingId String @db.ObjectId
  startDate DateTime
  endDate DateTime
  totalPrice Int
  createdAt DateTime @default(now())

  user User @relation(fields: [userId], references: [id], onDelete:
Cascade)
  listing Listing @relation(fields: [listingId], references: [id],
onDelete: Cascade)
}
```

User:

The User model represents individuals who interact with the Greecebnb platform. It includes attributes such as name, email, image, and hashedPassword for user identification and authentication. Additionally, it tracks user creation and update timestamps (createdAt and updatedAt), as well as user-specific data such as favoriteIds for storing IDs of listings marked as favorites. The model establishes relationships with other entities such as Account, Listing, and Reservation, indicating associations between users and their accounts, listings they own, and reservations they've made.

Account:

The Account model captures information about user accounts linked to external authentication providers such as Google or GitHub. Attributes include account type, provider details, and access tokens for authentication purposes. The model maintains a relationship with the User model to associate accounts with specific users, enabling seamless authentication and user management within the platform.

Listing:

The Listing model represents individual properties available for rental within the Greecebnb platform. Attributes include details such as title, description, imageSrc, price, and location, providing comprehensive information about each property. Additionally, the model tracks metadata such as creation timestamps (createdAt) and associations with the User model to identify the owner of each listing. This model is crucial for facilitating property discovery, booking, and management.

Reservation:

The Reservation model captures data related to user bookings for specific listings within the Greecebnb platform. Attributes include reservation details such as start and end dates, total price, and creation timestamp (createdAt). The model establishes relationships with the User and Listing models to associate reservations with the respective users and properties involved. This model enables efficient tracking of booking transactions and facilitates communication between users and property owners.

Load the Models

To load the created models into the database simply run the following command on the terminal:

“npx prisma db push”

Loading the data into the DB

To facilitate database operations, including user authentication and authorization, Greecebnb leverages a set of libraries provided by NextAuth.js along with additional dependencies.

Below are the libraries used along with their functionalities:

NextAuth.js:

NextAuth.js is a complete authentication solution for Next.js applications, offering support for various authentication providers such as OAuth, JWT, email/password, and more. It simplifies the implementation of authentication flows and provides a unified interface for managing user sessions, authentication providers, and access control.

@prisma/client:

@prisma/client is the Prisma Client library, which serves as the database access toolkit for Prisma-based applications. It provides a type-safe and auto-generated query builder that enables seamless interaction with the MongoDB database. Prisma Client abstracts away the complexities of database operations, allowing developers to perform ***CRUD*** (Create, Read, Update, Delete) operations with ease.

@next-auth/prisma-adapter:

@next-auth/prisma-adapter is a Prisma adapter specifically designed for use with NextAuth.js. It integrates NextAuth.js with Prisma-based databases, allowing seamless authentication and session management within Next.js applications. The adapter handles user authentication, session storage, and access token management, providing a robust foundation for building secure authentication flows.

bcrypt:

bcrypt is a library used for password hashing and encryption. It provides functions for securely hashing passwords before storing them in the database, enhancing security by protecting user credentials against unauthorized access or data breaches.

To install this simply run in the terminal the following commands and create a new folder:

“next-auth @prisma/client @next-auth/prisma-adapter”

“npm install bycrpt”

```
import { PrismaClient } from "@prisma/client"

declare global {
  var prisma: PrismaClient | undefined
}

const client = globalThis.prisma || new PrismaClient()
if (process.env.NODE_ENV !== "production") globalThis.prisma = client

export default client
```

Code

Register

```
import { NextResponse } from "next/server";
import bcrypt from "bcrypt";

import prisma from "@/app/libs/prismadb";

export async function POST(
  request: Request,
) {
  const body = await request.json();
  const {
    email,
    name,
    password,
   } = body;

   const hashedPassword = await bcrypt.hash(password, 12);

   const user = await prisma.user.create({
    data: {
      email,
      name,
      hashedPassword,
    }
  });

  return NextResponse.json(user);
}
```

Reservation

```
import { NextResponse } from "next/server";

import prisma from "@/app/libs/prismadb";
import getCurrentUser from "@/app/actions/getCurrentUser";

export async function POST(
  request: Request,
) {
  const currentUser = await getCurrentUser();

  if (!currentUser) {
    return NextResponse.error();
  }

  const body = await request.json();
  const {
    listingId,
    startDate,
    endDate,
    totalPrice
   } = body;

   if (!listingId || !startDate || !endDate || !totalPrice) {
    return NextResponse.error();
  }

  const listingAndReservation = await prisma.listing.update({
    where: {
      id: listingId
    },
    data: {
      reservations: {
        create: {
          userId: currentUser.id,
          startDate,
          endDate,
          totalPrice,
        }
      }
    }
  });

  return NextResponse.json(listingAndReservation);
}
```

Reservation ID

```
import { NextResponse } from "next/server";

import getCurrentUser from "@/app/actions/getCurrentUser";
import prisma from "@/app/libs/prismadb";

interface IParams {
  reservationId?: string;
}

export async function DELETE(
  request: Request,
  { params }: { params: IParams }
) {
  const currentUser = await getCurrentUser();

  if (!currentUser) {
    return NextResponse.error();
  }

  const { reservationId } = params;

  if (!reservationId || typeof reservationId !== 'string') {
    throw new Error('Invalid ID');
  }

  const reservation = await prisma.reservation.deleteMany({
    where: {
      id: reservationId,
      OR: [
        { userId: currentUser.id },
        { listing: { userId: currentUser.id } }
      ]
    }
  });

  return NextResponse.json(reservation);
}
```

Listings

```
import { NextResponse } from "next/server";

import prisma from "@/app/libs/prismadb";
import getCurrentUser from "@/app/actions/getCurrentUser";

export async function POST(
  request: Request,
) {
  const currentUser = await getCurrentUser();

  if (!currentUser) {
    return NextResponse.error();
  }

  const body = await request.json();
  const {
    title,
    description,
    imageSrc,
    category,
    roomCount,
    bathroomCount,
    guestCount,
    location,
    price,
   } = body;

  Object.keys(body).forEach((value: any) => {
    if (!body[value]) {
      NextResponse.error();
    }
  });

  const listing = await prisma.listing.create({
    data: {
      title,
      description,
      imageSrc,
      category,
      roomCount,
      bathroomCount,
      guestCount,
      locationValue: location.value,
      price: parseInt(price, 10),
      userId: currentUser.id
    }
  });

  return NextResponse.json(listing);
}
```

Favorites

```
import { NextResponse } from "next/server";

import getCurrentUser from "@/app/actions/getCurrentUser";
import prisma from "@/app/libs/prismadb";

interface IParams {
  listingId?: string;
}

export async function POST(
  request: Request,
  { params }: { params: IParams }
) {
  const currentUser = await getCurrentUser();

  if (!currentUser) {
    return NextResponse.error();
  }

  const { listingId } = params;

  if (!listingId || typeof listingId !== 'string') {
    throw new Error('Invalid ID');
  }

  let favoriteIds = [...(currentUser.favoriteIds || [])];

  favoriteIds.push(listingId);

  const user = await prisma.user.update({
    where: {
      id: currentUser.id
    },
    data: {
      favoriteIds
    }
  });

  return NextResponse.json(user);
}

export async function DELETE(
  request: Request,
  { params }: { params: IParams }
) {
  const currentUser = await getCurrentUser();

  if (!currentUser) {
    return NextResponse.error();
  }

  const { listingId } = params;

  if (!listingId || typeof listingId !== 'string') {
    throw new Error('Invalid ID');
  }

  let favoriteIds = [...(currentUser.favoriteIds || [])];

  favoriteIds = favoriteIds.filter((id) => id !== listingId);
  const user = await prisma.user.update({
    where: {
      id: currentUser.id
    },
    data: {
      favoriteIds
    }
  });

  return NextResponse.json(user);
}
```