Workshop on IoT - Parkitect

General

Parkitect is an innovative app that simplifies guest parking management for residential and commercial buildings. Designed to serve tenants, managers, and guests, the app provides an easy way to reserve and manage parking spaces, enhancing the overall experience for everyone involved.

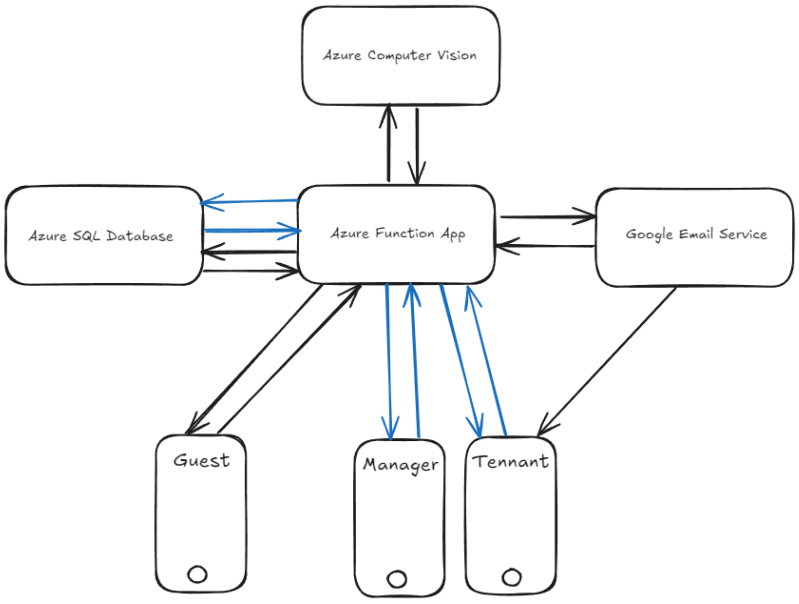
Key Features

1. **For Tenants**: Tenants have the convenience of booking parking spaces for their guests in advance. Each tenant has a personal account where they can schedule and manage reservations, making it easy to ensure a guest has a designated spot upon arrival.
2. **For Managers**: Building managers have full control over parking slot allocations, including the ability to add or remove users, assign parking spaces, and monitor current parking reservations in real time. This feature gives managers a complete view of parking usage, making it easier to oversee and organize the available spaces.
3. **For Guests**: Upon arriving, guests benefit from a seamless entry process. Their license plates are automatically scanned at the parking lot entrance, where the system verifies any existing booking. If a booking is confirmed, the guest gains entry and receives directions to their reserved parking slot. Tenants are then automatically notified via email that their guest has arrived.

Architecture

This project consists of two major components:

1. **The App**: Built with a **Python** backend and a React Native frontend using **TypeScript**.
2. **Cloud Services**: Integrated services include:
    1. **Azure Function App**: Stateless backend functions.
    2. **Azure SQL Database**: Stores the app's data, including tables for bookings, users, buildings, parkings, and parking availability.
    3. **Azure Computer Vision**: Used for license plate recognition functionality.
    4. **Email Service (Google Cloud Console)**: Integrated with Gmail API for sending automated emails to users.



Frontend (React Native with TypeScript)

The frontend is responsible for the user interface (UI) and is developed using React Native with TypeScript for type safety. The app is organized as follows:

- **App Entry Point**: The App.tsx file acts as the main entry point for the mobile app.
- **Configuration Files**: The frontend folder includes environment and configuration files like app.json, tsconfig.json, and babel.config.js, which manage the Expo app configuration, TypeScript setup, and Babel transpilation settings.
- **Screens**: The screens/ folder contains the individual screens for the app, each representing a different part of the user experience (e.g., booking screens, user profiles).

Backend (Python with Azure Function App)

The backend is stateless and deployed using Azure Function App, designed to handle HTTP requests triggered by the frontend. The key components of the backend include:

- **function_app.py**: This is the main entry point for the serverless backend. It defines various endpoints that handle HTTP requests from the app. Each function in this file corresponds to a unique API endpoint, providing a clear interface between the frontend and the backend.
- **helpers.py**: This file contains utility functions that assist the backend processes. These might include data formatting, validation, or other reusable logic shared across the backend functions.

- **booking_managment.py**: This file handles all booking-related logic, such as creating, updating, or deleting booking records. It directly interacts with the SQL database, managing data related to the bookings table and other associated records. The functions in this file are triggered by the function_app.py endpoints.
- **Email Integration**: Automated emails are sent via the Gmail API through the Google Cloud Console. Functions within the backend can trigger emails, such as notifying tenants when their guests arrive. The email service is integrated into the backend functions and specifically used to send notifications to tenants when a guest's car is recognized by the license plate recognition system. This ensures tenants are informed in real-time of their guest's arrival.
- **Database Integration**: The backend interacts with the Azure SQL Database. This database stores data across five key tables:
    - **bookings**: Stores booking information.
    - **users**: Holds user data.
    - **buildings**: Contains building information.
    - **parkings**: Tracks parking spots.
    - **parking_availability**: Manages parking spot availability.
    - The db/ folder contains configuration files and data manipulation scripts that interact with these SQL tables. These files are invoked by the backend functions to fetch, update, or delete data as required.

Cloud Services

1. **Azure SQL Database**: Used to store all of the app's core information across the five mentioned tables. It provides persistent data storage for users, bookings, and availability of parking spots.

2. **Azure Function App**: Each function is designed to be stateless and responds to HTTP requests from the frontend. The functions are lightweight and optimized for quick response times, only executing when triggered by user interactions within the app.

3. **Azure Computer Vision**: This service is integrated into the app to enable license plate recognition, which might be used for verifying cars or managing parking availability based on plate numbers.

4. **Google Cloud Email Service**: Integrated to send automated emails (guest arrival notices). Emails are triggered by backend functions and handled via the Gmail API, authenticated through Google Cloud Console.

Flow of Operations:

1. **User Interaction**: A user interacts with the app (the UI), performing actions such as booking a parking spot.
2. **API Call to Backend**: Each user action triggers an HTTP request to an API endpoint defined in the function_app.py file.
3. **Function Execution**: The corresponding backend function processes the request. For example, a booking request triggers a function in booking_managment.py to create or update a booking in the database.
4. **Database Interaction**: The function communicates with the SQL database, interacting with the relevant tables to retrieve or modify data.
5. **Response to Frontend**: Once the function completes, the result (e.g., success message, booking details) is sent back to the frontend, updating the UI based on the action performed.

This architecture leverages **Azure's cloud services** to ensure scalability, while the separation of concerns between frontend and backend makes it easier to manage and develop each component independently. The use of **React Native with TypeScript** ensures a cross-platform mobile app with type-safe code, and the integration of **serverless functions** allows the backend to scale automatically based on demand.