Final Project
Eric Feldman, Robert Maricutu

Report

Overview

Before we begin the report, to access this project please visit:
https://travel436.azurewebsites.net

The goal behind this application is to enable a user to enter some sort of searching criteria, be it a city, country, landmark, and find out about the cheapest hotel related to that search. To achieve this the user is presented with a straight forward home page. The home page contains the following:

- Destination textbox where a user can input their search term.
- A picklist for selecting how many people will be traveling (maximum of 3) with the user.
- A search button for initiating the search.
- Five tiles for some recommended travel destination to give some inspiration.
- A button to view a PDF of some useful travel tips.

Once an input has been entered and the result is returned, the application then takes the user to a second page to view the results. This page contains the following:

- A display window with information about the hotel including the name, address, price, reviews, and the location of the hotel.
- Inside of this window are also two buttons, the first to link to the Booking.com site that the user can actually book the hotel and the other is to return to the home page.
- On the lower portion there is a large embedded Google Maps widget this widget will display the actual location of the hotel's coordinates to the user with a marker.

Navigation through the website is simple and intuitive. The site doesn't offer a lot of complicated options as the goal is to just give someone an idea of how potentially inexpensive it could be to stay overnight at some of these areas.

Services

To achieve the desired functionality this application takes advantage of a number of web services. This section will cover a little bit of how these services are utilized in this program.

Github - Not only is this online code repository a very convenient way to keep track of code changes but it's also a powerful collaboration tool. This platform was further integrated into the development of this application by utilizing the continuous integration features. These features allowed us to link the Github page directly to our Azure app service so that as soon as a push is made to the master repo the application would be built and sent to Azure for automatic deployment.

Azure App Service - This service is a robust and developer friendly way to get a web application hosted. The app service allowed this node.js application to run on an Azure hosted server.

Azure Blob Storage - Blob storage is a convenient way to store objects such as large PDFs. This service is used so that we can host a viewable link to the travel tips document.

Consuming RESTful APIs - The bulk of the complex work is done by these handy APIs. Taking advantage of inexpensive and sometimes free RESTful APIs allows our application to provide a robust and complicated service while not forcing us to recreate the wheel. Strong utilization of Booking.com and Google Maps API were used. Booking APIs allowed for the smart "best-match" search feature and hotel lookup. Google Maps gave us an opportunity to add more depth and context to the search results. A simple map and marker gives the end-user an idea of where this destination they have stumbled upon actually is. Additionally, the Maps features allows the user to virtually explore the area around their potential accommodations. Without taking advantage of these existing services there is no way that we could have produced some of this complicated functionality with the resources available.

Azure Cosmos DB Table - Using a very scalable and dynamic storage allowed us to have quick access to information about our suggested travel destinations. The scalability of this database was seen as a potential large benefit for adding future functionality. Options like data-mining on customer searches for suggestion prediction or even simply large amounts of results being stored for analysis on potential future features.

Notifications - As monitoring is an important part of any web service we wanted to ensure that we as developers were alerted when something went significantly wrong. For 500 errors Azure alert notifications was used. For frequent errors occuring with our API calls the Twilio SMS API was used. In this way we were covered for a range of notifications scenarios.

Design

Figure 1: High-level Design of the application

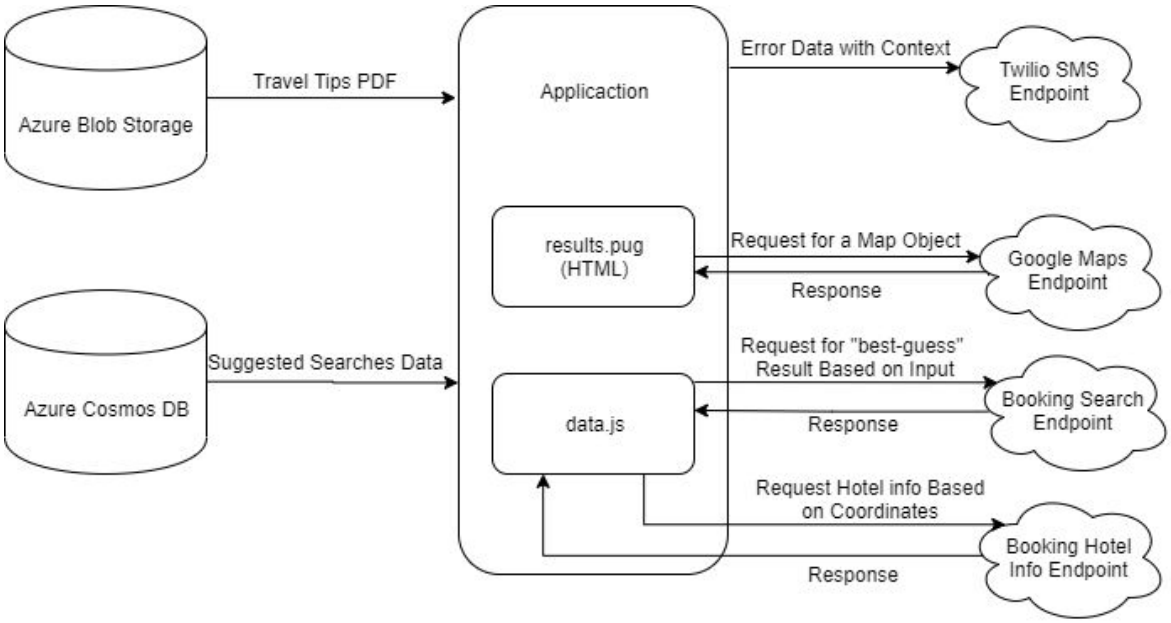
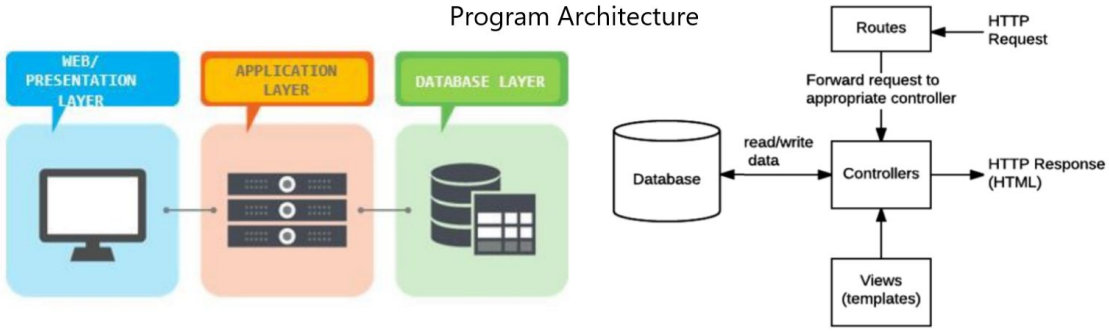


Figure 2: General Architecture of the Application



This is a Node.js application built with the Express framework. The decision to go with this framework was the flexibility in the library that were provided for doing front-end development. Neither of us have any extensive front-end development experience so using simple markup files like pug over HTML helped make some of the programming easier. The express framework helped organize the program separating the classes based on concern into views, routes, and assets. Views were the pug files which actually was the markup that the browser would display. The routes were the controller classes. These classes were responsible for the business logic and interacting with the view elements (with the exception of the Maps API as that was easily put into the markup). The routes classes handled the parsing, error checking, and navigation of the application.

Azure over AWS

The use of Azure was driven by two factors: price, integration. For the price, it was the case that we both had a significant amount of Azure credit left. With all the credits left, we knew that we could afford to use several different types of services on the Azure platform which gave us flexibility in design and peace of mind from a financial standpoint. For integration the node.js project that we began with already had built-in Azure integration which made for a great starting point. Transitioning to AWS wouldn't have been very hard but no work is easier than even a little work. The ease of use of Azure's developer portal however, also allowed for the integration of notifications and build/deploy tools like Kudu.These features were powerful, useful, and best of all very simple (relatively) to implement all inside of the Azure portal. Microsoft's quickstart guides for all of their features were robust with significant documentation which also helped with the ease of integration.

Monitoring

Monitor the system for availability is very important to our customers to ensure they are able to access our service. We have designed our service so that if anything is to fail, we would be notified through azure and Twilio SMS messages. The azure portal enables us to very easily view exactly what is occurring within our service, this includes, CPU time, requests, different HTTP errors, response time, etc. Through the portal, we set up various conditions on which when met, Azure will notify us of an issue. In addition, Twilio was implemented in the business logic to automatically send SMS messages to us specifying exactly when and which API call has failed.

Figure 3: Basic visual monitoring provided by Azure through the portal

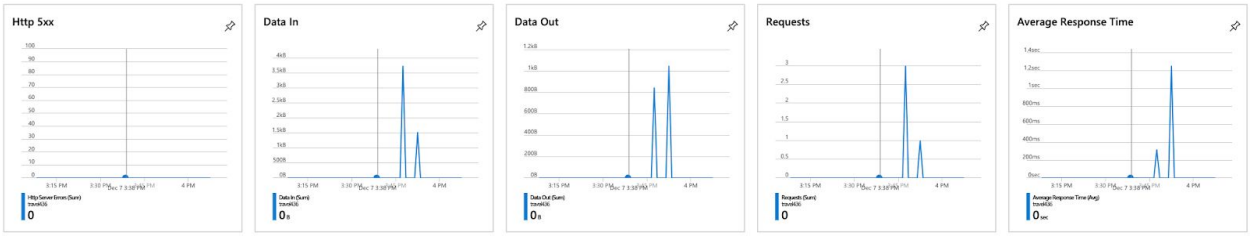
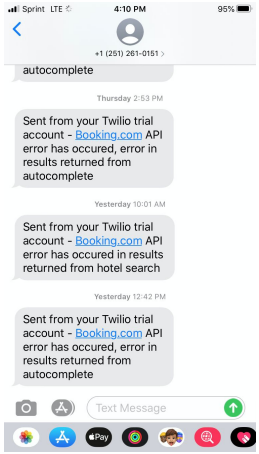


Figure 4: Twilio SMS API alerts received



SLA

This application has strict dependencies on 3 services and 4 API Endpoints. For 100% functionality all of these services must be working. However, the application is designed so that partial functionality is maintained if for instance if the Google Maps marker API goes down the results for the hotel would still show assuming the Booking API calls went through.

The dependencies are as follows:
Azure App Service - 99.95% (free tier)
Azure Cosmos DB - 99.99% (single Azure region)
Azure Blob Storage - 99.9% (standard LRS)
Google Maps API - 99.99%
Booking API - 99.5% (two endpoints are used at this service)
Twilio API - 99.9%

The overall SLA for this application would be
.9995 * .9999 * .999 * .995 * .995 * .999 = **.9875**

SLA is 98.75%
This is assuming no retry logic and that partial functionality is unexceptable.

Scaling Considerations

API Services

The largest bottleneck for this application currently are the restrictions on the Booking APIs. The APIs have a hard limit off 500 calls per month which is very small considering each query runs hits two endpoints. In addition, because of the unreliability of their service a future application would implement robust retry logic further driving up API calls. This can easily be overcome by purchasing a premium API subscription. The Google Maps API also has a similar restriction but is much more generous for their free threshold.

Hardware

This application is fairly lightweight and as such doesn't benefit much from vertical scaling however, using an elastic scaling solution to provide more instances for clients to interact with would be very beneficial. Using Azure App Service there is built-in autoscale which is a great benefit for us as we don't have to do any of the work. Staying within free-tier guidelines has forced us to set max instances very low but with the notification system we have developed we could easily increase those thresholds if we need to. Because monitoring is a part of this application Azure also has tools to allow the system to increase these thresholds automatically without any manual intervention. If this application were to take off this would be utilized.

Storage

From the design of this application we wanted there to be as little state as possible. The storage that is currently being stored is static and so just adding multiple backups for redundancy is enough. However as mentioned in the Services section the cosmos DB was chosen for potential scalability. If this application was really large and we wanted to make the suggestion feature dynamic this table would be the perfect choice as a relational database would be slower and for the use case eventual consistency is more than adequate.