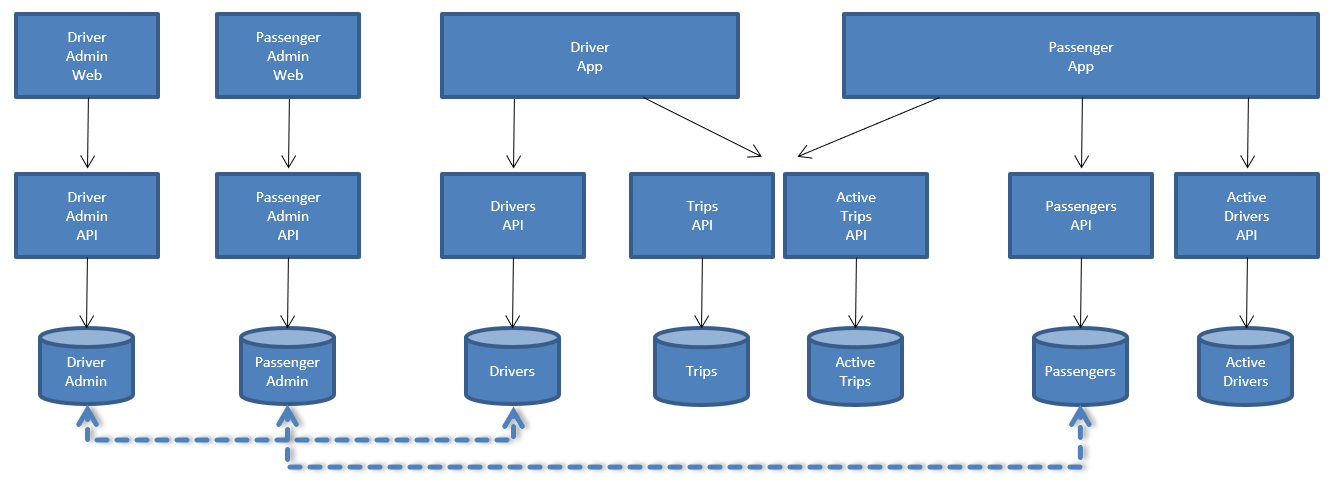
# Serverless Microservices Reference Architecture

This document contains the overall proposed architecture, including the Azure services used, the REST API surface area, how components communicate, and the flow of data through the system from the perspective of the user as well as from automated events.

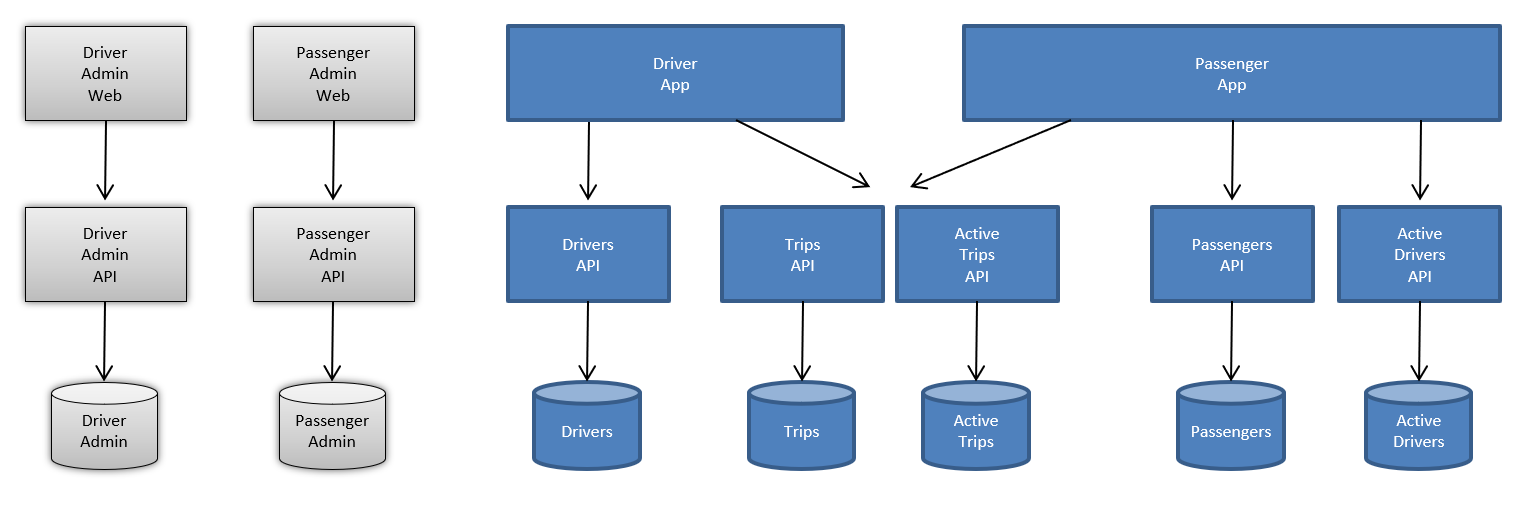
## Big Picture – design without events



The solution design will have a number of web applications and mobile apps (responsive web app for the reference solution):

* Driver admin– a web app for managing driver profiles and related data, used by the internal rideshare corporation (we need an approved name for this company)
* Passenger admin – a web app for managing passenger profiles and related data, used by internal corporation
* Driver app – for drivers to manage their profile, list trip history, manage their active trip
* Passenger app – for passengers to manage their profile, billing, list trip history and manage their active trip
* APIs will include administrative back ends; for the corporate apps
* Those back ends are the master record of all drivers and passenger data outside of AAD and active / transient data
* They are eventually consistent from updates that may take place through mobile app APIs
* APIs also include mobile app APIs that support access to driver, passenger details relevant to the app (eventually consistent two ways with driver and passenger admin)
* APIs for active drivers, trips and trip history also support mobile apps

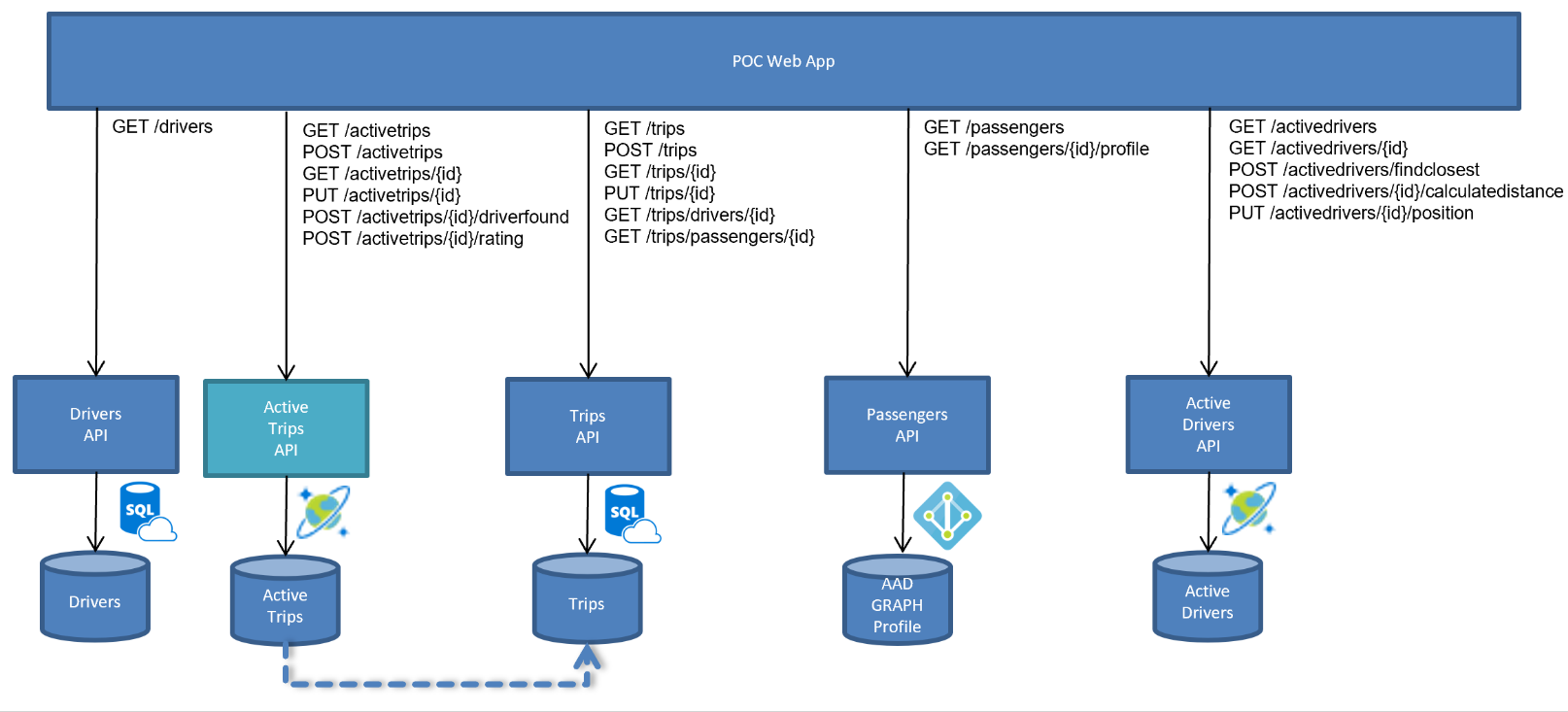
## Proof of concept – parts to simulate



The reference solution work will focus on simulating the driver app and passenger app.

No work will be done to emulate eventual consistency with admin data stores. These are points that can be elaborated on in the documentation as desired.

## Proof of concept – API endpoints



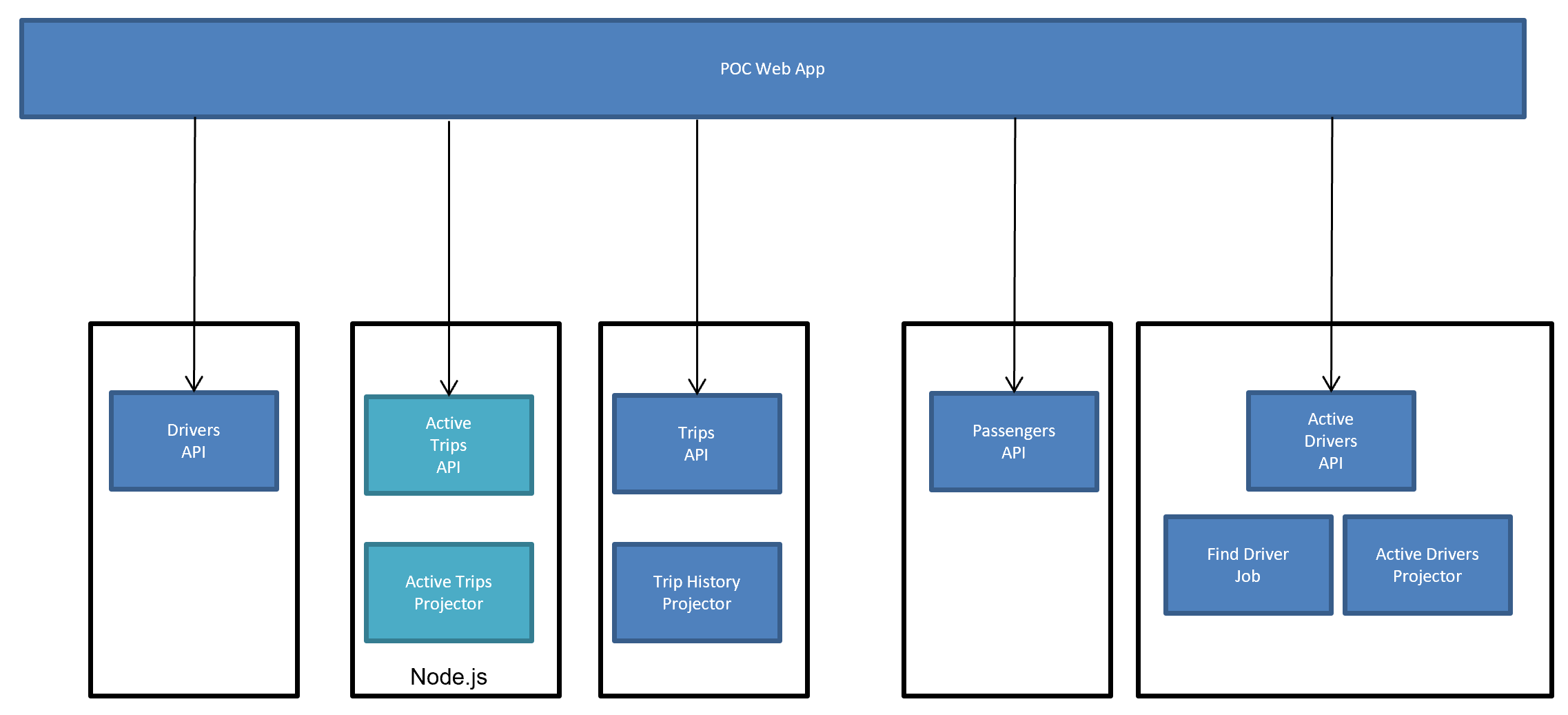
Passenger profiles will be created when a user signs up with Azure AD; Their profiles are stored there.

Graph API will be used to access those profiles, also on login, claims will be returned from the /userinfo endpoint, which is called to gather information for display in the UI, and included in the token (may not be all userinfo, thus graph API calls).

Core relational data, such as drivers and trips, are stored in Azure SQL DB. Transient data, related to trips and active drivers, are stored in Azure Cosmos DB.

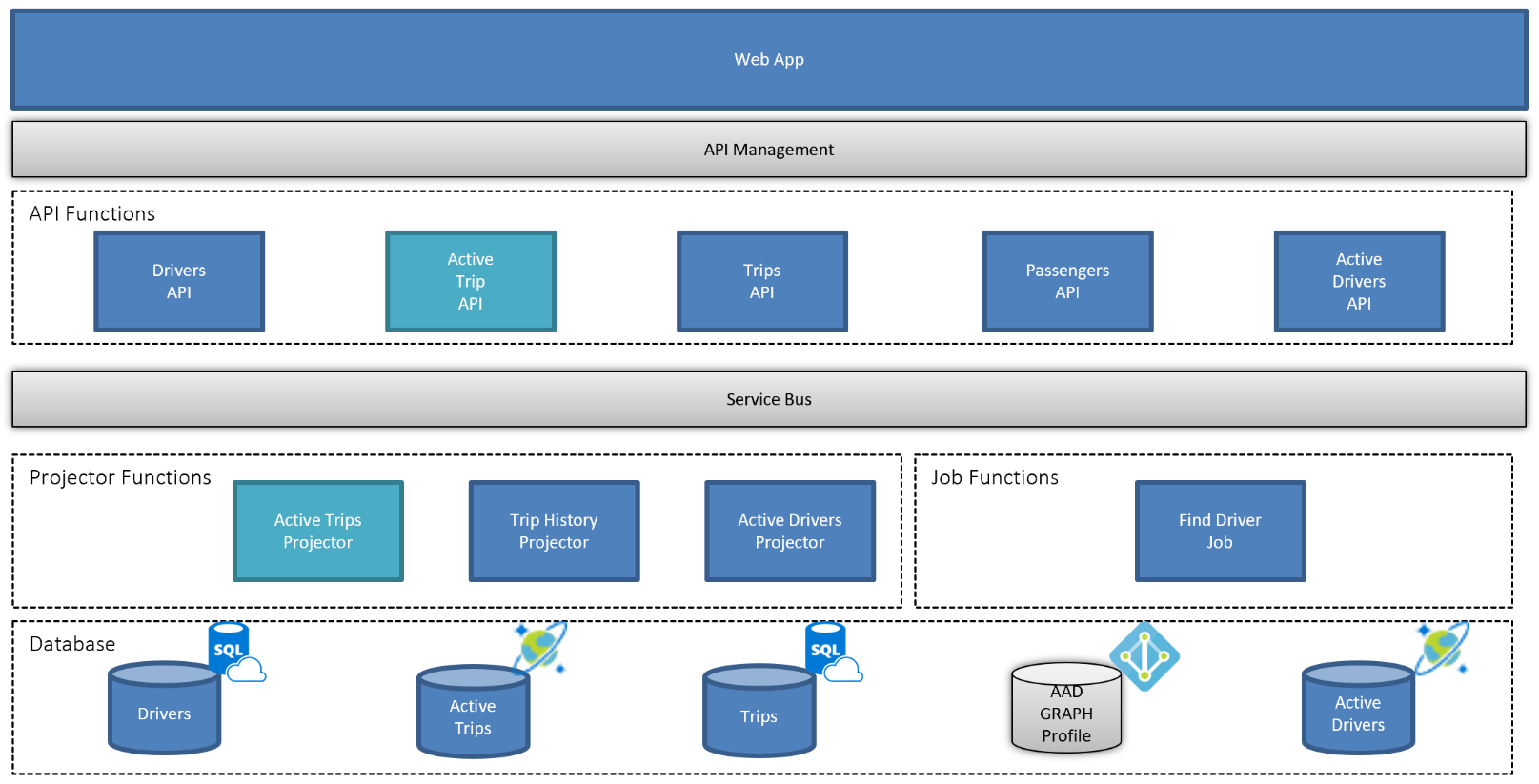
Active Trips can push final trip information to the Trips endpoint with eventual consistency via events. The list of routes per API are listed in the diagram above.

## Function Apps



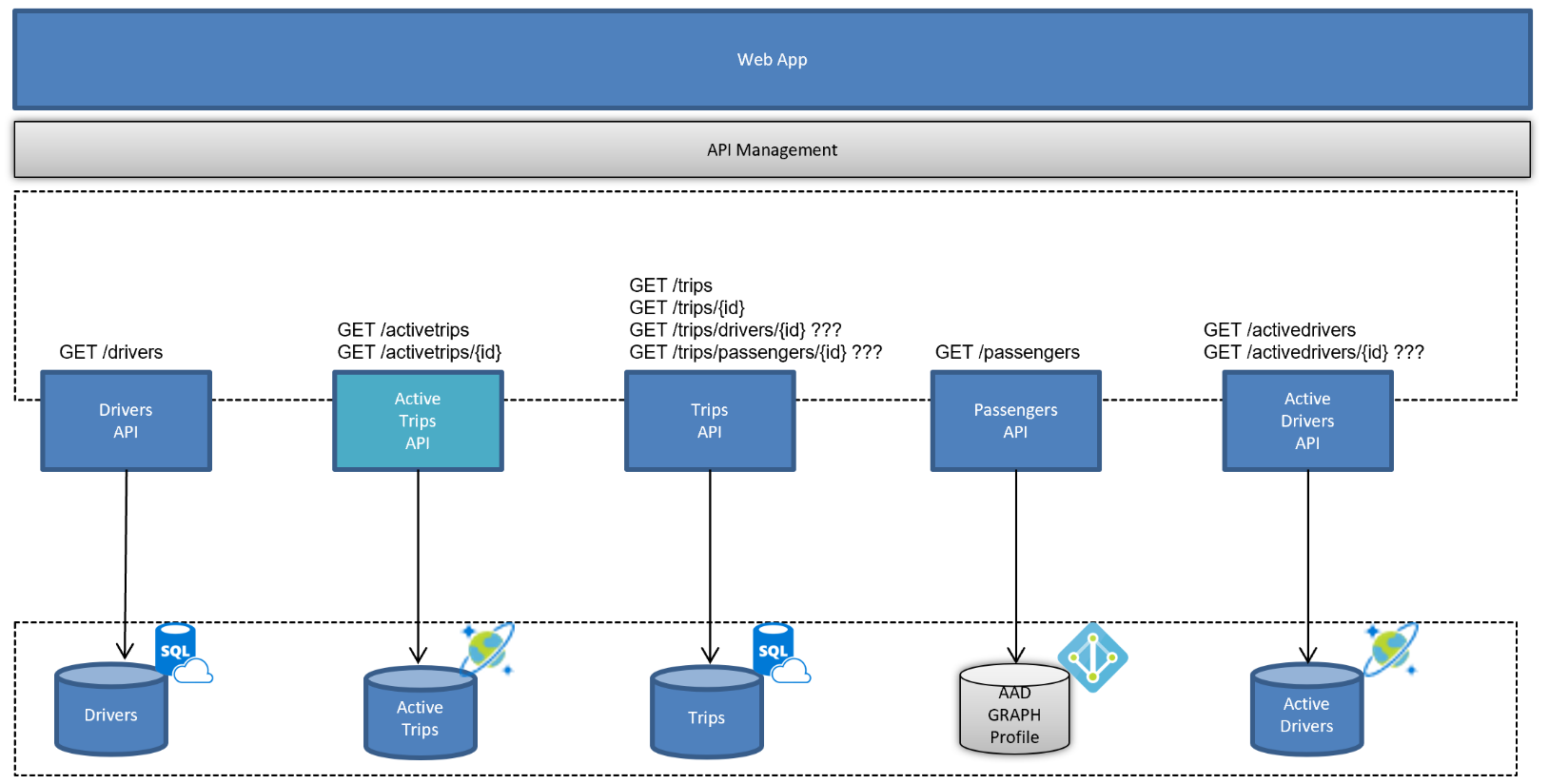
Each Function App can be viewed as a microservice, with the functions within serving as API endpoints and methods. The light blue Function App will be Node.js-based, and the dark blue ones will be .NET-based. There are 5 Function Apps, as indicated by the bounding boxes. Each Function App has functions serving as API endpoints, and some include projector or job functions as well.

## Azure resources



* Web application will call APIs through API Management
* All APIs implemented with Azure Functions
* Reads will go directly to the data source of choice
* Writes will go through a messaging system (Service Bus) to provide a CQRS implementation that can also be used for eventual consistency and to trigger back end activity
* Projectors are functions that pull a message; write the “event” to the database target
* In some cases, the same event may be written to two targets thus multiple consumers
* Jobs are for back end functions that will produce a result, possibly write a message to trigger follow on action such as updating the UI with a SignalR event (through SignalR Service) when a driver is found

## GET API requests

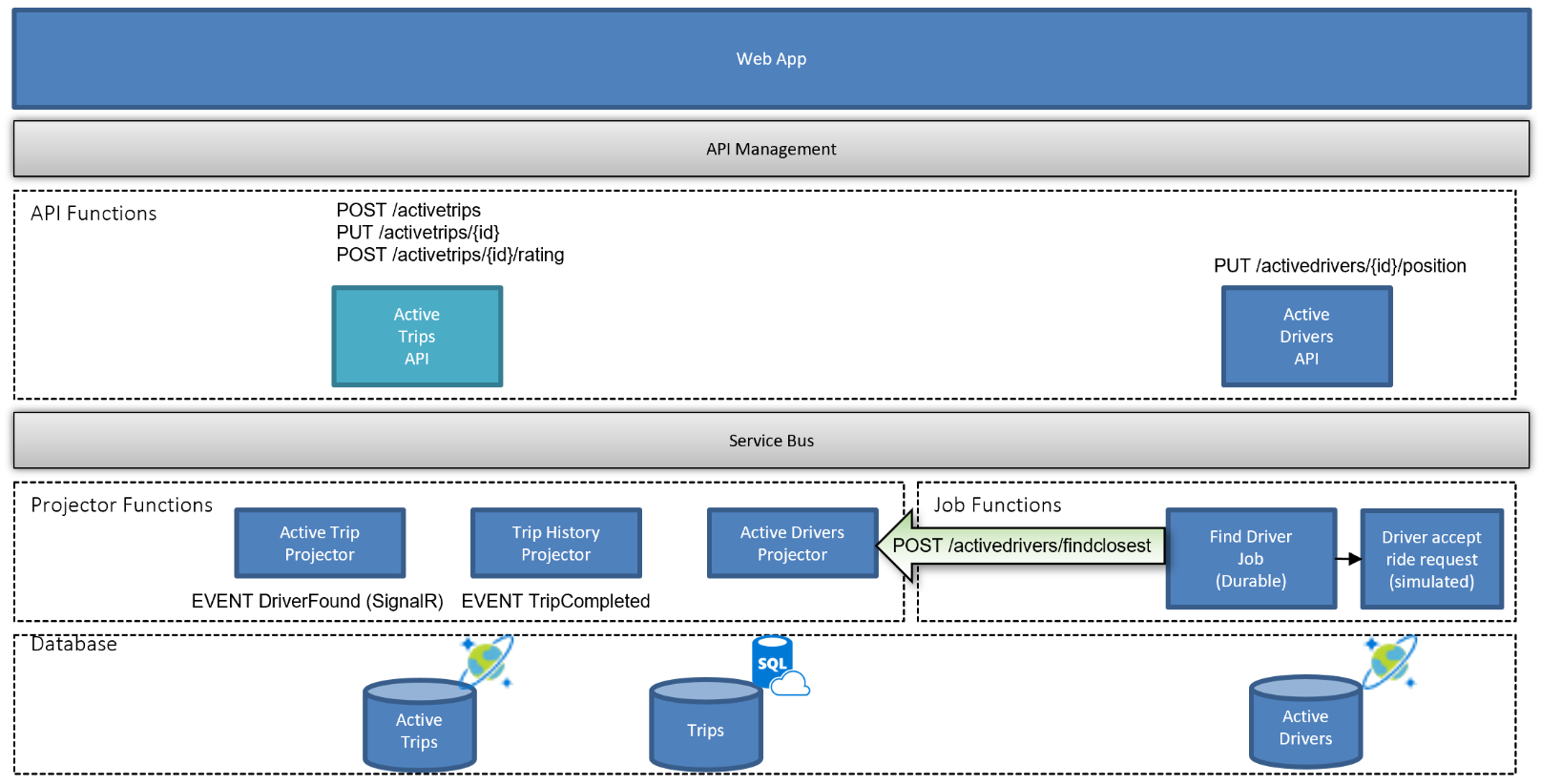


These are the GET requests which call the related data target directly.

/activetrips endpoints served by node.js-based Function App. This is to show multi-language support by Function Apps. The rationale for the scenario is that the client-side dev team who is maintaining the web app also maintains those endpoints. They have a different release cycle and it is more “natural” for node.js developers to use Cosmos DB than Azure SQL.

/activedrivers, /drivers, /trips used by .NET-based Function App.

## POST / PUT API requests



Writes will send a command to the API; which then creates a message/event to Service Bus. The message is used to make the back end eventually consistent for its respective service. In some cases, functions may write new events to trigger a workflow, such as Jobs sending an event that a driver is found – this could be an API call that raises message/event that triggers a SignalR action.

The /activedrivers/{id}/position endpoint will be called by a Function that randomly sets the driver location.

## Overall flow – page details

The web application(s) will contain the following pages:

### 1. Drivers Page

Simulates administering and managing drivers. Calls GET /drivers to show list of drivers from the SQL database. Calls GET /activedrivers to display active drivers (those who are either on a trip or available for a new trip) from Cosmos DB.

### 2. Passengers Page

Simulates managing passenger profiles. Gets list of passenger profiles and can see details about the passenger.

GET /passengers

GET /passengers/{id}/profile

Here’s the proposed workflow for creating a new passenger account (the user would be doing these steps):

Passenger Account Creation

- User clicks link to signup

- Redirect to AAD B2C for signup page, can include custom attributes

- User goes through the confirmation process, email etc. (probably required workflow so real accounts)

- Users stored in AAD

- User now logs in to site, gets AAD token

- Also gets profile info at the /userinfo endpoint (part of OIDC workflow)

- So far no back end on our side, user profile can be shown from the token / user info response claims

- As rides are created, ride history is created in Azure SQL Database, associated to the user id that came out of AAD (their sub)

**GET /passengers** returns list of profiles from AAD graph unless there is a way to activate/inactivate passenger profiles so that the user is still in AAD, but only a true passenger if they are active in the SQL database.

**GET /passenger/{id}/ridehistory**

Ride history would apply to either perspective (passenger or driver). There should be one truth about the ride, and there shouldn’t be much a difference in how either the passenger or driver sees the history. The driver would see their commission from the ride and would have the ability to rate the passenger, and the passenger would see what they were charged, have the ability to add a tip, and also to rate the driver.

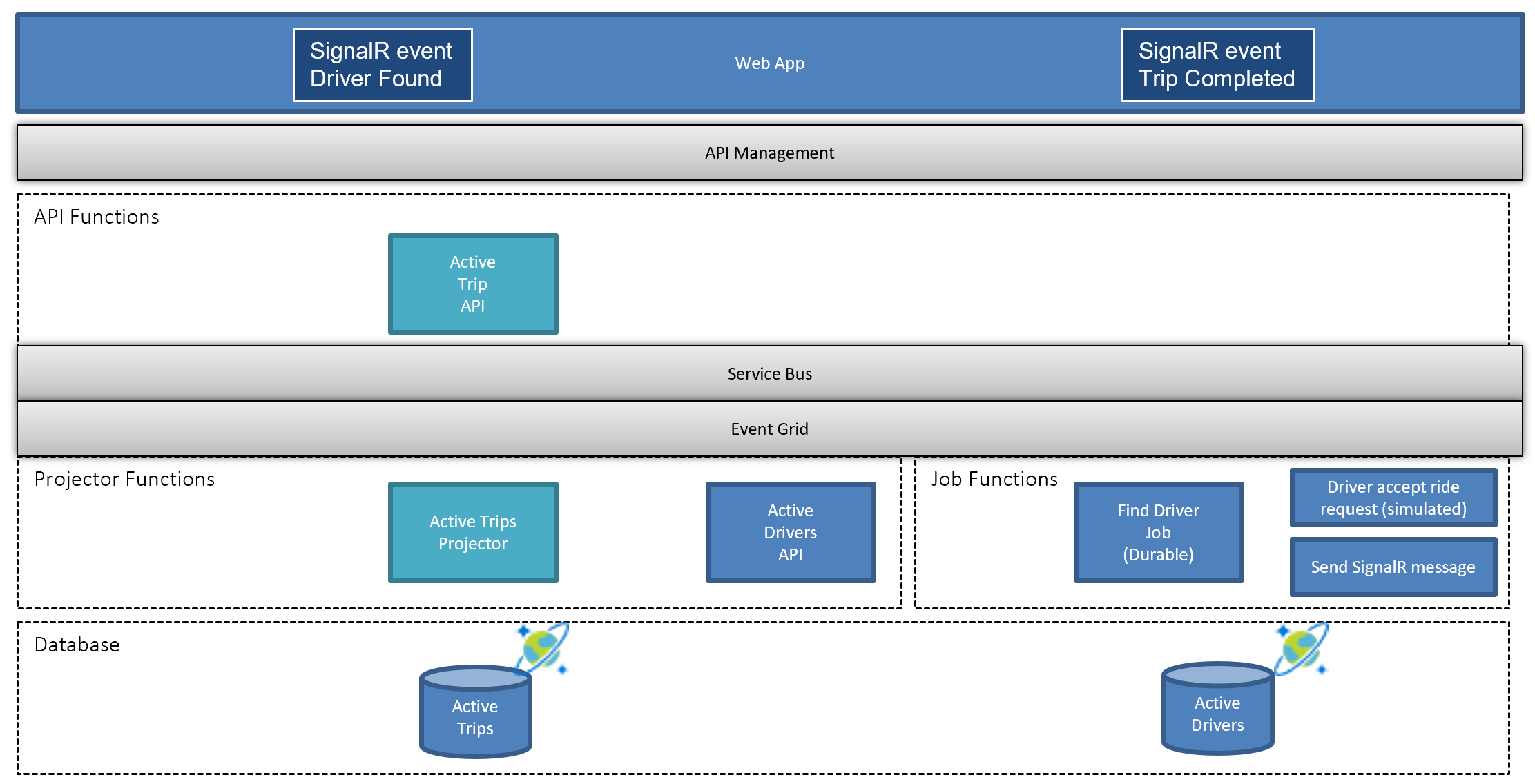
We will need to seed the database with rider history.

### 3. Trip activity page

From the user’s perspective (passenger or driver), they’d only see their active trips, assuming you’d only see one at a time. An admin (which we’d just give the user admin access for the POC) view would show all active trips. I think we shouldn’t restrict people from creating many trips as different users, but for simplicity’s sake, we should have a button to kick off several simulated, so they can see everything “flow” through the system.

### 4. Active trip / passenger app simulation

User lands on this page with new trip.



1. Rider uses web app to start a new ride, which passes in their location (simulated location)
2. Request goes through API Management to an Azure Function, which kicks off an event that is consumed by a Durable Function
3. The Durable Function uses function chaining to perform the following:
   1. Call function to retrieve list of active drivers from Cosmos DB that are within 15 miles of the passenger
   2. Pass list of closest active drivers to another function that simulates a random driver accepting the ride request
   3. Create and send event that driver was found (Event Grid)
      1. Consumed by Azure function that sends to SignalR Service, which updates UI
      2. Consumed by Azure function that saves data to Cosmos DB
      3. Consumed by Logic App that sends an email notification

**Event Grid Trigger /activetrips/{id}/driverfound**

When the trip is started, the trip information is updated by a Function that periodically updates the location and status. The web UI will be updated through push updates via websockets, as provided by the SignalR service. When the ride is over; completed, another event fires that should tell UI to refresh and ask for rating: POST /activetrips/{id}/rating