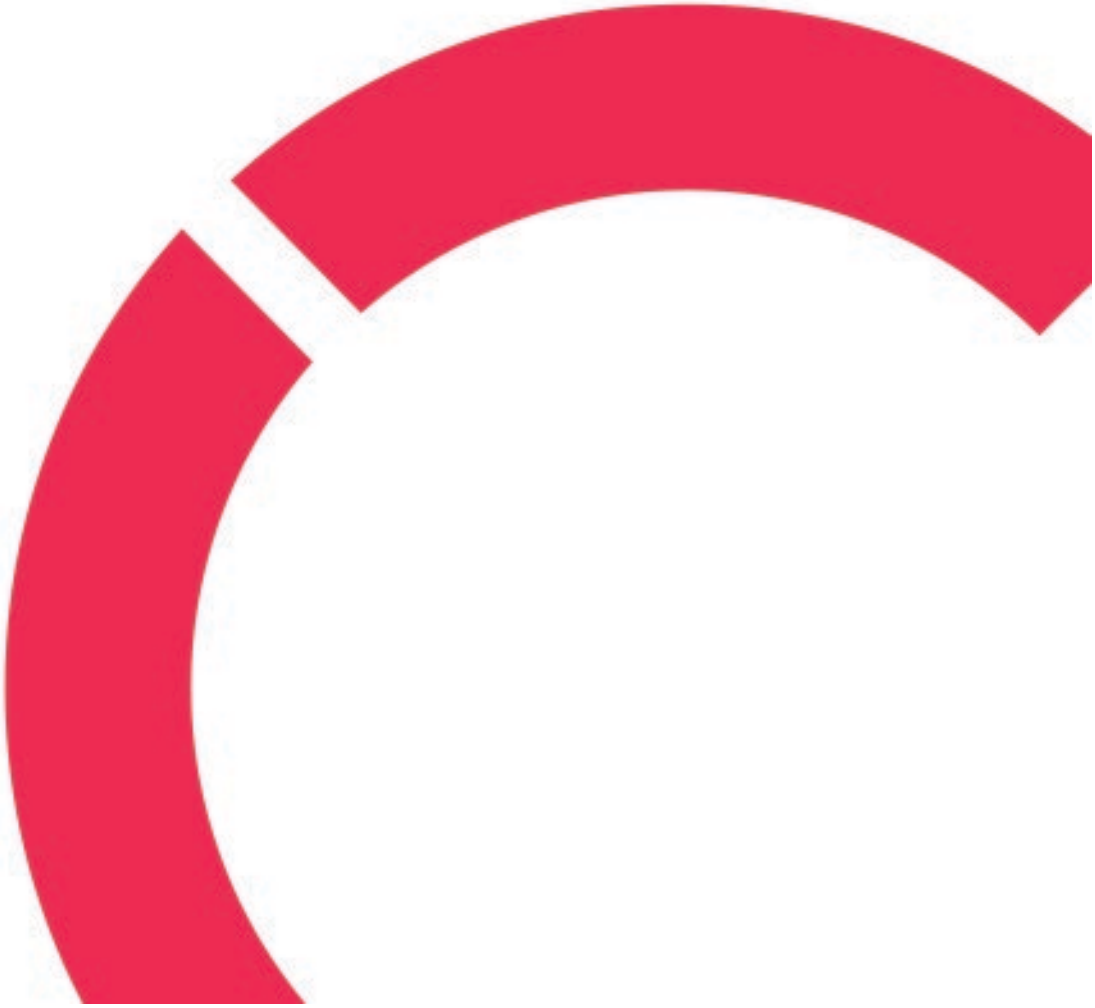
Shejuti Rahman

DEVELOPMENT OF AN IOT BACKEND SERVICE WITH AZURE

Thesis
CENTRIA UNIVERSITY OF APPLIED SCIENCES
Bachelor of Engineering, Information Technology
February 2025



ABSTRACT

Centria University of Applied Sciences	Date February 2025	Author Shejuti Rahman
Degree program Bachelor of Engineering, Information Technology		
Name of thesis DEVELOPMENT OF AN IOT BACKEND SERVIECE WITH AZURE		
Centria supervisor Janne Peltoniemi, Kyösti Marjakangas, Jari Isohanni		**Pages** 48 + 9

This thesis aimed to develop an Internet of Things (IoT) backend service using Microsoft Azure. The focus was to create a RESTful Application Programming Interface (API) in Python to manage IoT device data stored in Azure SQL. The API handled adding, deleting, updating records, modifying device IDs, verifying data accuracy, and ensuring secure CRUD operations with basic user validation.

The theoretical framework provided an overview of backend service development, emphasizing data storage and processing for IoT devices. Key tools included RESTful APIs, Python, JSON format, and SQL databases. This practice-based thesis utilized Azure Functions for serverless computing, Azure SQL for secure data storage, and Postman to test API functionality.

The project simulated IoT data in a custom-designed Azure SQL database, representing device information and temperature readings, as real IoT devices were unavailable. The system uniquely combined Azure Functions, a custom SQL schema, and RESTful APIs to enhance scalability and deployment.

The results showed that the backend service with Azure and the RESTful API with Python were effective in managing data, enabling device management, and safeguarding data integrity through validation. This work highlighted the development of a reliable and functional system for managing IoT-related data

In conclusion, this thesis demonstrated the development of a reliable and efficient IoT backend service using Microsoft Azure and a RESTful API with Python. The system ensured secure, validated data management in SQL databases and provided a strong foundation for future IoT data analysis.

Keywords
Azure Functions, Azure SQL. CRUD Operations, IoT Backend, Postman Testing, RESTful API, SQL Bindings, Python

CONCEPT DEFINITIONS

API

Application Programming Interface (API) is like a messenger that allows different software applications to interact and share data.

Azure Functions

A Microsoft service that runs small programs in the cloud without needing to set up or manage servers.

Azure Portal

An online platform provided by Microsoft to help users manage and monitor their Azure cloud services.

Azure SQL

A Microsoft cloud service that stores and manages data, designed to be reliable, able to handle growth, and secure.

Cacheability

A feature in web systems that stores responses temporarily so they can be reused without sending the same request multiple times.

CRUD

Create, Read, Update, delete – The four fundamental operations for managing data in a database.

Data Validation

Ensuring that the data entered into a system is correct, organized, and useful for its intended purpose.

HATEOAS

Hypermedia as the Engine of Application State (HATEOAS) is a rule for RESTful APIs to support dynamic and self-descriptive communication between clients and servers.

HTTP

HyperText Transfer Protocol (HTTP) is a set of rules used to transfer websites and other online content over the internet.

IoT

The Internet of Things (IoT) is a network of devices, like sensors and machines, connected to the internet to share and collect data.

JSON

JavaScript Object Notation (JSON) is a simple format for storing and sharing data that is easy for both people and computers to use.

Postman

A tool used for testing APIs by sending HTTP requests and analyzing responses.

REST

Representational State Transfer (REST) is a set of design rules for building APIs that use standard web protocols like HTTP to make communication between systems easier.

RESTful API

Representational State Transfer Application Programming Interface – A type of API that conforms to REST architectural principles.

Serverless Computing

A cloud-computing model where the cloud provider automatically decides how much computing power and resources are needed and adjusts them as required, without any manual setup or changes by the user.

SQL

Structured Query Language – A language used for managing and manipulating relational databases.

SQL Bindings

Mechanisms in Azure Functions that connect functions to SQL databases, allowing seamless data input and output.

Statelessness

A principle where each request to a server contains all the information it needs, so no previous interactions are required.

URL

A Uniform Resource Locator (URL) is a precise web address that directs users to a specific webpage on the internet.

VSC

Visual Studio Code (VSC) is a free, code editor developed by Microsoft, widely used for writing and debugging code in various programming languages.

XML

Extensible Markup Language (XML) is a markup language and file format for organizing and sharing data.

ABSTRACT
CONCEPT DEFINITIONS
CONTENTS

FIGURES

CODES

1 INTRODUCTION

The Internet of Things (IoT) has emerged as a rapidly growing technology enabling interconnection and communication between devices and machines. IoT has found applications in various domains, including healthcare, agriculture, transportation, and home automation. With the increasing number of connected devices, there is a growing need for an IoT backend service to manage and maintain the data generated by these devices. (Bahga & Madisetti 2014, 20-22; Rose, Eldridge & Chapin 2015, 4-5, 8, 11-12, 18, 22.)

To address these needs, an IoT backend service with Azure has become a popular solution. Azure is a cloud computing platform that provides a range of services for IoT, including data processing, storage, and analysis. Developing a Representational State Transfer Application Programming Interface (RESTFUL API) with Python provides a lightweight and efficient means of accessing and managing data from IoT devices. (Microsoft Learn 2024e; GeeksforGeeks 2024a.)

Developing an IoT backend service with Azure and a RESTFUL API using Python is a significant contribution to the field of IoT. This backend service, implemented in Python, utilizes a RESTFUL API to access and maintain data from IoT devices. Specifically, the API employs the POST method to add device ID and temperature data in JavaScript Object Notation (JSON) format, and the GET method to retrieve this data. Additionally, the Application Programming Interface (API) allows users to update device information and validate the data. Postman Application (APP) tests and validates these endpoints, ensuring accurate data handling. The managed data is stored in Azure Structured Query Language (SQL), enabling future analysis and maintenance. (Microsoft Learn 2023; Microsoft Learn 2024f; Postman Learning 2023; What is REST?.)

Building the backend service for IoT in Azure is advantageous because Azure's suite of services provides a comprehensive platform for developing, deploying, and managing IoT solutions at scale. This thesis focuses on explaining the backend services responsible for essential maintenance tasks, such as retrieving device information, updating device details (including temperature data), and deleting devices. These services are implemented using Azure Functions for serverless computing, Azure SQL Database for data storage and retrieval, and Azure Functions' App Keys to ensure secure access to Hyper Text Transfer Protocol (HTTP) triggered APIs and protect IoT data. (Microsoft Learn 2023; Microsoft Learn 2024f; Microsoft Learn 2024b; What is REST?.)

This thesis uses Azure Functions and a custom-designed database to efficiently manage data from multiple devices. Python was used to Create, Read, Update, and Delete (CRUD) operations, tailored to ensure smooth functionality and data integrity. A custom dataset was generated and stored in Azure SQL to simulate IoT device information and temperature readings, allowing the backend system to be tested and validated in the absence of real IoT data. This project does not focus on extracting or collecting data directly from IoT devices or sensors. Instead, it addresses the task of managing and maintaining IoT data once it is stored in the Azure SQL database. By implementing CRUD operations and ensuring secure data handling, this thesis contributes to the effective management of device-generated information.

In summary, this thesis will cover how these Azure Functions collectively manage devices and their associated data, including temperature readings, in an application. Additionally, it will discuss how to maintain the extracted data and save it back to the SQL database in Azure. These backend services are crucial for effectively operating IoT applications, ensuring data is accurately maintained and readily accessible for analysis and other functions.

2 THEORETICAL BACKGROUND

The main literature sources used in this thesis include Azure documentation, which provides insights into cloud computing and IoT backend services; REST API development guidelines for creating efficient data management systems; Python programming resources for implementing the backend service; and SQL database management materials for handling data storage and retrieval. Key concepts applied in this thesis include the Postman App for API testing, Python programming for backend development, and SQL database management for maintaining IoT device data in Azure.

2.1 Azure Cloud Services

Azure is Microsoft's public cloud platform, offering a broad collection of services such as computing, analytics, storage, and networking. Azure supports various service models, including Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). By leveraging virtualization technology, Azure provides scalable and flexible cloud solutions that can efficiently emulate hardware in software, allowing users to deploy and manage applications without worrying about underlying infrastructure maintenance. (Microsoft Azure 2024c; Microsoft Azure 2024b; GeeksforGeeks 2024b.)

Amazon Web Services (AWS) and Google Cloud Platform (GCP) also offer services to manage IoT data, but Azure was chosen for its seamless database integration, efficient application management, and flexible pricing. AWS provides Amazon Relational Database Service (Amazon RDS) and AWS Lambda, but AWS Lambda does not maintain a persistent database connection, requiring additional setup such as connection pooling or Amazon RDS Proxy to ensure smooth operations. In contrast, Azure Functions natively integrate with Azure SQL Database, simplifying the process and reducing complexity. Google Cloud Platform (GCP) is known for its strong ability to process large amounts of data and support AI-powered analytics. This makes it a good choice for businesses that work with big data. However, companies that already use Microsoft’s systems may find it harder to integrate GCP smoothly into their existing setup. Another reason for choosing Azure is its widespread adoption in enterprises that already rely on Windows Server, SQL Server, and Active Directory (AD), making integration with existing systems easier. Additionally, Microsoft provides cost-saving benefits, such as the Azure Hybrid Benefit, which helps businesses that already use Microsoft products save on cloud

expenses (AWS Documentation 2025a; AWS Documentation 2025b; AWS Repost 2025; Microsoft Learn 2024j; Microsoft Learn 2025).

In the context of my thesis, Azure's benefits are particularly relevant. Azure Functions, a serverless computing service allows to execute code in response to triggers such as HTTP requests without managing servers. This significantly reduces the time and effort needed for infrastructure management, enabling a focus on application development and deployment. Similarly, Azure SQL Database offers a managed relational database service with features like automated backups, scalability, and high availability. This ensures that the IoT backend can handle data efficiently and reliably, providing a robust platform for managing device data and performing CRUD operations. (Microsoft Learn 2023; Microsoft Azure 2024a; Microsoft Learn 2024f.)

These Azure services are highly accessible and can be configured through the Azure portal, which simplifies infrastructure management and accelerates development cycles. By utilizing Azure Functions and Azure SQL Database, IoT backend service benefits from Azure's security, scalability, and reliability, ensuring applications can scale efficiently as the number of connected devices grows. This setup allows for public accessibility, avoiding the need for ongoing server updates and network management, making it an ideal solution for your IoT project. (Microsoft Learn 2023; Microsoft Learn 2024g; Microsoft Learn 2024f.)

2.2 Restful Web Services

A RESTful web service is a web service that follows the principles of REST (Representational State Transfer) architecture. It operates using the standard HTTP protocol and employs its methods (GET, POST, PUT, DELETE) to execute CRUD operations on resources. These services explicitly use HTTP methods, manage stateless interactions, and are accessible through Uniform Resource Identifiers (URIs). (Fielding 2000; What is REST?.)

A defining characteristic of RESTful web services is their stateless nature. Each request from a client to the server must include all necessary information to understand and process the request because the server does not retain any state of the client session. This ensures that each interaction is independent and self-contained. (Fielding 2000; What is REST?.)

RESTful services are structured around resources, which serve as their primary abstraction. Each resource is identified by a unique URI. For example, in a RESTful web service designed to manage a collection of devices, endpoints might include GET /devices to retrieve a list of devices, POST /devices to add a new device, GET /devices/{id} to fetch a specific device by its ID, PUT /devices/{id} to update a specific device by its ID, and DELETE /devices/{id} to remove a specific device by its ID. These services employ standard HTTP methods to manipulate resources. The GET method retrieves data, POST creates new resources, PUT updates existing resources, and DELETE removes resources. This uniform interface simplifies interaction with the web service and enhances predictability. (What is REST?.)

In RESTful services, resources are typically represented in JSON or Extensible Markup Language (XML) format. Clients interact with these representations, which can be exchanged between the client and server, facilitating easy data transfer and integration with other systems. (Gupta 2023.)

RESTful web services are extensively used to develop maintainable web applications. Their popularity stems from leveraging existing web standards and technologies, making integration with other services and applications straightforward. By adhering to REST principles, these services ensure scalability, performance, and simplicity in web application development. (What is REST?; Tuple.)

2.2.1 Rest

REST (Representational State Transfer) is an architectural style for designing networked applications. It relies on a stateless, client-server, cacheable communications protocol, with the HTTP protocol being almost universally used for this purpose. (Fielding 2000; What is REST?.)

One of the key constraints of REST is statelessness. This means that each client-server interaction is stateless, and the server does not store any information about the client state between requests. Each request from a client must contain all the necessary information to understand and process the request. Another important constraint is the client-server architecture. In this architecture, the client and server are separate entities that communicate over a network. This separation of concerns allows the client and server to evolve independently, making the system more modular and flexible. (Fielding 2000; The Six Constraints.)

Cacheability is also a critical aspect of REST. Responses from the server must be explicitly marked as cacheable or non-cacheable. This improves performance by reducing the number of interactions between the client and server, as cacheable responses can be reused for subsequent requests. (Fielding 2000; The Six Constraints.)

REST is defined by four interface constraints. The first is the identification of resources, where resources are identified in requests through Uniform Resource Identifiers (URIs). The second constraint is the manipulation of resources through representations, meaning clients manipulate resources using representations (e.g., JSON, XML) of the resource. The third constraint is that messages must be self-descriptive, including enough information to describe how to process the message. The fourth constraint is Hypermedia As The Engine Of Application State (HATEOAS), where clients interact with resources entirely through hypermedia provided dynamically by application servers. (What is REST?; Tuple.)

A layered system is another characteristic of REST. In this system, a client cannot ordinarily tell whether it is connected directly to the end server or an intermediary along the way. This enhances scalability and security by allowing intermediaries to handle aspects such as load balancing and security enforcement. (Fielding 2000; Gupta 2024.)

Code on demand is an optional constraint in REST. This allows servers to extend client functionality by transferring executable code (e.g., applets, scripts) to the client, enhancing the flexibility and capabilities of the client. (Fielding 2000.)

The advantages of REST include scalability, due to statelessness and cacheable responses. Performance is also enhanced, as RESTful services are typically lightweight, reducing latency and improving response times. Modifiability is a key benefit, as the separation of client and server allows them to evolve independently. Finally, simplicity is a notable advantage, as RESTful systems are often simpler and easier to understand and implement. (Fielding 2000; Tharindu 2024.)

This thesis highlights the benefit of simplified server management, reduced server load, and the ability to independently update the client and server. The uniform interface and stateless interactions make the API easier to develop and use, particularly when tested with tools like Postman. These features collectively ensure that the API remains efficient, scalable, and easy to maintain, meeting the project's requirements effectively. (Fielding 2000; Tharindu 2024.)

2.2.2 GET / POST

In web development, GET and POST are the two most common types of HTTP requests used for client-server communication. These methods serve different purposes and are fundamental to how the web functions. (W3Schools 2024a; Stephens 2023.)

The GET method is used to retrieve data from a server without altering it, making it ideal for repeated access to information. This method is particularly useful when specific information is needed, such as listing all devices from an SQL database or obtaining details about a particular device by its ID. Since GET requests include parameters in the URL, they are appropriate for searches and data retrieval tasks where the visibility of request details is not an issue. (RapidAPI 2024a; Stephens 2023.)

When a user enters a URL into their web browser or clicks on a link, a GET request is sent to the server, asking for the specified resource, such as a webpage or an image. The server processes this request and sends back the requested information. A notable feature of GET requests is that the data being requested is appended to the URL, making it visible in the browser's address bar. This characteristic allows GET requests to be cached and bookmarked by users, enhancing their efficiency for retrieving and displaying data. (Host4Geeks 2023; MDN Web Docs 2024a.)

However, because GET parameters are part of the URL, they are visible and stored in the browser history. This visibility limits the data size and can reduce security, as sensitive information might be exposed. Despite these drawbacks, the GET method remains a convenient and effective way to access and retrieve non-sensitive information from a server, especially when data security is not a primary concern. (FreeCodeCamp 2022; Apidog 2024a.)

On the other hand, POST requests send data to a server for processing, commonly used when a user submits a form on a website, such as a login, registration, or data entry form. The data is included in the body of the request, enhancing security since it is not visible in the URL. The server processes the data and performs necessary actions, such as saving it to a database or updating records. Unlike GET requests, POST requests are not cacheable or bookmarkable, making them suitable for actions that modify server-side data. (MDN Web Docs 2024b; RapidAPI 2024b.)

POST requests create new items or update existing ones. They can modify server-stored data, so repeating the same request can yield different results each time. In this project, POST is used to add

new device entries and record temperature data for devices. These requests involve reading JSON data, storing it in an SQL database, and sending a response to indicate the operation's success. Unlike GET, POST parameters are in the request body, enhancing security and avoiding URL length limitations. Additionally, POST requests are typically not saved in browser history or cached, making them ideal for transmitting larger amounts of data and sensitive information. (Apidog 2024b.)

The primary difference between GET and POST requests lies in how data is transmitted and processed. GET requests are used for retrieving data without affecting the server's state, while POST requests are used for sending data that may result in changes on the server. Understanding the appropriate use of these methods is crucial for developing efficient and secure web applications. (FreeCodeCamp 2022; W3Schools 2024a.)

In summary, GET and POST requests are integral to the functionality of the web, each serving distinct purposes in client-server communication. GET requests are ideal for fetching data, while POST requests are essential for submitting data. Proper implementation of these methods ensures effective and secure data transmission in web applications. (FreeCodeCamp 2022; W3Schools 2024a.)

2.2.3 JSON

JSON is a way to format data that is easy for people to read and write, and simple for computers to understand and generate. It uses pairs of keys and values and lists, making it popular for exchanging data between web clients and servers, especially in RESTful APIs. JSON is not limited to any specific programming language but is familiar to programmers using JavaScript, Python, and Java. This makes JSON great for sharing data on the web. Its structure, with objects in curly braces {} and arrays in square brackets [], allows for flexible and clear data representation. (JSON org; MDN Web Docs 2024c.)

For example, in this thesis, JSON was used to add new device entries and record temperature data. The JSON data has an outer object in curly braces {}. Inside this object, there is a key "device" with a value that is another object with details about the device, like its ID, name, type, and status. There is also a key "temperatureReadings" with an array of temperature readings. Each reading is an object with "timestamp" and "temperature" keys, showing the time and the recorded temperature.

Here is an example of this JSON data:

```
{
  "device": {
    "id": 9,
    "name": "Thermostat",
    "type": "Temperature Sensor",
    "status": "active"
  },
  "temperatureReadings": [
    {
      "timestamp": "2024-06-20T15:30:00Z",
      "temperature": 25.5
    },
    {
      "timestamp": "2024-06-20T16:00:00Z",
      "temperature": 26.0
    }
  ]
}
```

Code 1. Example of json data.

This format lets JSON clearly and effectively show complex data. JSON helped communicate between client-side applications and the RESTful API, making it easy to retrieve, add, update, and delete device data in the SQL database. Tools like Postman were useful for testing the API endpoints. Also, JSON's small data size reduced the amount of data sent, improving performance, especially in mobile or low-bandwidth situations.

2.3 SQL Databases

SQL is a standard programming language designed for managing and manipulating relational databases. It enables users to execute tasks like querying, updating, and structuring database management. SQL databases use a tabular format, consisting of rows and columns, to organize data effectively, making them suitable for complex queries and transaction processing. Each column represents an attribute of the data, and each row represents a record, facilitating efficient data organization and manipulation. Relationships between tables are established through primary and foreign keys, enabling sophisticated data modeling. (GeeksforGeeks 2024c; W3Schools 2024b.)

Essential SQL commands include SELECT, INSERT, UPDATE, and DELETE, which are vital for managing structured data in Relational Database Management Systems (RDBMS). These commands allow users to create, read, update, and delete database records. They also help define database schemas, manage access control, and ensure data integrity using constraints like primary and foreign keys. SQL is fundamental for maintaining organized and efficient data handling in relational databases. (W3Schools 2024b.)

SQL is highly integrated with Azure, Microsoft's cloud platform, through services like Azure SQL Database and Azure SQL Managed Instance. These services offer scalable, managed database solutions that seamlessly integrate with other Azure services, enhancing data storage, analysis, and management. This integration simplifies the deployment, management, and scaling of SQL databases in the cloud, benefiting applications that require robust data handling and transaction processing capabilities. (Microsoft Learn 2024f.)

In the context of IoT (Internet of Things), SQL databases are beneficial for storing and analyzing the vast amounts of data generated by IoT devices. They enable efficient querying and management of this data, supporting real-time analytics and decision-making. However, SQL databases have limitations, such as scalability issues with extremely large datasets and rigidity compared to NoSQL databases, which are more flexible in handling unstructured data. (Parmar 2023.)

2.4 Access Control

Access control is an important part of system security because it controls who or what can access certain resources. The main goal is to protect sensitive data and functions while making sure that authorized users or devices can still perform their tasks. A key idea in access control is the principle of least privilege, which means giving users or systems only the access they need and nothing more. This reduces the risk of security breaches and limits the potential damage if something goes wrong. Access control can be implemented in many ways, including shared secrets, tokens, or role-based permissions. (Microsoft Learn 2024h; Microsoft Learn 2024i.)

In cloud platforms like Azure Functions, access control is especially important for securing APIs and backend services. Azure App Keys provide a simple yet effective way to manage access. These keys work like shared passwords and can be assigned to individual functions or the entire application. They

are lightweight and ideal for systems like IoT, where devices often have limited processing power. Administrators can also easily revoke or update App Keys to maintain security without disrupting the entire system. (Microsoft Learn 2024h; Microsoft Learn 2024i.)

In this thesis, Azure Functions' App Keys were used to protect backend API endpoints. Each function was assigned a unique key, ensuring that only authorized requests could access it. This followed the principle of least privilege, as each key provided access only to the specific function it was linked to. Assigning unique keys also made access management simpler. For example, if a key needed to be revoked due to a security issue, only that function would be affected, leaving the rest of the system unaffected. During testing, tools like Postman were used to simulate API requests. The x-functions-key was included in the request headers as a shared secret to validate access. If a request did not include the correct key, it was immediately rejected. Although App Keys were effective for this project, future improvements could include more advanced methods, such as OAuth 2.0, which would support user-specific access and provide even more flexibility. (Microsoft Learn 2024i.)

3 BUILDING IOT BACKEND WITH AZURE

Building an IoT backend with Azure involves using Azure Functions and Azure SQL Database to manage device data and developing RESTFUL APIs for backend operations. Azure Functions provides a serverless compute service that triggers events like HTTP requests to handle operations like data creation, reading, updating, and deletion. RESTFUL APIs are developed to facilitate these operations, ensuring seamless communication between clients and backend services.

New device records and temperature readings are processed by Azure Functions and securely stored in Azure SQL Database using robust data handling techniques. The database ensures high availability, automated backups, and scalability, integrating seamlessly with Azure Functions for secure data operations.

Figure 1 visually represents the overall project architecture, demonstrating the interaction between the development environment, deployment platform, client, and SQL database. The workflow begins when Postman sends an HTTP request to a RESTFUL API's endpoint. This request triggers the corresponding Azure Function. The function processes the request and interacts with the SQL database to execute SQL queries. The SQL Server processes these queries and returns the results to the Azure Function. The function then constructs an HTTP response with the processed data and sends it back to Postman, completing the cycle. The step-by-step explanations are given below.

Visual Studio Code (VSC): The development of the RESTFUL API begins in VSC using Python. The RESTFUL API provides the endpoints with which the client (Postman app) interacts. Each endpoint corresponds to a specific operation managed by the Azure Functions. When an HTTP request is sent to an endpoint, it triggers the appropriate function within the Azure Function App. At this point, Azure Functions (HTTP triggers) are written to handle HTTP requests and interact with the SQL database. Once the code is developed, it is deployed to the Azure Cloud Service as an Azure Function App. This step ensures the serverless functions are correctly hosted and ready to handle incoming requests.

Azure Function App: The Azure Function App is responsible for hosting the serverless functions. These functions are triggered by HTTP requests and handle operations such as adding new devices, recording temperature readings, retrieving device information, updating device details, and deleting

devices. When an HTTP request is received, the function app processes the request, executes the necessary SQL query on the SQL Server, and returns the response.

Postman App: For testing purposes, the Postman app is used to send HTTP requests to the API endpoints. This allows for thorough testing of the various functionalities provided by the API, such as adding devices, retrieving device details, updating devices, deleting devices, and recording temperature readings. The endpoints specified in Postman correspond to the operations supported by the Azure Functions.

SQL Server: The SQL Server hosts the database that stores IoT data, including device information and temperature readings. When the Azure Function App triggers a query, the SQL Server executes it to read from or write to the database. The results of these queries are then sent back to the Azure Function App, which uses this data to construct the HTTP response.

This architecture, developed using Python, ensures seamless communication between the client, the backend services, and the database, providing a robust and scalable solution for managing IoT device data.

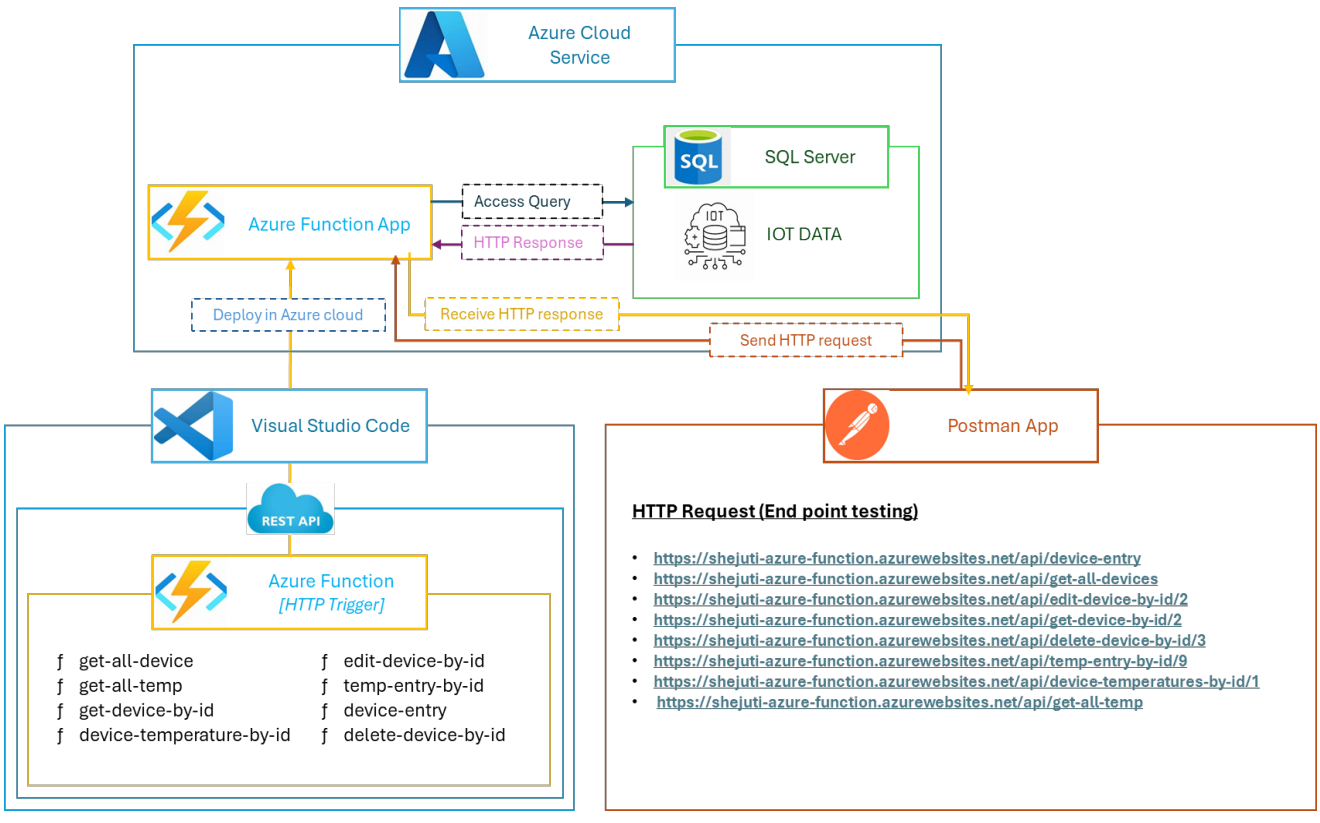


FIGURE 1. Project architecture

3.1 System Configuration

Initially, all the necessary tools and systems must be set up properly. Access to these systems needs to be ensured by signing up. Three different platforms or systems are used: Microsoft Azure, Visual Studio Code, and Postman App. The seven steps to be followed are described below.

3.1.1 Sign Up for an Azure Account

As a new starter, one must create an account with Azure. To create a new Azure account, start from Azure's website (https://azure.microsoft.com/en-us/pricing/purchase-options/azure-account/) and click on the "Try Azure For Free" button. Then follow the prompts to create a new Azure account. Email addresses, creating passwords, and verifying identity are needed for that. A phone number and a credit card are required to verify identity. Credit cards are for identity verification only and will not be charged unless one upgrades to a paid plan.

3.1.2 Install Python

Python needs to install from https://www.python.org/downloads/. Choose any version of python 3.8 - python 3.11and install, these are the supported versions for Azure functions (Microsoft Learn 2024a).

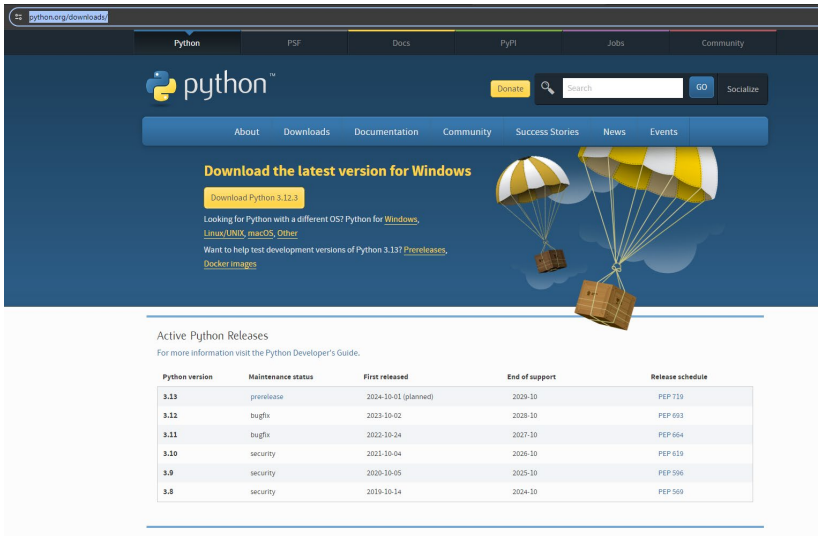


FIGURE 2. Download Python (adapted from Python org)

It is worth noting that there are various methods for installing and managing Python, depending on the user's needs. For example, the Anaconda distribution includes Python pre-installed and can be configured by following the setup instructions provided on the Anaconda website https://www.anaconda.com/download (Anaconda Documentation, 2024). Another example is PyCharm, an integrated development environment (IDE), which allows users to install and configure Python during its setup process (JetBrains 2024). In this thesis, installing Python is essential, as it will be used alongside Visual Studio Code for developing Azure Functions.

3.1.3 Install Visual Studio Code

In this thesis Visual Studio Code has been utilized for deployment, Facilitating seamless debugging and deployments. VSC is one of the supported versions of Azure Function. The Azure Functions extension for Visual Studio Code facilitates the local development of functions and their deployment to Azure. Figure 3 shows that Visual Studio Code is downloaded from https://code.visualstudio.com/. (Microsoft Learn 2024c.)

After installation, The Python extension for Visual Studio Code from: (https://marketplace.visualstudio.com/items?itemName=ms-python.python), the Azure Functions extension for Visual Studio Code from: (https://marketplace.visualstudio.com/items?itemName=ms-azuretools.vscode-azurefunctions), and The Azurite V3 extension local storage emulator version 1.8.1 or later for Visual Studio Code from (https://marketplace.visualstudio.com/items?itemName=Azurite.azurite) need to install. (Microsoft Learn 2024d.)

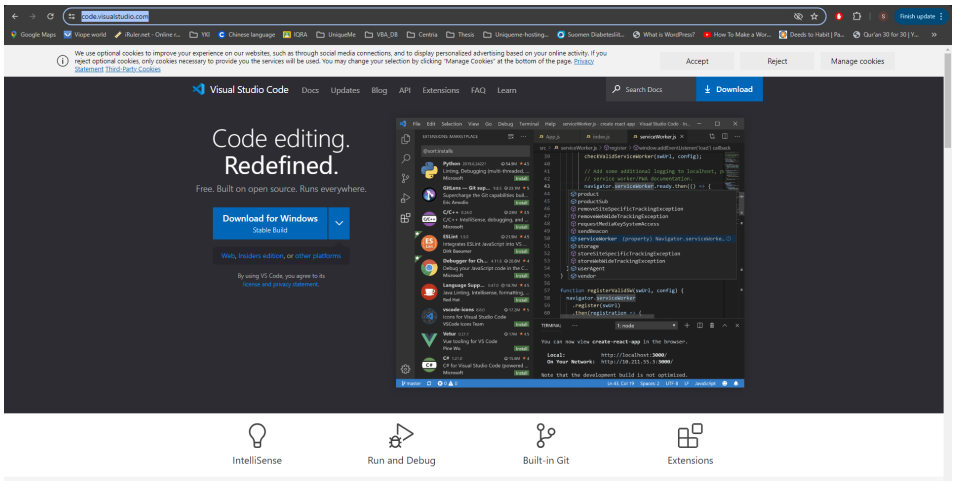


FIGURE 3. Download Visual Studio Code

3.1.4 Update or install Core Tools

After installing Visual Studio Code, Azure Functions Core Tools need to be installed before creating an Azure function. To initiate the installation, open the Visual Studio Code command palette by pressing F1, then type the command "Azure Functions: Install or Update Core Tools." Core Tools are necessary for running and debugging functions locally. (Microsoft Learn 2024h.)

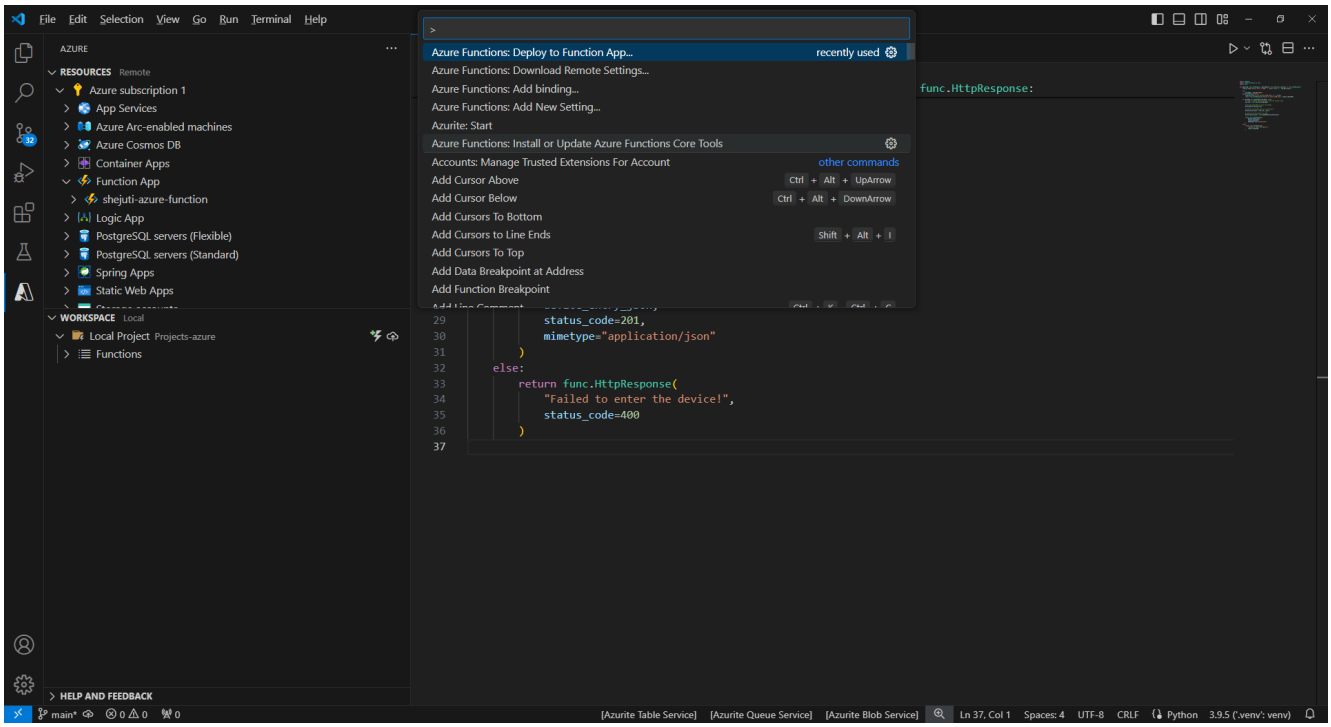


FIGURE 4. Installing Azure Function core tool

3.1.5 Creating Function Locally

Before publishing a project directly to Azure, a local version of an Azure Function is created in Python using Visual Studio Code. The process involves selecting the Azure icon, creating a new project in an empty directory, choosing Python (Programming Model V2), and setting up an HTTP trigger function named `HttpExample` with ANONYMOUS authorization. The `local.settings.json` file is then verified and updated to use development storage. The Azurite emulator is started, and the function is run locally by pressing F5 or using the Run and Debug icon. The function is tested by sending a request with the message body `{ "name": "Azure" }` and checking the function's response in the Terminal panel. Detailed instructions are followed and are available at https://learn.microsoft.com/en-

us/azure/azure-functions/create-first-function-vs-code-python#configure-your-environment. Below, Figure 5 shows the steps for creating and executing the function from Visual Studio Code.

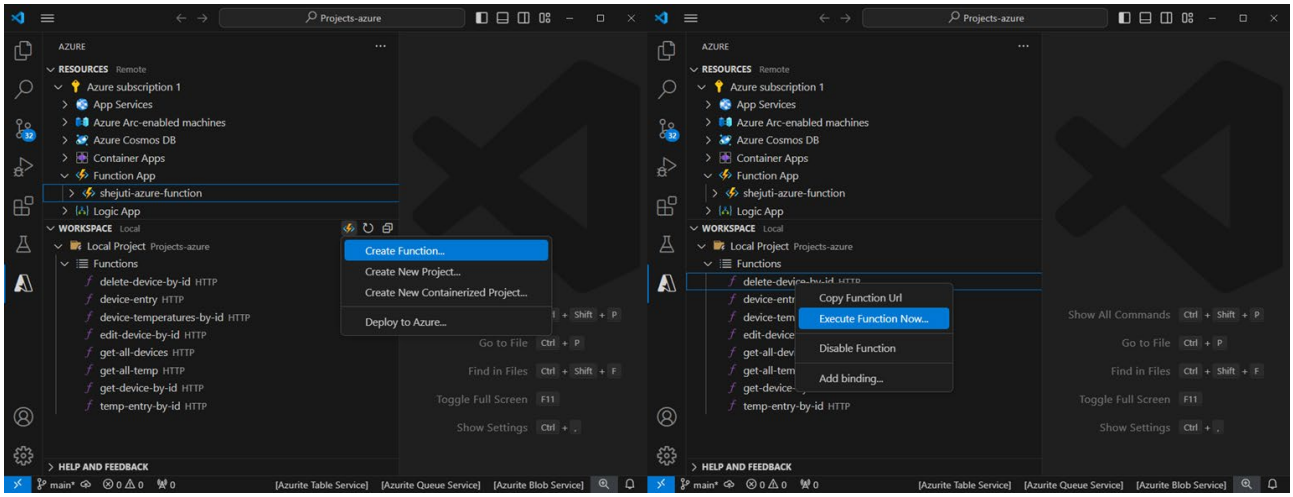


FIGURE 5. Creating and executing functions locally

3.1.6 Write and Deploy Function APP in Azure

To create an app, signing in to Azure is the first step. A function app and related resources are made in the Azure subscription by selecting the Azure icon in the Activity bar, then the "+" icon and "Create Function App in Azure." Information such as the subscription, a globally unique name for the function app, runtime stack, and location should be provided. The Azure extension will display the status of resource creation. Upon completion, a resource group, a standard Azure Storage account, a function app, an App Service plan, and an Application Insights instance are created.

Deploying the local project to Azure is achieved by right-clicking the newly created function app resource in the Resources area and selecting "Deploy to Function App." Figure 6 below shows the function app being right-clicked to deploy in Azure. Any prompts regarding overwriting previous deployments should be confirmed. Once deployment is complete, the output and creation results can be reviewed.

To execute the function in Azure, expand the subscription and the new function app, right-click the HttpExample function, and choose "Execute Function Now." Testing the function involves sending a request with the message body {"name": "Azure"} and checking for the response notification in VSC.

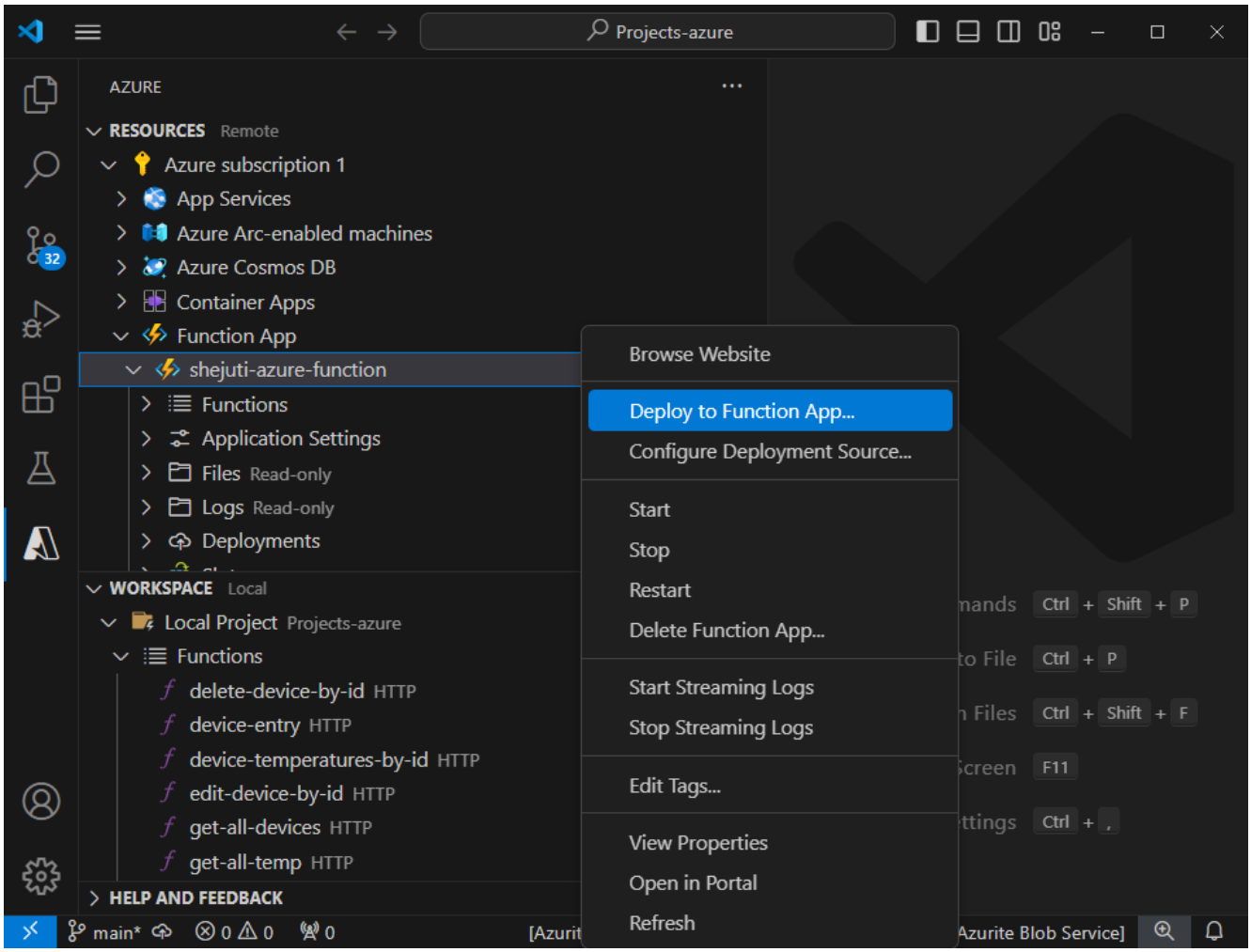


FIGURE 6. Deploying to Azure.

3.1.7 Configure Azure SQL Connection String

An Azure SQL Database was first created through the Azure portal to configure the connection string for the Azure Function App. The steps included navigating to "Create a resource" and selecting "SQL Database". The necessary details such as the subscription, resource group (myResourceGroup), database name (iotdata), server name (shejuti-db-server), server admin login, and password were provided.

The setting "Allow Azure services and resources to access this server" was enabled. Once the database was created, the ADO.NET connection string for SQL authentication was copied from the "Connection strings" section under the SQL Database settings in the Azure portal. The connection string is stated below.

```
Server=tcp:shejuti-db-server.database.windows.net,1433;Initial
```

```
Catalog=iotdata;Persist Security Info=False;User ID=azureuser;Pass-
word={your_password};MultipleActiveResultSets=False;En-
crypt=True;TrustServerCertificate=False;Connection Timeout=30;
```

Next, in Visual Studio Code, the local.settings.json file for the Azure Function project was opened, and the connection string entry was added under the "Values" section. The local.settings.json file was configured as presented below.

```
  "IsEncrypted": false,
  "Values": {
    "AzureWebJobsStorage":
"DefaultEndpointsProtocol=https;AccountName=shejutiazurefunction;Acc
ountKey=iqITs2x6RrCqZNlFVJkHBae7XPB+ZaApucibxsl5QLt8rxF7I8cPVb0msYcp
EkdfB6wf8kQVgbfG+ASt9WIfxA==;EndpointSuffix=core.windows.net",
    "FUNCTIONS_WORKER_RUNTIME": "python",
    "SqlConnectionString": "Server=tcp:shejuti-db-
server.database.windows.net,1433;Initial Catalog=iotdata;Persist
Security Info=False;User
ID=******;Password=******;MultipleActiveResultSets=False;Encrypt=Tru
e;TrustServerCertificate=False;Connection Timeout=30;",
    "FUNCTIONS_EXTENSION_VERSION": "~4",
    "WEBSITE_CONTENTAZUREFILECONNECTIONSTRING":
"DefaultEndpointsProtocol=https;AccountName=shejutiazurefunction;Acc
ountKey=iqITs2x6RrCqZNlFVJkHBae7XPB+ZaApucibxsl5QLt8rxF7I8cPVb0msYcp
EkdfB6wf8kQVgbfG+ASt9WIfxA==;EndpointSuffix=core.windows.net",
    "WEBSITE_CONTENTSHARE": "shejuti-azure-functionaa7202",
    "APPINSIGHTS_INSTRUMENTATIONKEY": "bb0a5c11-48bb-427b-87e3-
233be99d9523",
    "AzureWebJobsFeatureFlags": "EnableWorkerIndexing"
  }
}
```

Code 2. Configuration for Azure SQL Database connection.

This configuration file includes several important settings. The `IsEncrypted` parameter is set to `false`, meaning the settings are not encrypted. The `AzureWebJobsStorage` contains the connection string for the Azure Storage account utilized by the function app. The `FUNCTIONS_WORKER_RUNTIME` specifies that the functions will run using Python. The `SqlConnectionString` includes essential details such as the server name, database name, user ID, and password necessary to establish a secure connection to the SQL database.
The `FUNCTIONS_EXTENSION_VERSION` defines the version of the Azure Functions runtime being used. Both `WEBSITE_CONTENTAZUREFILECONNECTIONSTRING` and

`WEBSITE_CONTENTSHARE` are designated for storing and sharing the function app's content files. The `APPINSIGHTS_INSTRUMENTATIONKEY` is the key for Azure Application Insights, which facilitates monitoring and logging. Lastly, the `AzureWebJobsFeatureFlags` setting enables additional features for the function app.

After creating the database and configuring the connection string, three tables named Device, iotItems, and Temperature were created in the database. Their relationships were defined using the following SQL queries in the Query editor of the Azure portal.

```
CREATE TABLE Device (
    Id UNIQUEIDENTIFIER PRIMARY KEY,
    Name NVARCHAR(200) NOT NULL,
    Description NVARCHAR(MAX) NULL,
    Created_At DATETIME2 NULL,
    Updated_At DATETIME2 NULL
);
CREATE TABLE iotItems (
    Id UNIQUEIDENTIFIER PRIMARY KEY,
    Name NVARCHAR(200) NOT NULL,
    Value NVARCHAR(MAX) NOT NULL
);
CREATE TABLE Temperature (
    Id INT PRIMARY KEY NOT NULL,
    Device_Id NVARCHAR(200) NOT NULL,
    Value NVARCHAR(MAX) NOT NULL,
    Checksum NVARCHAR(MAX) NULL,
    Token NVARCHAR(MAX) NULL,
    Created_At DATETIME2 NULL,
    FOREIGN KEY (Device_Id) REFERENCES Device(Id)
);
```

Code 3. SQL scripts for creating Device, iotItems, and Temperature tables in Azure SQL Database

These commands created the Device table with a unique identifier, name, description, created and updated timestamps; the iotItems table with a unique identifier, name, and value; and the Temperature table with a primary key identifier, device ID, value, checksum, token, and created timestamp, with a foreign key linking to the Device table.

The function code was updated to use this connection string for database operations. Finally, the updated Function App was deployed by right-clicking the Azure Function project in Visual Studio Code and selecting "Deploy to Function App", following the prompts to complete the deployment. By

following these steps, the Azure Function App was configured to securely connect to and interact with the Azure SQL Database using the configured connection string.

3.1.8 Installing Postman to Test Endpoint

The Postman application is downloaded from the official Postman website (https://www.postman.com/downloads/). The Windows version is selected, and the on-screen instructions are followed to complete the installation. After the installation is completed, Postman is launched, and a new account is created or signed into if one already exists. After logging in, a new workspace named "IOT Backend Management API" is created to organize API requests.

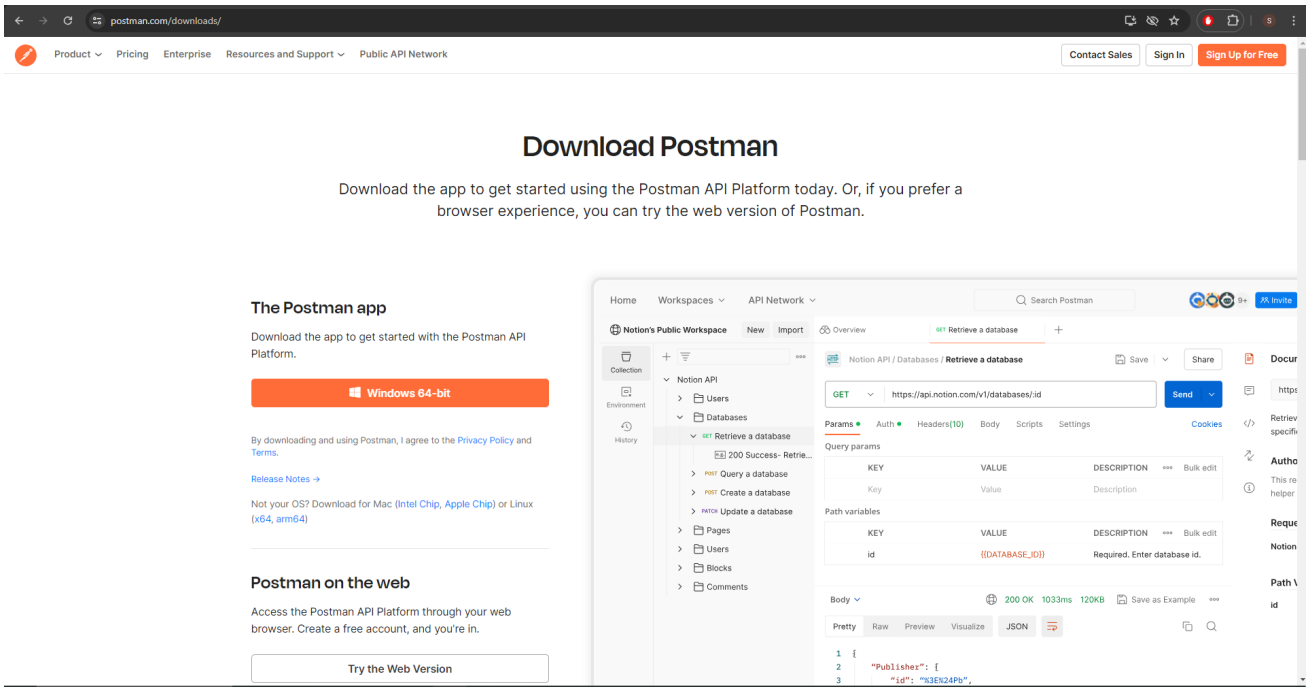


FIGURE 7. Downloading Postman (adapted from Download Postman)

Within this workspace, related requests are grouped into collections. For example, requests for creating a device entry (POST), retrieving all devices (GET), and updating device details (PUT) are organized into specific collections. This intuitive interface allows requests to be configured with the necessary HTTP method, URL, headers, and body data, streamlining the API testing process. Figure 7 shows the official Postman download page, where the Windows version of the Postman application can be

downloaded. It highlights the "Windows 64-bit" download button. Figure 8 illustrates the Postman interface set up for testing API endpoints under the "IOT Backend Management API" collection.

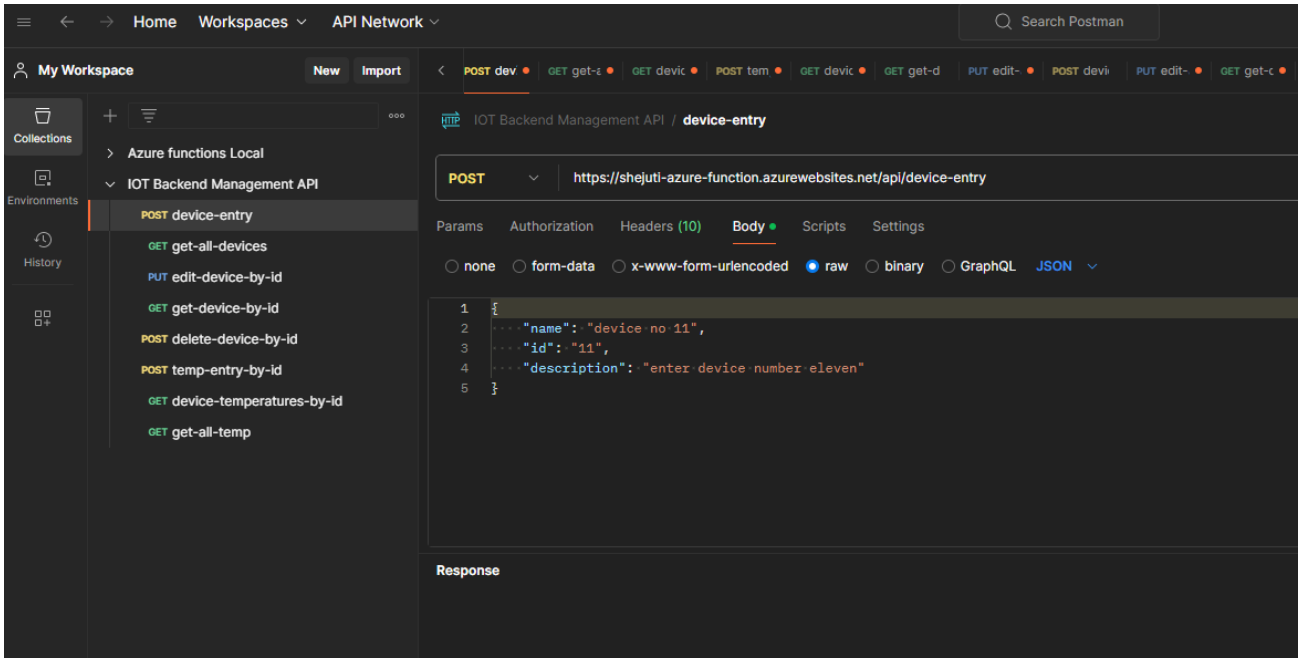


FIGURE 8. Postman Interface

3.2 Code Implementation:

To implement the API that interacts with an SQL database hosted on Azure SQL Server, Python in Visual Studio Code (VSC) is used to write the code. Azure Function App is utilized to create several functions that perform specific tasks. The API, implemented as a set of Azure Functions, is designed to manage and manipulate device data within the SQL database. This includes endpoints for retrieving, adding, updating, and deleting device data and adding and retrieving device temperature data for devices. The functions contained in the API are explained below.

Get All Devices, which retrieves a list of all devices stored in the SQL database. Get Device by ID, which fetches the details of a specific device identified by its ID. Create Device Entry, which adds a new device entry to the SQL database. Edit Device by ID, which updates the details of an existing device identified by its ID. Delete Device by ID, which removes a device from the SQL database by its ID. Enter Temperature by Device ID, which logs temperature data for a specific device and Get Device Temperatures by ID, which retrieves all temperature readings for a particular device.

Specific HTTP requests trigger each function, interact with the SQL database using SQL commands, and return appropriate responses based on the request and the state of the data. Below, each function is explained individually in detail.

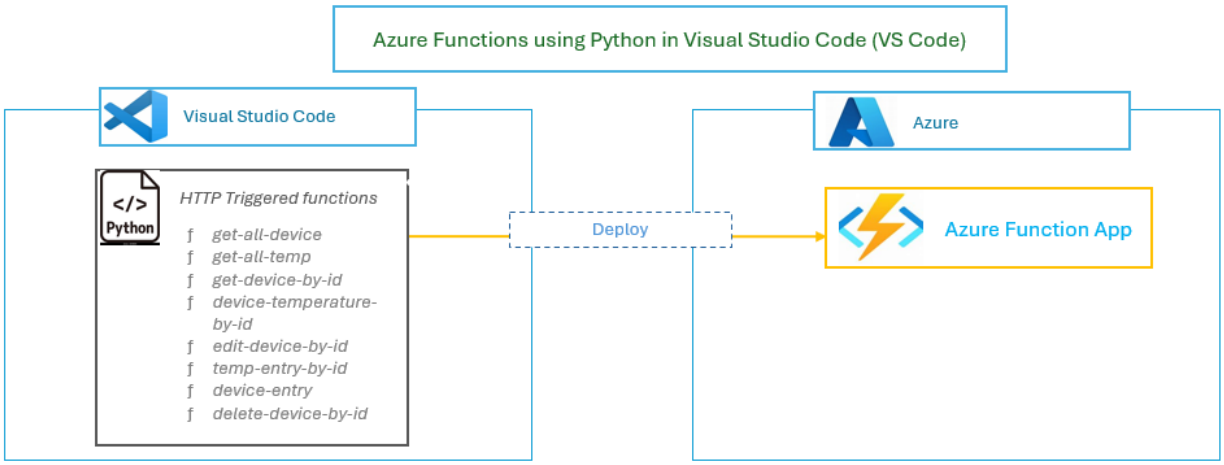


FIGURE 9. Developing and deploying Azure Functions using Python in Visual Studio Code

Figure 9 illustrates the workflow for developing and deploying Azure Functions using Python in Visual Studio Code (VS Code). The process starts with writing HTTP-triggered functions in Python within VS Code. Once developed, these functions are deployed to Azure, resulting in a fully operational Azure Function App that can be accessed and managed through the Azure portal.

3.2.1 Get All Devices:

The Azure Function `get-all-devices` is designed to fetch a list of all devices stored in the SQL database table `dbo.Device` when it receives an HTTP GET request. This function is implemented in Python and consists of two components: a Python code file (`__init__.py`) that defines the function logic and a configuration file (`function.json`) that specifies the function's bindings and interaction with the SQL database. When a GET request is sent to this function, it executes a SQL query to retrieve all devices from the database, converts the resulting data into JSON format, and returns the JSON data with a success status code (`200`). This function provides an efficient way to obtain an overview of all devices stored in the database.

The Python code for the `get-all-devices` Azure Function is written as follows:

```
import logging
import json
import azure.functions as func

def main(req: func.HttpRequest, getAllDevices: func.SqlRowList) ->
func.HttpResponse:
    # Log a message indicating the function is retrieving the device
list
    logging.info('Get a list of all the existed devices :: ')

    # Convert each SQL row into a JSON object
    rows = list(map(lambda r: json.loads(r.to_json()),
getAllDevices))

    # Create an HTTP response with the JSON data and a status code
of 200
    return func.HttpResponse(
        json.dumps(rows),  # Convert the list of devices to a JSON
string
        status_code=200,   # HTTP status code for success
        mimetype="application/json"  # Indicate the response is in
JSON format
    )
```

Code 4. Python script for get-all-devices Azure Function.

Code 4 begins by importing the necessary modules. The `logging` module is used to capture log messages, which are essential for monitoring and debugging purposes. For example, the command `logging.info('Get a list of all the existed devices :: ')` logs an informational message to track the function's activity. The `json` module enables the function to parse and generate JSON data, facilitating the conversion of SQL rows into JSON format and preparing the HTTP response. The `azure.functions` module provides Azure Function-specific classes and methods, including `func.HttpRequest` for handling HTTP requests and `func.HttpResponse` for generating HTTP responses. The function converts the SQL query results, held in `getAllDevices`, into a list of Python dictionaries using the `to_json` method. This list is then serialized into a JSON string using `json.dumps` and returned as an HTTP response with a success status code of `200` and a MIME type of `application/json`.

The `function.json` configuration file defines how the Azure Function interacts with HTTP requests and the SQL database. It specifies that the code file to execute is `__init__.py` and sets up

three bindings: an HTTP trigger, an HTTP response, and an SQL input binding. The HTTP trigger binding, defined as `httpTrigger`, listens for HTTP GET requests and is referenced in the code by the name `req`. This trigger has an authorization level of `function`, which requires a key for access, ensuring an additional layer of security. The HTTP response binding, defined as `http`, sends the function's return value as the HTTP response to the client. The SQL input binding, named `getAllDevices`, executes the SQL query `select [id], [name], [description], [created_at], [updated_at] from dbo.Device` to retrieve all device records from the database. This query's results are processed in the Python code. The configuration also uses the connection string setting `SqlConnectionString` to securely connect to the database.

The complete `function.json` configuration file for the `get-all-devices` Azure Function is as follows:

```
{
  "scriptFile": "__init__.py",
  "bindings": [
    {
      "authLevel": "function",
      "type": "httpTrigger",
      "direction": "in",
      "name": "req",
      "methods": [
        "get"
      ]
    },
    {
      "type": "http",
      "direction": "out",
      "name": "$return"
    },
    {
      "type": "sql",
      "direction": "in",
      "name": "getAllDevices",
      "commandText": "select [id], [name], [description],
[created_at], [updated_at] from dbo.Device",
      "commandType": "text",
      "connectionStringSetting": "SqlConnectionString"
    }
  ] }
```

Code 5. Configuration file for get-all-devices Azure Function.

The `get-all-devices` Azure Function retrieves all device records from an SQL database and returns them in JSON format via an HTTP GET request. The Python code handles the request processing and JSON conversion, while the configuration file ensures seamless interaction with the HTTP and SQL bindings. This setup is an efficient and secure method for obtaining a comprehensive overview of all devices stored in the database.

3.2.2 Get All Temperature

The Azure Function `get-all-temp` retrieves all temperature readings along with their associated device IDs from the SQL database table `dbo.Temperature` when it receives an HTTP GET request. This function is implemented in Python and comprises two main components: a Python code file (`__init__.py`) and a configuration file (`function.json`). Upon receiving a GET request, the function executes an SQL query to fetch all temperature data, converts the results into JSON format, and returns the JSON data along with a success status code (`200`).

The Python code (`__init__.py`) handles HTTP requests, interacts with the SQL database, and returns the query results in JSON format. It imports necessary modules such as `logging` for capturing log messages, `json` for handling JSON data, and `azure.functions` for utilizing Azure Function-specific classes and methods. The code logs a message indicating the retrieval of temperature readings, processes the SQL rows into JSON objects, and constructs an HTTP response containing the data. The Python code for this function is stated below.

```
import logging
import json
import azure.functions as func

def main(req: func.HttpRequest, allTemperatures: func.SqlRowList) ->
func.HttpResponse:
    # Log a message indicating the function is retrieving
temperature readings
    logging.info('Retrieving all temperature readings with device ID
and name.')

    # Convert each SQL row into a JSON object
    rows = list(map(lambda r: json.loads(r.to_json()),
allTemperatures))
```

```
    # Create an HTTP response with the JSON data and a status code
of 200
    return func.HttpResponse(
        json.dumps(rows),  # Convert the list of readings to a JSON
string
        status_code=200,   # HTTP status code for success
        mimetype="application/json"  # Indicate the response is in
JSON format
    )
```

Code 6. Python script for get-all-temp Azure Function

The configuration file, `function.json`, defines the function's interaction with HTTP requests and the SQL database. It specifies that the code file to execute is `__init__.py` and includes three bindings: an HTTP trigger, an HTTP response, and an SQL input binding. The HTTP trigger binding listens for GET requests and requires a key for access (`authLevel: function`), providing an additional layer of security. The input binding, named `allTemperatures`, executes the SQL query `select [value], [device_id] from dbo.Temperature`, which retrieves temperature data and their associated device IDs. The function sends the query results as an HTTP response using the output binding `$return`. The configuration file is presented below:

```
{
  "scriptFile": "__init__.py",
  "bindings": [
    {
      "authLevel": "function",
      "type": "httpTrigger",
      "direction": "in",
      "name": "req",
      "methods": [
        "get"
      ]
    },
    {
      "type": "http",
      "direction": "out",
      "name": "$return"
    },
    {
      "type": "sql",
      "direction": "in",
      "name": "allTemperatures",
      "commandText": "select [value], [device_id] from
dbo.Temperature",
      "commandType": "Text",
```

```
      "connectionStringSetting": "SqlConnectionString"
    }
  ]
}
```

Code 7. Configuration file for get-all-temp Azure Function

The configuration file ensures seamless interaction between the Azure Function, HTTP requests, and the SQL database. It defines the code file, the bindings, and the SQL query to execute. The SQL input binding retrieves temperature readings from the `dbo.Temperature` table, ensuring all records are fetched. The data is then processed in the Python code and returned as a JSON response to the client. In summary, the `get-all-temp` Azure Function efficiently retrieves temperature data for all devices from the SQL database. The Python code processes the request and converts the query results into JSON format, while the configuration file establishes the required bindings and specifies the SQL query. This function is an essential component for monitoring and analyzing temperature readings in the system.

3.2.3 Get Device By ID

The Azure Function `get-device-by-id` is designed to retrieve details of a specific device from the SQL database table `dbo.Device` using a device ID provided in an HTTP GET request. Written in Python, this function comprises two main components: a Python code file (`__init__.py`) and a configuration file (`function.json`). When a GET request is made with a device ID included in the URL, the function executes a SQL query to fetch the corresponding device data, converts the results into JSON format, and returns the JSON data along with a success status code (`200`). If the request does not include a device ID, the function immediately returns a `400 Bad Request` response indicating the missing parameter. The Python code for this function is presented below.

```
import logging  # Import the logging module to enable logging in the
function
import azure.functions as func  # Import Azure Functions' tools for
creating and managing functions
import json  # Import the JSON module to handle JSON data

def main(req: func.HttpRequest, getDeviceById: func.SqlRowList) ->
func.HttpResponse:
    device_id = req.route_params['id']  # Extract the device ID from
the route parameters
```

```
    if not device_id:  # Check if the device ID is not specified
        return func.HttpResponse(
            "Device ID not specified in the route.",  # Return an
error message if the device ID is missing
            status_code=400  # HTTP status code 400 indicates a bad
request
        )

    logging.info(f"Device by ID trigger :: ID: {device_id}")  # Log
an informational message with the device ID
    row = list(map(lambda r: json.loads(r.to_json()),
getDeviceById))  # Convert the SQL row(s) to a JSON object
    return func.HttpResponse(
        json.dumps(row),  # Convert the list of JSON objects to a
JSON string
        status_code=200,  # HTTP status code 200 indicates success
        mimetype="application/json"  # Specify that the response
content is in JSON format
    )
```

Code 8. Python script for the get-device-by-id Azure Function

This Python code includes necessary imports, such as `logging` for capturing log messages, `json` for processing JSON data, and `azure.functions` for working with Azure Function-specific tools. Upon receiving a request, the function checks if the device ID is present in the route parameters. If not, it returns a `400 Bad Request` response with an error message. When the device ID is provided, the function logs the ID being requested, executes the SQL query through the `getDeviceById` binding, converts the query results into a JSON object, and sends the results back as an HTTP response with a `200` success status code and a `application/json` MIME type.

The `function.json` configuration file defines how the Azure Function interacts with HTTP requests and the SQL database. The configuration specifies that the code file to execute is `__init__.py`, and it includes three bindings: an HTTP trigger, an SQL input binding, and an HTTP output binding. The full configuration file is presented below.

```
{
  "scriptFile": "__init__.py",
  "bindings": [
    {
      "authLevel": "function",
      "type": "httpTrigger",
      "direction": "in",
      "name": "req",
      "methods": [
        "get"
```

```
      ],
      "route": "get-device-by-id/{id}"
    },
    {
      "name": "getDeviceById",
      "type": "sql",
      "direction": "in",
      "commandText": "select [id], [name], [description],
[created_at], [updated_at] from dbo.Device where id = {id}",
      "commandType": "Text",
      "connectionStringSetting": "SqlConnectionString"
    },
    {
      "type": "http",
      "direction": "out",
      "name": "$return"
    }
  ]
}
```

Code 9. Configuration file for the get-device-by-id Azure Function

The configuration file defines how the function interacts with the HTTP request and the SQL database. The HTTP trigger listens for GET requests at the route `get-device-by-id/{id}`, where `{id}` represents the device ID passed in the URL. The `authLevel` is set to `function`, requiring an access key for additional security. The SQL input binding, named `getDeviceById`, executes a SQL query to fetch device data from the `dbo.Device` table using the provided device ID. This ensures that only the requested device data is retrieved. Finally, the HTTP output binding, named `$return`, sends the response back to the client in JSON format.

In summary, the `get-device-by-id` Azure Function efficiently retrieves the details of a specified device by ID. The Python code processes the request, performs the SQL query, and returns the results as a JSON response, while the configuration file ensures seamless integration with HTTP and SQL bindings.

3.2.4 Device Temperatures By ID

The Azure Function `device-temperatures-by-id` retrieves temperature readings for a specific device from the SQL database table `dbo.Temperature` when an HTTP GET request is made with a

device ID. This function is implemented in Python and consists of two main components: a Python code file (`__init__.py`) and a configuration file (`function.json`). Upon receiving a GET request with a device ID in the URL, the function executes a SQL query to fetch all temperature readings associated with that device, orders the results by the creation date in descending order, converts the query results into JSON format, and returns the JSON data with a success status code (`200`). If the request does not include a device ID, the function returns a `400 Bad Request` response. The Python code for this function is as follows:

```
import logging  # Enables logging in the function
import azure.functions as func  # Imports Azure Functions tools
import json  # Handles JSON data

def main(req: func.HttpRequest, deviceTemperatures: func.SqlRowList)
-> func.HttpResponse:
    device_id = req.route_params['id']  # Extracts device ID from
route parameters
    if not device_id:  # Checks if device ID is specified
        return func.HttpResponse(
            "Device ID not specified in the route.",  # Error mes-
sage if device ID is missing
            status_code=400  # HTTP status code 400 indicates a bad
request
        )

    logging.info(f"Device by ID trigger :: ID: {device_id}")  # Logs
the device ID being queried
    row = list(map(lambda r: json.loads(r.to_json()),
deviceTemperatures))  # Converts SQL rows to JSON objects
    return func.HttpResponse(
        json.dumps(row),  # Converts the list of JSON objects to a
JSON string
        status_code=200,  # HTTP status code 200 indicates success
        mimetype="application/json"  # Specifies that the response
content is in JSON format
    )
```

Code 10. Python code for the device-temperatures-by-id Azure Function.

This Python code includes imports for logging, JSON handling, and Azure Functions tools. It begins by extracting the device ID from the URL route parameters. If the device ID is missing, the function immediately responds with a `400 Bad Request` response and an appropriate error message. When the device ID is provided, the function logs the request with the `logging.info` statement, executes the SQL query through the `deviceTemperatures` binding, converts the query results into JSON format, and sends the data back as an HTTP response with a success status code (`200`). The logging

statement `logging.info(f"Device by ID trigger :: ID: {device_id}")` helps in debugging and tracking the requests.

The `function.json` configuration file defines how the Azure Function interacts with HTTP requests and the SQL database. The configuration specifies the code file to execute as `__init__.py` and includes three bindings: an HTTP trigger, an SQL input binding, and an HTTP output binding. The complete configuration file is stated below.

```
{
  "scriptFile": "__init__.py",
  "bindings": [
    {
      "authLevel": "function",
      "type": "httpTrigger",
      "direction": "in",
      "name": "req",
      "methods": [
        "get"
      ],
      "route": "device-temperatures-by-id/{id}"
    },
    {
      "name": "deviceTemperatures",
      "type": "sql",
      "direction": "in",
      "commandText": "select [value], [created_at] from dbo.Tempera-
ture where device_id = {id} order by [created_at] desc",
      "commandType": "Text",
      "connectionStringSetting": "SqlConnectionString"
    },
    {
      "type": "http",
      "direction": "out",
      "name": "$return"
    }
  ]
}
```

Code 11. Configuration file for the device-temperatures-by-id Azure Function

This configuration file specifies that the Azure Function is triggered by HTTP GET requests at the route `device-temperatures-by-id/{id}`, with `{id}` representing the device ID. The `authLevel` is set to `function`, which requires an access key for security. The SQL input binding, named `deviceTemperatures`, executes a query to fetch all temperature readings from the

dbo.Temperature table where the device_id matches the provided ID. The results are ordered by the created_at field in descending order, ensuring that the most recent readings are listed first. The HTTP output binding, named $return, sends the response back to the client in JSON format. In summary, the device-temperatures-by-id Azure Function efficiently retrieves temperature data for a specific device by its ID, orders the results, and returns them in JSON format. The Python code handles the request processing and data conversion, while the configuration file ensures seamless interaction with HTTP and SQL bindings, enabling a robust and secure endpoint.

3.2.5 Device Entry

The Azure Function device-entry enables users to add a new device entry to the SQL database table dbo.Device through an HTTP POST request. Upon receiving a POST request with JSON data, the function parses the data, logs the request, creates a SQL row from the data, and inserts it into the database. If successful, it responds with the inserted device details in JSON format and a 201 Created status code, indicating successful creation. If an error occurs, such as invalid JSON input, it returns a 400 Bad Request status code with an error message.

The function imports the logging, azure.functions as func, and json modules. The logging module is used for monitoring and debugging, the azure.functions module provides Azure-specific functionality for handling HTTP requests and SQL bindings, and the json module facilitates parsing and generating JSON data. The main function, main(req, deviceEntry), processes the incoming HTTP request (req) and interacts with the SQL database using the deviceEntry output binding.

The function begins by logging the request body and attempts to parse it as JSON. If parsing fails, an error is logged, and the function returns a 400 Bad Request response. Valid JSON data is validated and converted into a SqlRow object, which is then assigned to the deviceEntry binding for insertion into the database. The SqlRow object's properties are serialized into a JSON response and returned with a 201 Created status code to indicate success. If validation fails, the function responds with a 400 Bad Request error message, ensuring clear feedback to the client.
The function.json file configures the HTTP and SQL bindings. An httpTrigger with admin authorization listens for POST requests, while an HTTP output binding sends responses to the client.

The SQL output binding inserts data into the `dbo.Device` table using a connection string defined in `SqlConnectionString`. These configurations ensure secure and seamless interaction between the function and the database.

The complete implementation of the `device-entry` Azure Function is provided in APPENDIX 1 and APPENDIX 2. The code in APPENDIX 1 implements the core logic of the function, handling the HTTP request, parsing JSON data, validating input, and inserting a new device entry into the database. APPENDIX 2 contains the `function.json` configuration file, which defines the HTTP and SQL bindings. Together, they demonstrate a robust and scalable implementation for managing device entries in the SQL database.

3.2.6 Temperature Entry By ID

The Azure Function `temp-entry-by-id` is designed to add a new temperature entry associated with a specific device to an SQL database table via an HTTP POST request. When a request is received, the function logs the incoming request body for debugging purposes. It then extracts the device ID from the URL route parameters. If the device ID is not present, the function immediately returns a `400 Bad Request` response indicating that the device ID is missing.

Once the device ID is successfully extracted, the function logs this information and parses the JSON data from the request body. This parsed data is augmented with the device ID to associate the temperature entry with the specified device. If parsing the JSON data fails, the function logs an error and returns a `400 Bad Request` response indicating the parsing issue.

If the request body is successfully parsed and confirmed to be a dictionary, the function creates an `SqlRow` object using the data. This object is then set to the `tempEntry` output binding, representing the SQL database table. The `SqlRow` object is converted to a dictionary and subsequently serialized to a JSON string for inclusion in the response.

Finally, if all processing is successful, the function returns a `201 Created` response with the inserted temperature entry details in JSON format. If validation of the JSON data fails at any point, the function returns a `400 Bad Request` response indicating that the temperature entry could not be

created.The complete implementation of the Azure Function and its configuration are included in APPENDIX 3 and APPENDIX 4, respectively.

3.2.7 Edit device By ID

The Azure Function edit-device-by-id is designed to manage the updating of device entries in a SQL database. It serves as a backend endpoint that responds to HTTP PUT requests, processes the request data, retrieves existing device records, updates the records with new data, and saves the updated information back to the database. The primary goal of this function is to provide a seamless mechanism for updating device entries in the database via an HTTP endpoint. When a client sends a request to update a device, the function retrieves the existing device data, updates it with the new data provided in the request, and saves the updated data back to the database. The implementation also includes robust error handling to ensure appropriate responses are returned to the client in case of errors.

The configuration file for this Azure Function is structured to enable HTTP PUT requests to update device entries in a SQL database. The scriptFile is set to __init__.py, which specifies the main script file for the function. The bindings section defines the triggers and bindings. The authLevel is set to admin, requiring high-level permissions for access. The httpTrigger listens for PUT requests at the route edit-device-by-id/{id}, where {id} is the device ID passed as a route parameter. The trigger receives the request data and assigns it to req, representing the input for the function. The HTTP response is defined as an output binding with the name $return, meaning the return value of the function is sent back as the HTTP response to the client.

Additionally, the configuration file includes two SQL bindings. The first, editDevice, is an output binding that updates the dbo.Device table in the SQL database. It uses the connection string defined by SqlConnectionString to connect to the database. The second, getDevice, is an input binding that retrieves the current device data from the database. The SQL query “select [id], [name], [description], [created_at], [updated_at] from dbo.Device where id = {id}” is used to fetch the device details for the given ID. This data ensures that the device exists and merges new data with the existing data.

The function begins by importing necessary modules, including logging for capturing log messages, json for handling JSON data, DateTime for managing timestamps, and azure.functions for utilizing

Azure Function-specific classes and methods. The main function, main, accepts three parameters: req, which represents the incoming HTTP request; editDevice, an output binding for updating the device entry in the database; and getDevice, an input binding for retrieving existing device entries from the database.

Upon receiving an HTTP request, the function extracts the device ID from the route parameters. If the device ID is not provided, the function immediately returns a 400 Bad Request response, indicating that the device ID is missing. If the device ID is present, the function logs the attempt to edit the device using this ID and retrieves the existing device information from the getDevice parameter. The SQL rows retrieved are converted into a list of dictionaries. If the device is not found in the database, the function returns a 404 Not Found response.

Next, the function attempts to parse the JSON body of the request. If parsing fails, an error is logged, and a 400 Bad Request response is returned, indicating a failure to parse the JSON data. If the request body is successfully parsed and validated as a dictionary, the function updates the device information using a helper function, prepare_device_item. This helper function merges the existing device data with the new data provided in the request, ensuring that essential fields like id, created_at, and updated_at are properly managed. It also ensures that if certain fields, such as name or description, are not provided in the new data, they are retained from the existing data.

The updated device information is then converted into an SQL row and saved back to the database using the editDevice output binding. The updated device entry is serialized into a JSON response and returned to the client with a 201 Created status code. If the update process fails at any point, the function returns a 400 Bad Request response with an appropriate error message. The complete implementation of the Azure Function is detailed in APPENDIX 5, and the configuration file is included in APPENDIX 6.

3.2.8 Delete Device By ID

The Azure Function delete-device-by-id ensures the deletion of a device entry by interacting with the SQL database and providing a confirmation response back to the client. It is designed to delete a device entry from a SQL database based on the device ID provided in the HTTP request URL. The

function handles HTTP DELETE requests, processes the request to extract the device ID, interacts with the database to delete the specified device entry, and returns an appropriate response to the client.

The script begins by importing necessary modules: logging for capturing log messages, azure.functions for utilizing Azure Function-specific classes and methods, and json for handling JSON data. The primary function, main, takes two parameters: req, representing the incoming HTTP request, and deleteDevice, an input binding for interacting with the SQL database.

When the function is triggered by an HTTP DELETE request, it extracts the device_id from the route parameters in the request URL. If the device ID is not provided, the function immediately returns a 400 Bad Request response, indicating that the device ID is missing. If the device ID is present, the function logs the attempt to delete the device using this ID.

The function retrieves the device entry corresponding to the provided ID from the deleteDevice parameter, converting the SQL rows into a list of dictionaries using the to_json method. This list of dictionaries is then converted to a JSON string. Finally, the function returns a 200 OK response with the JSON string, indicating that the device entry has been successfully deleted.

This script ensures that the device ID is correctly extracted from the request, performs the deletion operation in the SQL database, and returns the appropriate response to the client, confirming the deletion of the device entry. The code for the delete-device-by-id Azure Function is presented below.

```
import logging
import azure.functions as func
import json
def main(req: func.HttpRequest, deleteDevice: func.SqlRowList) ->
func.HttpResponse:
    device_id = req.route_params['id']
    if not device_id:
        return func.HttpResponse(
            "Device ID not specified in the route.",
            status_code=400
        )
    logging.info(f"Device delete by ID trigger :: ID: {device_id}")
    row = list(map(lambda r: json.loads(r.to_json()), deleteDevice))
    return func.HttpResponse(
        json.dumps(row),
        status_code=200,
```

```
        mimetype="application/json"
    )
```

Code 12. Script for delete-device-by-id Azure Function.

The JSON configuration file, function.json, provides a RESTful API endpoint for deleting a device by its ID, making it part of a RESTful API implementation. The configuration file sets up an Azure Function to delete a device entry from an SQL database using an HTTP POST request. It includes three bindings: an HTTP trigger, an HTTP output, and an SQL input.

The scriptFile property indicates that the main function code is located in __init__.py. The HTTP trigger listens for HTTP POST requests at the route delete-device-by-id/{id}, where {id} is a route parameter representing the device to be deleted. This trigger requires administrative authorization (authLevel: admin), meaning it needs higher-level permissions to execute. The HTTP response output binding, named $return, sends the function's return value as the HTTP response to the client, providing feedback on the outcome of the delete operation.

The SQL input binding, named deleteDevice, executes an SQL command to delete the device entry from the dbo.Device table where the id matches the provided route parameter. The commandText specifies the SQL query, and commandType is set to text, indicating a straightforward SQL command. The connectionStringSetting uses SqlConnectionString to connect to the SQL database. The complete JSON configuration is provided in APPENDIX 7.

3.3 Endpoint testing with Postman App

This section will outline the process of using Postman for testing the API endpoints developed for this project. Postman is a popular application designed for API testing and debugging. Its user-friendly interface allows users to send different types of HTTP requests and review responses efficiently, making it an essential tool for verifying API functionality. (GeeksforGeeks 2024d.)

This tool eliminates the need for extensive coding when creating, organizing, or automating API tests (freeCodeCamp 2024). Figure 10 provides a visualization of the process of utilizing this app. A detailed explanation of how Postman was utilized for this thesis is also provided below. To start the process, Postman was downloaded and installed from the official website. After launching the

application, a new workspace was created specifically for this project, named " IOT Backend Management API" This workspace helped organize all the requests related to the project in one place, making it easy to manage and navigate through different API tests.

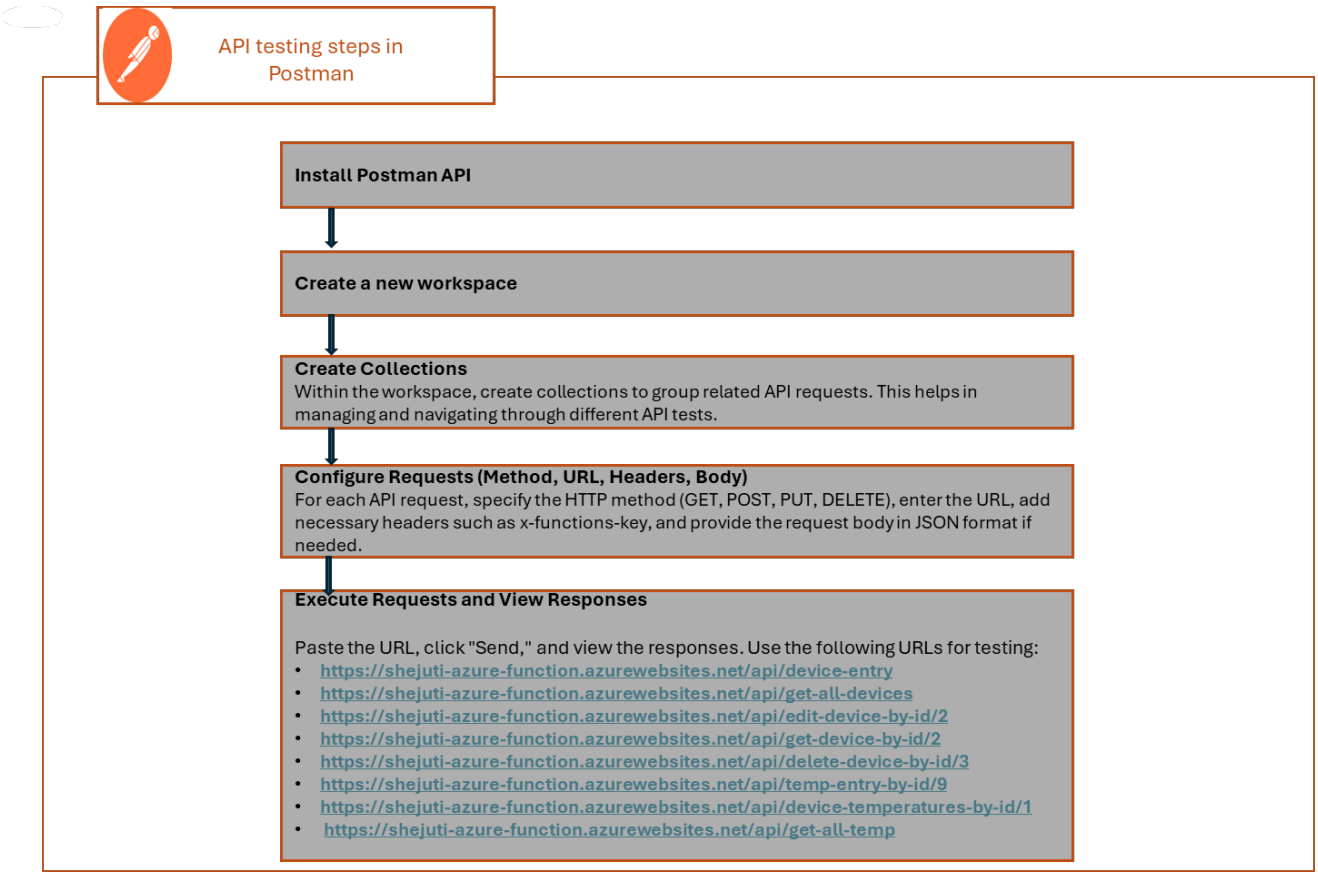


FIGURE 10. API testing steps in Postman APP

Within this workspace, collections were created to group related API requests. Separate collections were made for device entry, device update, device deletion, and retrieving device information. Each request was configured with the appropriate HTTP method (such as GET, POST, PUT, DELETE), URL, headers, and body data as needed. This setup was straightforward: the HTTP method was specified, and the URL was pasted into Postman. For authentication, the x-functions-key was added in the headers.

Figure 11 below, shows the "Postman Interface for API Testing." The Postman application is set up with various API requests under the "IOT Backend Management API" collection. To give a specific example, a POST request is configured for the "device-entry" endpoint with the URL `https://shejuti-azure-function.azurewebsites.net/api/device-entry`. The request body contains JSON data to create a new device entry with the name "device no 11", ID "11",

and description "enter device number eleven". After clicking "Send," Postman displays the response from the server below, quickly verifying if the API is working as expected. The response indicates a successful creation with status `201 Created` and the returned JSON data matching the input.

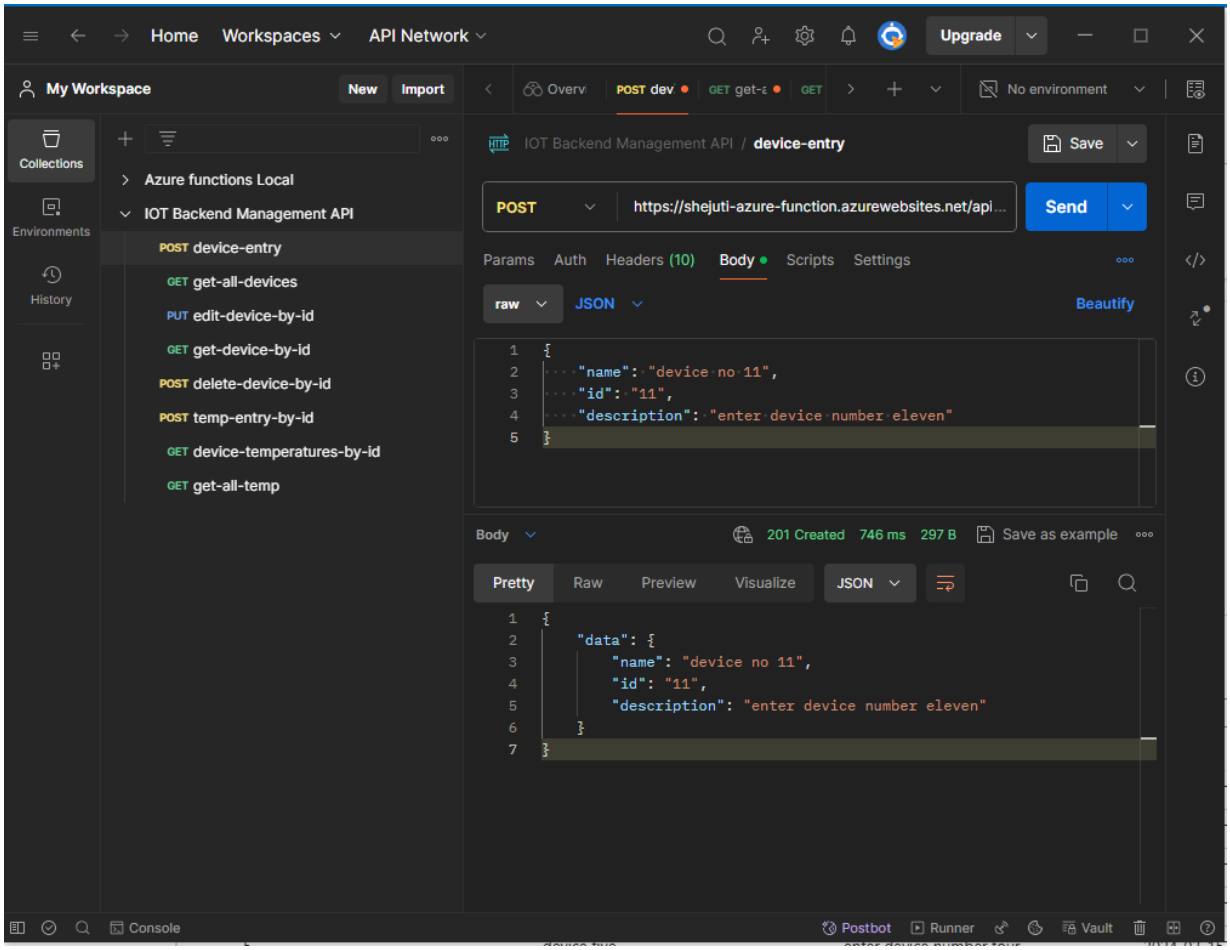


FIGURE 11. Postman Interface for API Testing

Similarly, to retrieve all devices, a GET request was used with the URL `https://shejuti-azure-function.azurewebsites.net/api/get-all-devices`, including the x-functions-key in the headers. For updating a device, a PUT request was configured to `https://shejuti-azure-function.azurewebsites.net/api/edit-device-by-id/1`, and for deleting a device, a POST request to https://shejuti-azure-function.azurewebsites.net/api/delete-device-by-id/3.

Additionally, retrieving the URLs for these endpoints was made easy by using Visual Studio Code with the Azure extension. By navigating to the function app in Visual Studio Code, right-clicking on the specific function, and selecting "Copy Function URL," the correct URL for each API endpoint was quickly acquired as shown in Figure 12.

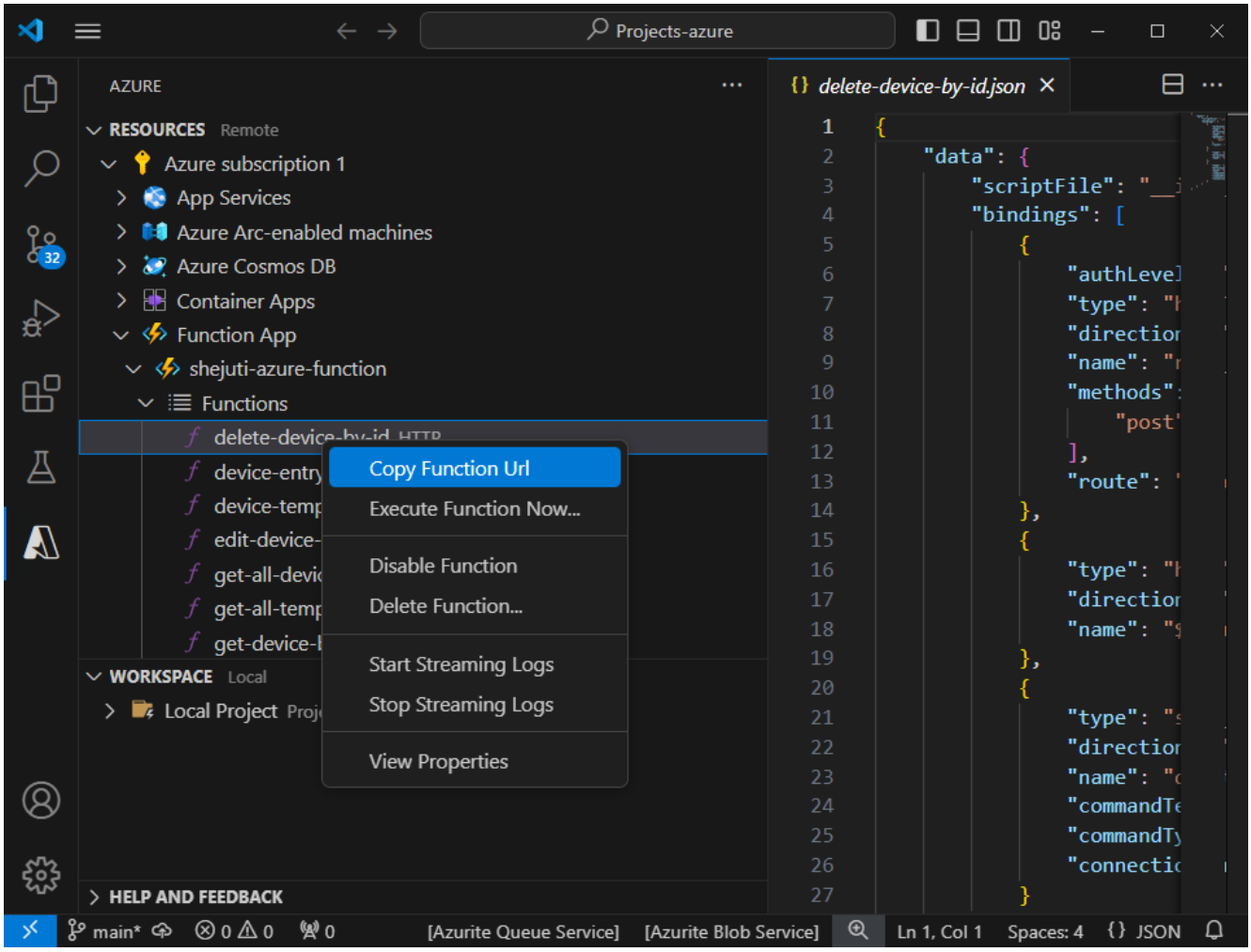


FIGURE 12. Retrieving URL for endpoints test from Visual Studio Code

Testing API endpoints without Postman can be more complicated and take longer. For example, using cURL requires crafting commands for each request and interpreting the responses manually, which makes the process both difficult and error-prone. (StackShare.)

Similarly, building custom client applications involves preparing an environment for development, creating scripts to handle HTTP requests and responses, and troubleshooting these scripts to ensure they work as intended. (Hamilton 2024a.)

An alternative method can be plugins or automated testing tools using IDEs. However, this requires downloading and configuring additional software, designing comprehensive test cases, and carefully

managing the test execution process. These methods often take more time and are more complex compared to the streamlined workflows offered by Postman. (Hamilton 2024b.)

Postman made API testing much easier for this project. It enabled efficient testing and verification of API endpoints by inputting URLs, setting headers, and executing requests with minimal effort. Furthermore, Postman's powerful features, including automation, environment management, and collaboration tools, proved essential for ensuring the reliability and functionality of APIs. (Postman Learning 2024; Hamilton 2024b; Hamilton 2024b.)

4 CONCLUSION

The main goal of this thesis was to design and build an Internet of Things (IoT) backend service using Microsoft Azure, with a focus on creating a RESTful API in Python to manage and process data stored in a SQL database. The data used in this project was generated as a demo, simulating the type of data typically produced by IoT devices. The project aimed to develop a scalable and secure backend system capable of efficiently handling device data, validating it, and securely storing it in the SQL database, while also ensuring that the system is user-friendly.

To achieve these objectives, Microsoft Azure's cloud platform was employed. Serverless computing via Azure Functions facilitated efficient infrastructure management, while the SQL database provided secure data storage. The RESTful API was rigorously tested using Postman, verifying the accuracy and functionality of the endpoints. The results demonstrated that the backend service effectively met the project's goals, with the API performing reliably and securely managing the data.

However, challenges were encountered during the development process. While Postman proved effective for manual testing, the absence of automated testing increased the effort and time required. The need to keep the Azure SQL database active during testing also led to increased costs, particularly for extended testing periods. Additionally, using an Azure Function key for API security exposed potential vulnerabilities, as the key could be intercepted by unauthorized users, posing a security risk.

In addressing these challenges, several improvements can be considered for future work. Implementing OAuth 2.0 for API authentication would provide better protection by enabling limited access without exposing credentials. Azure API Management could further enhance security by enforcing stricter security policies and managing API keys more effectively. Additionally, integrating the RESTful API and backend service with Azure IoT Hub would enable real-time processing of data from actual IoT devices, enhancing the system's ability to manage and analyze live IoT data.

From this research and development process, several valuable insights were gained. The work highlighted the importance of scalable and secure solutions in cloud environments and underscored the need for robust testing strategies and strong security measures. Automated testing tools and deeper exploration of advanced authentication methods emerged as areas for growth, both for this project and for future IoT backend systems.

The objectives of this thesis were largely achieved, as the developed system successfully demonstrated the creation of a scalable, secure, and user-friendly RESTful API for managing IoT device data in Azure SQL. However, there is room for improvement in automating testing processes and implementing advanced security measures. Addressing these aspects in future iterations could enhance the system's overall robustness and adaptability. This project establishes a strong foundation for further advancements and provides practical recommendations for developers aiming to build scalable and secure IoT backend solutions in cloud environments.

REFERENCES

Anaconda Documentation. 2024. *Anaconda 2024.10-1 (Oct 23, 2024).* Available at: https://docs.anaconda.com/anaconda/release-notes/#anaconda-2024-10-1-oct-23-2024 . Accessed on 17 December 2024.

Apidog. 2024a. *GET vs POST Request: The Difference Between HTTP Methods*. Apidog Blog. Available at: https://apidog.com/blog/get-vs-post-request-the-difference-between-http-methods/ . Accessed on 22 November 2024.

Apidog. 2024b. *An Ultimate Guide to HTTP POST Request Method*. Available at: https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Welcome.html . Accessed on 24 November 2024.

AWS Documentation. 2025a. *What is Amazon Relational Database Service (Amazon RDS)?*. Available at: https://apidog.com/articles/http-post-request/. Accessed on 06 February 2025.

AWS Documentation. 2025b. *Using AWS Lambda with Amazon RDS*. Available at: https://docs.aws.amazon.com/lambda/latest/dg/services-rds.html. Accessed on 06 February 2025.

AWS Repost. 2025. *Lambda to DB connectivity - best practices*. Available at: https://repost.aws/questions/QULXSqEPGbQx6qiyBa1D1Udg/lambda-to-db-connectivity-best-practices. Accessed on 06 February 2025.

Bahga, A. & Madisetti, V. 2014. *Internet of Things: A Hands-On Approach*. 1st Edition. Blacksburg: VPT Press. Available at: https://books.google.co.in/books/about/Internet_of_Things.html?id=JPKGBAAAQBAJ&printsec=frontcover&source=kp_read_button&redir_esc=y#v=onepage&q&f=false. Accessed on 22 November 2024.

Fielding, R. T. 2000. *Architectural Styles and the Design of Network-based Software Architectures.* Irvine: University of California. Doctoral Dissertation. Available at: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm . Accessed 1 June 2024.

FreeCodeCamp. 2022. *HTTP Request Methods – Get vs Put vs Post Explained with Code Examples.* Available at: https://www.freecodecamp.org/news/http-request-methods-explained/ . Accessed on 24 November 2024.

FreeCodeCamp. 2023. *How to Use an API with Postman – A Step-by-Step Guide.* Available at: https://www.freecodecamp.org/news/how-to-use-an-api-with-postman. Accessed on 24 November 2024.

GeeksforGeeks. 2024a. *Internet of Things with Python*. Available at: https://www.geeksforgeeks.org/internet-of-things-with-python/ . Accessed on 26 May 2024.

GeeksforGeeks. 2024b. *What is Microsoft Azure?.* Available at: https://www.geeksforgeeks.org/what-is-microsoft-azure/ . Accessed on 21 November 2024.

GeeksforGeeks. 2024c. *What is SQL?.* Available at: https://www.geeksforgeeks.org/what-is-sql/ . Accessed on 22 November 2024.

GeeksforGeeks. 2024d. *How to test an API using Postman?*. Available at: https://www.geeksforgeeks.org/how-to-test-an-api-using-postman/. Accessed on 22 November 2024.

Gupta, L. 2023. *Difference between JSON and XML.* Available at: https://restfulapi.net/json-vs-xml/ . Accessed on 21 November 2024.

Gupta, L. 2024. *REST Architectural Constraints*. Available at: https://restfulapi.net/rest-architectural-constraints/ . Accessed on 21 November 2024.

Hamilton, T. 2024a. *API Testing Tutorial: What is API Test Automation?*. Available at: https://www.guru99.com/api-testing.html . Accessed on 01 June 2024.

Hamilton, T. 2024b. *Automation Testing*. Available at: https://www.guru99.com/automation-testing.html. Accessed on 01 June 2024.

Host4Geeks. 2023. *Decoding the Web: What is the Function of the HTTP GET Message*. Available at: https://host4geeks.com/blog/decoding-the-web-what-is-the-function-of-the-http-get-message/ . Accessed on 21 November 2024.

JetBrains. 2024. Focus on code and data. Available at: https://www.jetbrains.com/pycharm/#:~:text=Focus%20on%20code,of%20the%20rest . Accessed on 17 December 2024.

JSON org. *Introducing JSON*. Available at: https://www.json.org/json-en.html . Accessed on 22 November 2024.

MDN Web Docs. 2024a. *GET*. Available at: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/GET . Accessed on 21 November 2024.

MDN Web Docs. 2024b. *POST*. Available at: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST . Accessed on 21 November 2024.

MDN Web Docs. 2024c. *Working with JSON*. Available at: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON . Accessed on 22 November 2024.

Microsoft Azure. 2024a. *Azure Functions HTTP trigger*. Available at: https://learn.microsoft.com/en-us/azure/azure-functions/functions-bindings-http-webhook-trigger . Accessed on 21 November 2024.

Microsoft Azure. 2024b. *What is SaaS?*. Available at: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-saas/ . Accessed on 21 November 2024.

Microsoft Azure. 2024c. *What is Azure*?. Available at: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-azure/ . Accessed 26 May 2024.

Microsoft Learn. 2023. *Azure Functions overview*. Available at: https://learn.microsoft.com/en-us/azure/azure-functions/functions-overview . Accessed on 26 May 2024.

Microsoft Learn. 2024a. *Python version*. Available at: https://learn.microsoft.com/en-us/azure/azure-functions/functions-reference-python?utm_source=chatgpt.com&tabs=get-started%2Casgi%2Capplication-level&pivots=python-mode-configuration#python-version . Accessed on 26 August 2024.

Microsoft Learn. 2024b. *Access key authorization*. Available at: https://learn.microsoft.com/en-us/azure/azure-functions/functions-bindings-http-webhook-trigger?tabs=in-process#authorization-keys . Accessed on 26 May 2024.

Microsoft Learn. 2024c. *Install or update Core Tools*. Available at: https://learn.microsoft.com/en-us/azure/azure-functions/create-first-function-vs-code-python#install-or-update-core-tools . Accessed on 26 May 2024.

Microsoft Learn. 2024d. *Create a function in Azure with Python using Visual Studio Code*. Available at: https://learn.microsoft.com/en-us/azure/azure-functions/create-first-function-vs-code-python#install-or-update-core-tools . Accessed on 26 May 2024.

Microsoft Learn. 2024e. *What is Azure Internet of Things (IoT)?*. Available at: https://learn.microsoft.com/en-us/azure/iot/iot-introduction . Accessed on 26 May 2024.

Microsoft Learn. 2024f. *What is Azure SQL Database?*. Available at: https://learn.microsoft.com/en-us/azure/azure-sql/database/sql-database-paas-overview?view=azuresql . Accessed 26 May 2024.

Microsoft Learn. 2024g. *What is the Azure portal?*. Available at: https://learn.microsoft.com/en-us/azure/azure-portal/azure-portal-overview . Accessed on 21 November 2024.

Microsoft Learn. 2024h. *Develop Azure Functions by using Visual Studio Code.* Available at: https://learn.microsoft.com/en-us/azure/azure-functions/functions-develop-vs-code?utm_source=chatgpt.com&tabs=node-v4%2Cpython-v2%2Cisolated-process%2Cquick-create&pivots=programming-language-python . Accessed 27 November 2024.

Microsoft Learn. 2024i. *Introduction to securing Azure Functions with App Keys*. Available at: https://learn.microsoft.com/en-us/azure/azure-functions/security-concepts . Accessed 27 November 2024.

Microsoft Learn. 2024j. *Azure SQL bindings for Azure Functions overview*. Available at: https://learn.microsoft.com/en-us/azure/azure-functions/functions-bindings-azure-sql?tabs=isolated-process%2Cextensionv4&pivots=programming-language-python. Accessed 06 February 2025.

Microsoft Learn. 2025. *Azure Hybrid Benefit for Windows Server*. Available at: https://learn.microsoft.com/en-us/windows-server/get-started/azure-hybrid-benefit?tabs=azure . Accessed 05 February 2025.

Parmar P. 2023. *A Comprehensive Guide to Database for IoT: All You Need to Know*. Go Roboted. Available at: https://goroboted.com/a-comprehensive-guide-to-database-for-iot-all-you-need-to-know/. Accessed on 21 November 2024.

Postman Learning. 2023. *Postman documentation overview.* Available at: https://learning.postman.com/docs/introduction/overview/ . Accessed on 26 May 2024.

Postman Learning. 2024. *API collaboration features in Postman*. Available at: https://learning.postman.com/docs/collaborating-in-postman/api-collaboration-features/?utm_source=chatgpt.com . Accessed on 21 November 2024.

Postman. *Download postman*. Available at: https://www.postman.com/downloads/. Accessed on 21 February 2024.

Python org. *Download the latest version for Windows*. Available at: https://www.python.org/downloads/. Accessed on 22 November 2024.

RapidAPI. 2024a. *GET – What is the GET Method?*. RapidAPI Blog. Available at: https://rapidapi.com/blog/api-glossary/get/. Accessed on 22 November 2024.

RapidAPI. 2024b. *POST – What is the POST Method?*. RapidAPI Blog. Available at: https://rapidapi.com/blog/api-glossary/post/. Accessed on 22 November 2024.

REST API Tutorial. *The Six Constraints*. Available at: https://www.restapitutorial.com/introduction/restconstraints . Accessed on 21 November 2024.

REST API Tutorial. *What is REST?*. Available at: https://www.restapitutorial.com/introduction/whatisrest . Accessed on 21 November 2024.

Rose, K., Eldridge, S. & Chapin, L. 2015. *The Internet of Things: An Overview – Understanding the Issues and Challenges of a More Connected World*. Internet Society. Available at: https://www.internetsociety.org/wp-content/uploads/2017/08/ISOC-IoT-Overview-20151221-en.pdf . Accessed on 26 May 2024.

StackShare. Curl vs postman: *What are the differences?* https://stackshare.io/stackups/curl-vs-postman. Accessed: 01 June 2024.

Stephens, L. *GET vs POST*. Available at: https://www.akto.io/academy/get-vs-post . Accessed on 22 November 2024.

Tharindu, K. 2024. *What Are REST APIs and How Can You Master Them?*. DTK2Globle Blog. Available at: https://blog.dtk2globle.com/what-are-rest-apis . Accessed on 22 November 2024.

Tuple. *REST (Representational State Transfer)*. Available at: https://www.tuple.nl/knowledge-base/rest-representational-state-transfer . Accessed on 21 November 2024.

W3Schools. 2024a. *HTTP Request Methods*. Available at: https://www.w3schools.com/tags/ref_http-methods.asp . Accessed on 22 November 2024.

W3Schools. 2024b. *Introduction to SQL*. Available at: https://www.w3schools.com/sql/sql_intro.asp . Accessed on 21 November 2024.

Implementation of device-entry Azure Function

```
import logging
import azure.functions as func
import json

def main(req: func.HttpRequest, deviceEntry: func.Out[func.SqlRow])
-> func.HttpResponse:
   logging.info('Device entry trigger :: request body %s',
req.get_json())

   try:
      req_body = req.get_json()
   except ValueError as e:
      logging.error('Error parsing JSON data: %s', str(e))
      return func.HttpResponse("Failed to parse JSON data", sta-
tus_code=400)

   if req_body and isinstance(req_body, dict):
      # Create a SqlRow object manually from the request body
      sql_row = func.SqlRow(**req_body)

      # Set the deviceEntry using the SqlRow
      deviceEntry.set(sql_row)

      # Convert the SqlRow data to a dictionary
      device_entry_dict = sql_row.__dict__

      # Convert the dictionary to JSON
      device_entry_json = json.dumps(device_entry_dict)
      return func.HttpResponse(

         device_entry_json,
         status_code=201,
         mimetype="application/json"
      )
   else:
      return func.HttpResponse(
         "Failed to enter the device!",
         status_code=400
      )
```

Configuration file for device-entry Azure Function

```
{
  "scriptFile": "__init__.py",
  "bindings": [
    {
      "authLevel": "admin",
      "type": "httpTrigger",
      "direction": "in",
      "name": "req",
      "methods": [
        "post"
      ]
    },
    {
      "type": "http",
      "direction": "out",
      "name": "$return"
    },
    {
      "type": "sql",
      "direction": "out",
      "name": "deviceEntry",
      "commandText": "dbo.Device",
      "connectionStringSetting": "SqlConnectionString"
    }
  ]
}
```

Implementation of temp-entry-by-id Azure Function

```
import logging  # Used for logging information and errors
import azure.functions as func  # Azure Functions SDK for handling
HTTP requests and interacting with output bindings
import json  # Used for parsing and generating JSON data
import datetime  # Standard library for working with date and time

def main(req: func.HttpRequest, tempEntry: func.Out[func.SqlRow]) ->
func.HttpResponse:
    logging.info('Temperature entry trigger :: request body %s',
req.get_json())  # Log the incoming request body for debugging
purposes

    device_id = req.route_params['id']  # Extract the device ID from
the URL route parameters
    if not device_id:
        # Return a 400 response if the device ID is not specified in
the route
        return func.HttpResponse(
            "Device ID not specified in the route.",
            status_code=400
        )

    logging.info(f"Device by ID trigger :: ID: {device_id}")  # Log
the extracted device ID

    try:
        req_body = req.get_json()  # Parse the request body as JSON
        req_body["device_id"] = device_id  # Add the extracted
device ID to the request body
    except ValueError as e:
        logging.error('Error parsing JSON data: %s', str(e))  # Log
an error and return a 400 response if JSON parsing fails
        return func.HttpResponse(
            "Failed to parse temperature JSON data",
            status_code=400
        )

    if req_body and isinstance(req_body, dict):  # Check if the
parsed JSON exists and is a dictionary
        sql_row = func.SqlRow(**req_body)  # Create an SqlRow object
from the request body
        tempEntry.set(sql_row)  # Set the tempEntry using the
created SqlRow
        device_entry_dict = sql_row.__dict__  # Convert the SqlRow
data to a dictionary
```

```
        device_entry_json = json.dumps(device_entry_dict)  # Convert
the dictionary to JSON

        return func.HttpResponse(
            device_entry_json,
            status_code=201,
            mimetype="application/json"
        )  # Return the inserted temperature entry as a JSON re-
sponse with a 201 status code
    else:
        return func.HttpResponse(
            "Failed to enter the device temperature!",
            status_code=400
        )  # Return a 400 response if the JSON validation fails
```

Configuration file for temp-entry-by-id Azure Function

The function.json configuration file defines the triggers, bindings, and database connections for the function. The scriptFile is set to __init__.py, which specifies the main script file for the function. The bindings section defines the triggers and bindings. The authLevel is set to function, indicating that the function requires an API key for authorization.

The httpTrigger binding specifies that the function is triggered by HTTP POST requests. It is configured to accept HTTP requests as input (direction: in) and is named req. The route parameter temp-entry-by-id/{id} is used to capture the device ID from the URL.

The HTTP response is configured as an output binding (direction: out) and is named $return. Additionally, the sql binding is configured to output data to an SQL database table named dbo.Temperature. This SQL binding uses the connection string specified by SqlConnectionString to connect to the database. The complete configuration file is as follows:

```
{
  "scriptFile": "__init__.py",
  "bindings": [
    {
      "authLevel": "function",
      "type": "httpTrigger",
      "direction": "in",
      "name": "req",
      "methods": ["post"],
      "route": "temp-entry-by-id/{id}"
    },
    {
      "type": "http",
      "direction": "out",
      "name": "$return"
    },
    {
      "type": "sql",
      "direction": "out",
      "name": "tempEntry",
      "commandText": "dbo.Temperature",
      "connectionStringSetting": "SqlConnectionString"
    }
  ]

}
```

Implementation of edit-device-by-id Azure Function

```
import logging
import json
import datetime
import azure.functions as func

def main(req: func.HttpRequest, editDevice: func.Out[func.SqlRow],
getDevice: func.SqlRowList) -> func.HttpResponse:
    device_id = req.route_params['id']
    if not device_id:
        return func.HttpResponse(
            "Device ID not specified in the route.",
            status_code=400
        )
    logging.info(f"Edit Device by ID :: {device_id}")
    row = list(map(lambda r: json.loads(r.to_json()), getDevice))
    if not row:
        return func.HttpResponse(
            "Device not found",
            status_code=404
        )
    try:
        req_body = req.get_json()
        logging.info('Update Device entry trigger :: request body is
%s', req_body)
    except ValueError as e:
        logging.error('Error parsing JSON data: %s', str(e))
        return func.HttpResponse(
            "Failed to parse JSON data",
            status_code=400
        )

   if req_body and isinstance(req_body, dict):
        edited = prepare_device_item(row[0], req_body)
        sql_row = func.SqlRow(**edited)
        editDevice.set(sql_row)
        device_entry_dict = sql_row.__dict__
        device_entry_json = json.dumps(device_entry_dict)
        return func.HttpResponse(
            device_entry_json,
            status_code=201,
            mimetype="application/json"
        )
    else:
        return func.HttpResponse(
            "Failed to update the device!",
            status_code=400)
```

```
 def prepare_device_item(old, new):
new["id"] = old["id"]
    new["created_at"] = old["created_at"]
    new["updated_at"] = datetime.datetime.utcnow().isoformat()
    if "name" not in new:
        new["name"] = old["name"]
    if "description" not in new:
        new["description"] = old["description"]
    return new
```

Configuration file for edit-device-by-id Azure Function

```
{
  "scriptFile": "__init__.py",
  "bindings": [
    {
      "authLevel": "admin",
      "type": "httpTrigger",
      "direction": "in",
      "name": "req",
      "methods": [
        "put"
      ],
      "route": "edit-device-by-id/{id}"
    },
    {
      "type": "http",
      "direction": "out",
      "name": "$return"
    },
    {
      "type": "sql",
      "direction": "out",
      "name": "editDevice",
      "commandText": "dbo.Device",
      "connectionStringSetting": "SqlConnectionString"
    },
    {
      "name": "getDevice",
      "type": "sql",
      "direction": "in",

      "commandText": "select [id], [name], [description], [cre-
ated_at], [updated_at] from dbo.Device where id = {id}",
      "commandType": "Text",
      "connectionStringSetting": "SqlConnectionString"
    }
  ]
}
```

Configuration File for delete-device-by-id Azure Function

```
{
  "scriptFile": "__init__.py",
  "bindings": [
    {
      "authLevel": "admin",
      "type": "httpTrigger",
      "direction": "in",
      "name": "req",
      "methods": [
        "post"
      ],
      "route": "delete-device-by-id/{id}"
    },
    {
      "type": "http",
      "direction": "out",
      "name": "$return"
    },
    {
      "type": "sql",
      "direction": "in",
      "name": "deleteDevice",
      "commandText": "Delete from dbo.Device where id = {id}",
      "commandType": "text",
      "connectionStringSetting": "SqlConnectionString"
    }
  ]
}
```