Implementing Serverless Functions in a Microservice Architecture

Introduction

This tutorial demonstrated how to implement a Function-as-a-Service (FaaS) using the Google Cloud Functions Framework. We will build an invoice generator that runs effectively "serverless" and integrate it into a Node.js Order Service.

The Architecture

We are building the following flow:

1. Order Service: Receives an order.
2. Order Service: Makes an HTTP call to the Invoice Function
3. Invoice Function: Generates a JSON response (simulating a PDF link) and logs the operation

Implementation

1. Creating the Function

We use `@google-cloud/functions/framework`. This library allows us to write standard cloud functions and run them locally for development.

Create a directory `functions/invoice-generator` and initialize a Node project.

File: `index.js`

```
const functions = require('@google-cloud/functions-framework');

// Register an HTTP function named 'generateInvoice'
functions.http('generateInvoice', async (req, res) => {
    // 1. Handle CORS (Important for direct browser access, though we call server-to-server)
    res.set('Access-Control-Allow-Origin', '*');

    if (req.method !== 'POST') {
        return res.status(405).send('Method Not Allowed');
    }

    try {
        const { orderId, total, customerName } = req.body;

        // 2. Simulate heavy processing (Wait 100ms)
        await new Promise(resolve => setTimeout(resolve, 100));

        console.log(`Generating Invoice for Order ${orderId}...`);

        // 3. Return the Result
        const response = {
            invoiceId: `INV-${orderId}`,
            status: 'generated',
            url: `https://storage.googleapis.com/invoices/${orderId}.pdf`
        };

        res.status(200).json(response);

    } catch (error) {
```

```
        res.status(500).json({ error: error.message });
    }
});
```

2.a Containerizing the Function

To integrate this into our local development environment (Docker Compose), we wrap the function in a container.

File: `Dockerfile`

```
FROM node:20-alpine
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
# Expose default port
EXPOSE 8080
# Use npx to run the framework CLI
CMD ["npx", "@google-cloud/functions-framework", "--target=generateInvoice"]
```

2.b Setting up Google Cloud CLI

While running the function in Docker is perfect for development, the ultimate goal of FaaS is to run in the cloud. To do this, we need to set up Google Cloud CLI.

Initialization and Login

```
gcloud init
```

Enabling required APIs

```
gcloud services enable cloudfunctions.googleapis.com
gcloud services enable cloudbuild.googleapis.com
gcloud services enable artifactregistry.googleapis.com
```

Deploying the Function

```
cd functions/invoice-generator

gcloud functions deploy generateInvoice \
--runtime nodejs20 \
--trigger-http \
--allow-unauthenticated
```

3. Triggering the Function from a Microservice

In our architecture, the `order-service` triggers the function. It treats the FaaS as just another HTTP endpoint.

File: `services/order-serivce/src/index.ts`

```
import axios from 'axios';

const INVOICE_GENERATOR_URL = process.env.INVOICE_GENERATOR_URL || 'http://invoice-
generator:8080';

app.post('/api/orders', authenticateToken, async (req, res) => {
    // 1. Save Order to DB
```

```
    const order = await Order.create({ ...req.body });

    // 2. Trigger the FaaS (Fire and Forget)
    // We don't await this because we don't want to block the user response
    // while the PDF is generating.
    axios.post(INVOICE_GENERATOR_URL, {
        orderId: order.id,
        customerName: order.customerName,
        total: order.total
    })
    .then(response => {
        console.log(`Invoice ready at: ${response.data.url}`);
    })
    .catch(err => {
        console.error("Invoice generation failed:", err.message);
    });

    res.status(201).json(order);
});
```

4. Configuration & Testing

In your `docker-compose.yaml`, you link the services together.

```
invoice-generator:
    build: ./functions/invoice-generator
    ports:
      - "8080:8080"

  order-service:
    environment:
    # Local Docker
    # - INVOICE_GENERATOR_URL=http://invoice-generator:8080
    # Cloud
      - INVOICE_GENERATOR_URL=https://us-central1-logistics-
system.cloudfunctions.net/generateInvoice
```

Testing the result:

1. Start the stack: `docker-compose up`
2. Log in to the Frontend
3. Click “Place Order”
4. Check the docker logs

Expected output:

```
invoice-generator-1  | ================================================
invoice-generator-1  | GENERATING INVOICE
invoice-generator-1  | ================================================
invoice-generator-1  | Order ID:       9b1deb4d-3b7d...
invoice-generator-1  | Customer:       John Doe
invoice-generator-1  | Total Amount:   $45.00
invoice-generator-1  | Invoice generated successfully
```

Conclusion

By using the Google Cloud Functions Framework, we achieved a fully functional workflow: We developed and tested our FaaS locally using Docker, and then deployed that exact same code to Google Cloud Platform to leverage infinite scalability.

We have effectively decoupled a resource-intensive task (PDF generation) from our core Order Service. This architecture offers distinct advantages:

- Elastic Scalability: Whether we receive 1 order or 10000 concurrent orders, the cloud infrastructure automatically provisions the necessary compute power
- Cost Efficiency: When no orders are being placed, our invoice generator “scales to zero” consuming no resources and costing nothing
- Resilience: Failures in the invoice generator do not crash the main Order Service, ensuring the core business flow remains uninterrupted

This pattern allows us to build granular, efficient, and highly maintainable software.