Pattern: Serverless deployment

Context

You have applied the Microservice architecture pattern (/patterns/microservices.html) and architected your system as a set of services. Each service is deployed as a set of service instances for throughput and availability.

Problem

How are services packaged and deployed?

Forces

- Services are written using a variety of languages, frameworks, and framework versions
- Each service consists of multiple service instances for throughput and availability
- Service must be independently deployable and scalable
- Service instances need to be isolated from one another
- You need to be able to quickly build and deploy a service
- You need to be able to constrain the resources (CPU and memory) consumed by a service
- You need to monitor the behavior of each service instance
- You want deployment to reliable
- You must deploy the application as cost-effectively as possible

Solution

Use a deployment infrastructure that hides any concept of servers (i.e. reserved or preallocated resources)- physical or virtual hosts, or containers. The infrastructure takes your service’s code and runs it. You are charged for each request based on the resources consumed.

To deploy your service using this approach, you package the code (e.g. as a ZIP file), upload it to the deployment infrastructure and describe the desired performance characteristics.

The deployment infrastructure is a utility operated by a public cloud provider. It typically uses either containers or virtual machines to isolate the services. However, these details are hidden from you. Neither you nor anyone else in your organization is responsible for managing any low-level infrastructure such as operating systems, virtual machines, etc.

Examples

There are a few different serverless deployment environments:

- AWS Lambda (https://aws.amazon.com/lambda/)
- Google Cloud Functions (https://cloud.google.com/functions/docs)
- Azure Functions (https://azure.microsoft.com/en-us/services/functions/)

They offer similar functionality but AWS Lambda has the richest feature set. An AWS Lambda function is a stateless component that is invoked to handle events. To create an AWS Lambda function you package your the NodeJS, Java or Python code for your service in a ZIP file, and upload it to AWS Lambda. You also specify the name of function that handles events as well as resource limits.

When an event occurs, AWS Lambda finds an idle instance of your function, launching one if none are available and invokes the handler function. AWS Lambda runs enough instances of your function to handle the load. Under the covers, it uses containers to isolate each instance of a lambda function. As you might expect, AWS Lambda runs the containers on EC2 instances.

There are four ways to invoke a lambda function. One option is to configure your lambda function to be invoked in response to an event generated by an AWS service such as S3, DynamoDB or Kinesis. Examples of events include the following:

- an object being created in a S3 bucket
- an item is created, updated or deleted in a DynamoDB table
- a message is available to read from a Kinesis stream
- an email being received via the Simple email service.

Another way to invoke a lambda function is to configure the AWS Lambda Gateway to route HTTP requests to your lambda. AWS Gateway transforms an HTTP request into an event object, invokes the lambda function, and generates a HTTP response from the lambda function's result.

You can also explicitly invoke your lambda function using the AWS Lambda Web Service API. Your application that invokes the lambda function supplies a JSON object, which is passed to the lambda function. The web service call returns the value returned by the lambda.

The fourth way to invoke a lambda function is periodically using a cron-like mechanism. You can, for example, tell AWS to invoke you lambda function every five minutes.

The cost of each invocation is a function of the duration of the invocation, which is measured in 100 millisecond increments, and the memory consumed.

Resulting context

The benefits of using serverless deployment include:

- It eliminates the need to spend time on the undifferentiated heavy lifting of managing low-level infrastructure. Instead, you can focus on your code.

- The serverless deployment infrastructure is extremely elastic. It automatically scales your services to handle the load.

- You pay for each request rather than provisioning what might be under utilized virtual machines or containers.

The drawbacks of serverless deployment include:

- Significant limitation and constraints - A serverless deployment environment typically has far more constraints that a VM-based or Container-based infrastructure. For example, AWS Lambda only supports a few languages. It is only suitable for deploying stateless applications that run in response to a request. You cannot deploy a long running stateful application such as a database or message broker.

- Limited "input sources" - lambdas can only respond to requests from a limited set of input sources. AWS Lambda is not intended to run services that, for example, subscribe to a message broker such as RabbitMQ.

- Applications must startup quickly - serverless deployment is not a good fit your service takes a long time to start

- Risk of high latency - the time it takes for the infrastructure to provision an instance of your function and for the function to initialize might result in significant latency. Moreover, a serverless deployment infrastructure can only react to increases in load. You cannot proactively pre-provision capacity. As a result, your application might initially exhibit high latency when there are sudden, massive spikes in load.

The deployment infrastructure will internally deploy your application using one of the other patterns. It will most likely use Service Service per Host pattern (single-service-per-host.html).

Related patterns

- The Multiple Service Instances per Host pattern (multiple-services-per-host.html) is an alternative deployment solution

- The Single Service per Host pattern (single-service-per-host.html) is an alternative deployment solution

Tweet

Follow @MicroSvcArch