Serverless Applications
AWS Lambda

Outline

- Overview of Serverless
- Serverless Architectures
- AWS Lambda Example Architecture
- Overview of Containers
- Container Architectures
- Where we go from here
- AWS Lambda + AWS API Gateway

Need for Virtual Machines

The Issue

- Deployment of server applications is getting complicated since software can have many types of requirements.

The Solution

- Run each individual application on a separate virtual machine.

Virtualization

Offers a hardware abstraction layer that can adjust to the specific CPU, memory, storage and network needs of applications on a per server basis.

Virtual Machines are expensive

The Problems with Virtual machines

- Money – You need to predict the instance size you need. You are charged for every CPU cycle, even when the system is "running its thumbs"
- Time – Many operations related to virtual machines are typically slow

The Solution

- Serverless Architectures
- Containers

Containers

Operating System Level virtualization, a lightweight approach to virtualization that only provides the bare minimum that an application requires to run and function as intended.

What is Serverless?

- *Serverless architectures refer to applications that significantly depend on third party-services (known as Backend as a Service or "Baas") or on custom code that's run in ephemeral containers (Function as a Service or "FaaS"), the best known vendor host of which currently is AWS Lambda. By using these ideas, and by moving much behavior to the front end, such architectures remove the need for the traditional 'always on' server system sitting behind an application.*

- "**No server is easier to manage than no server**" – Werner Vogels

Note: slides provided by Nate Slater, Senior Manager AWS Solutions Architecture, "*State of Container and Serverless Architectures*"

Features of Serverless Architectures

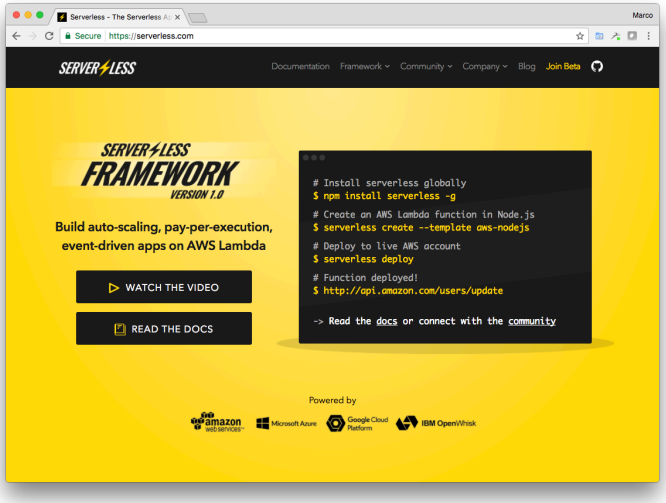
- No compute resource to manage
- Provisioning and scaling handled by the service itself
- You write code and the execution environment is provided by the service
- Core functionality (e.g. database, authentication and authorization) is provided by at-scale Web Services

The origins of “Functions-as-a-Service”

- **AWS Lambda** – Announced at re:Invent 2014. First web service of it’s kind that completely abstracted the execution environment from the code
- **API Gateway** – Launched in mid-2015. Critical ingredient for building service endpoints with Lambda.
- Combined with existing back-plane services like DynamoDB, Cloudformation, and S3 and “serverless” development was born.

The FaaS Development Framework Ecosystem

- Serverless Framework (serverless.com)
  - Open source framework for building serverless applications with AWS Lambda, Microsoft Azure and IBM OpenWhisk (and soon Google Cloud Functions)
  - Supports node.js, python, and java
- Chalice
  - Python based framework for microservice development with AWS Lambda
- Apex
  - A set of tools written in Go to manage serverless deployments to AWS Lambda
- • Serverless Application Module (SAM)
  - AWS framework that extends Cloudformation



FaaS – How Does it Work?

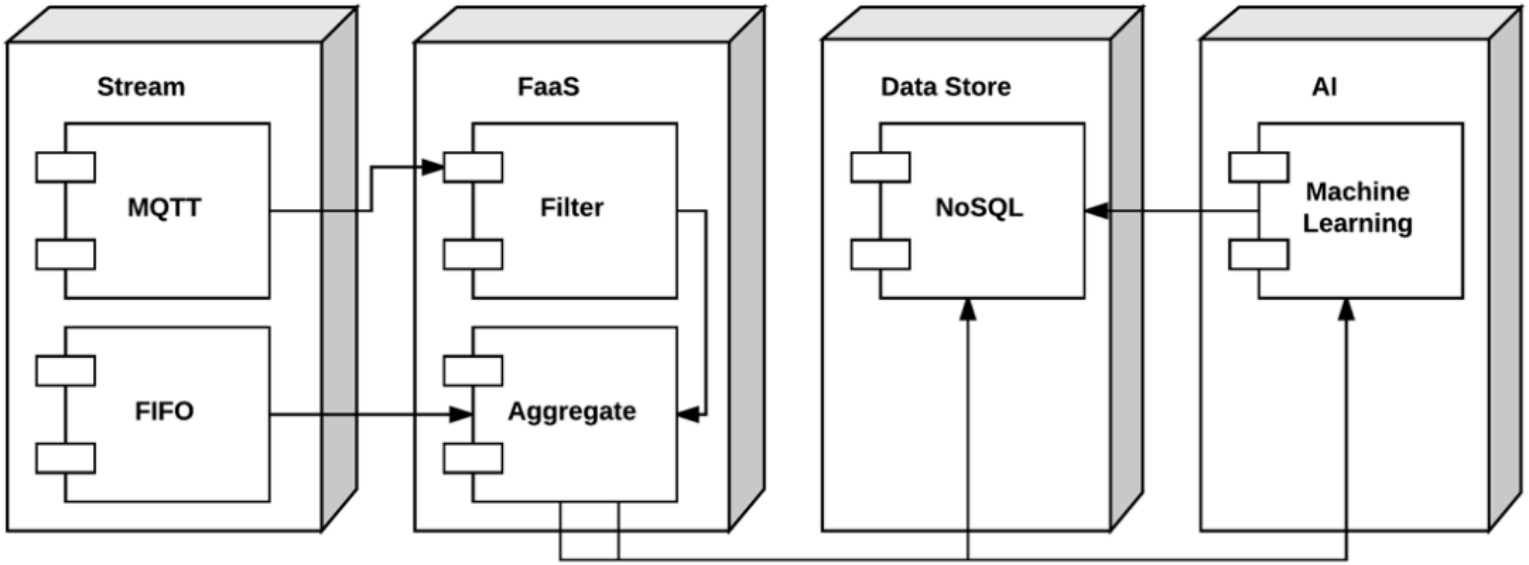
- You write a function and deploy it to the cloud service for execution.
- Example: node.js with AWS Lambda:

```
module.exports.handler = function(event, context, callback) {
  console.log("event: " + JSON.stringify(event));
    if ((!event.hasOwnProperty("email") ||
!event.hasOwnProperty("restaurantId")) || (!event.email ||
!event.restaurantId)) { callback("[BadRequest] email and
restaurantId are required");
  return; }
}
```

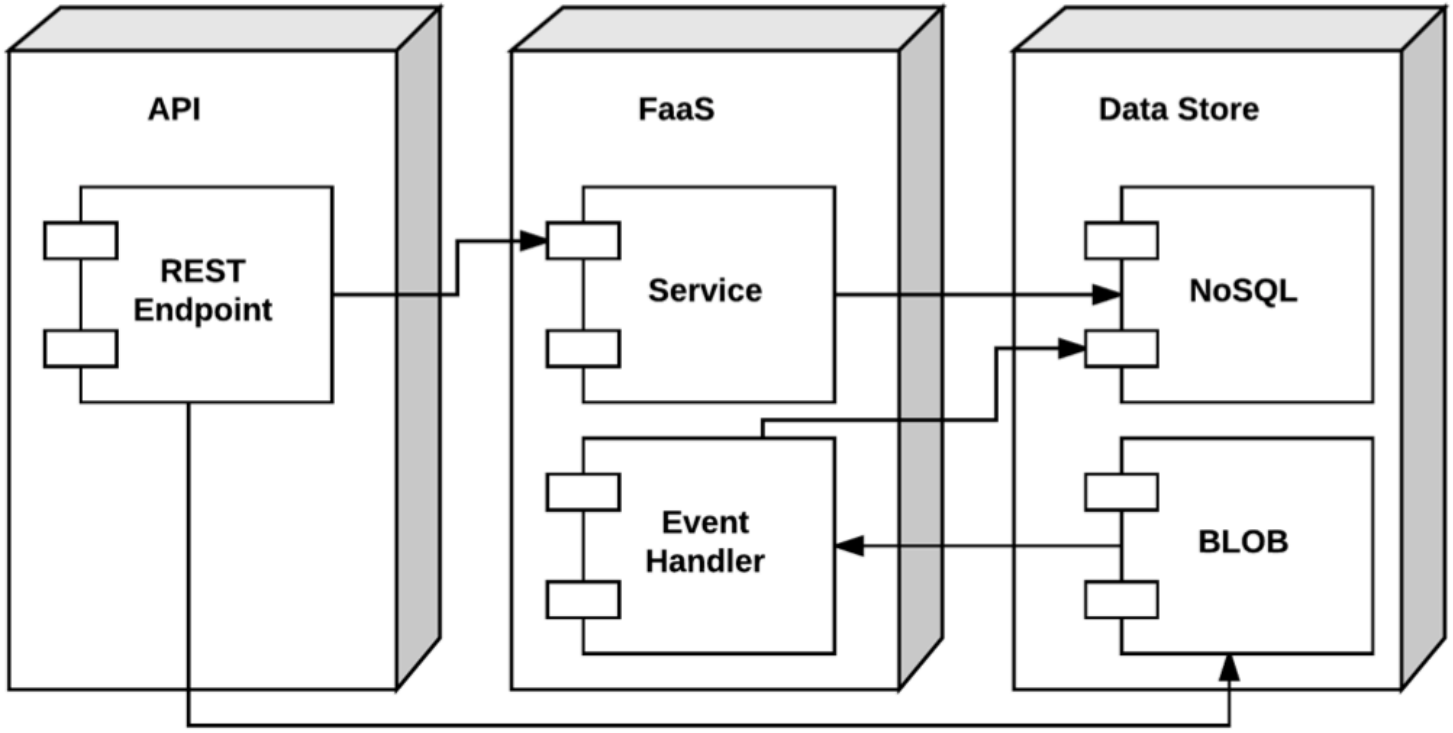
Backend-as-a-Service

- Data Stores
  - NoSQL Databases
  - BLOB Storage
  - Cache (CDN)
- Analytics
  - Query
  - Search
  - IoT
  - Stream Processing
- AI
  - Machine Learning
  - Image Recognition
  - Natural Language Processing/Understanding
  - Speech to Text/Text to Speech

Serverless Architecture – Stream Analytics/IoT



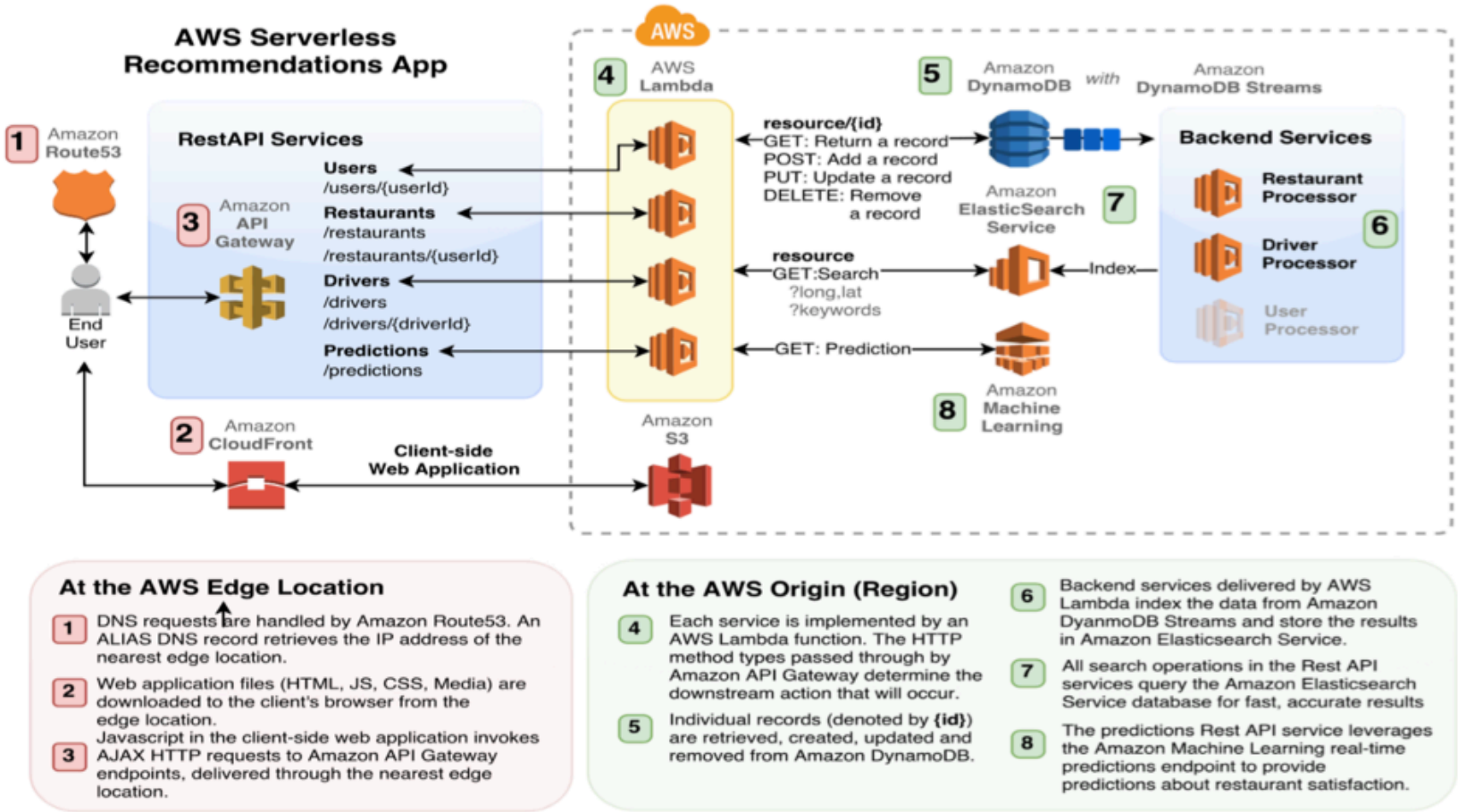
Serverless Architecture - Microservices



AWS Lambda Example Architecture

- Search
- Location-Awareness
- Machine-learning powered recommendations
- NoSQL
- Microservices
- API Management
- Static website with CDN
- Not a single server to manage!

AWS Lambda Example Architecture (cont'd)



At the AWS Edge Location

1. DNS requests are handled by Amazon Route53. An ALIAS DNS record retrieves the IP address of the nearest edge location.
2. Web application files (HTML, JS, CSS, Media) are downloaded to the client's browser from the edge location.
3. Javascript in the client-side web application invokes AJAX HTTP requests to Amazon API Gateway endpoints, delivered through the nearest edge location.

At the AWS Origin (Region)

4. Each service is implemented by an AWS Lambda function. The HTTP method types passed through by Amazon API Gateway determine the downstream action that will occur.
5. Individual records (denoted by **{id}**) are retrieved, created, updated and removed from Amazon DynamoDB.
6. Backend services delivered by AWS Lambda index the data from Amazon DyanmoDB Streams and store the results in Amazon Elasticsearch Service.
7. All search operations in the Rest API services query the Amazon Elasticsearch Service database for fast, accurate results
8. The predictions Rest API service leverages the Amazon Machine Learning real-time predictions endpoint to provide predictions about restaurant satisfaction.

What are Containers?

- Virtualization at the OS level
- Based on Linux kernel features
  - cgroups
  - namespaces
- Concept has been around for a while
  - Solaris ”zones” – early form of containerization
  - LXC – Early form of containerization on Linux
- Docker has brought containers mainstream

Containers – Key Features

- Lightweight
  - All containers running on the same host share a single Linux kernel
  - Container images don't require a full OS install like a virtual machine image
- Portable
  - Execution environment abstracts the underlying host from the container
  - No dependency on a specific virtual machine technology
  - Container images can be shared using GitHub-like repositories, such as DockerHub

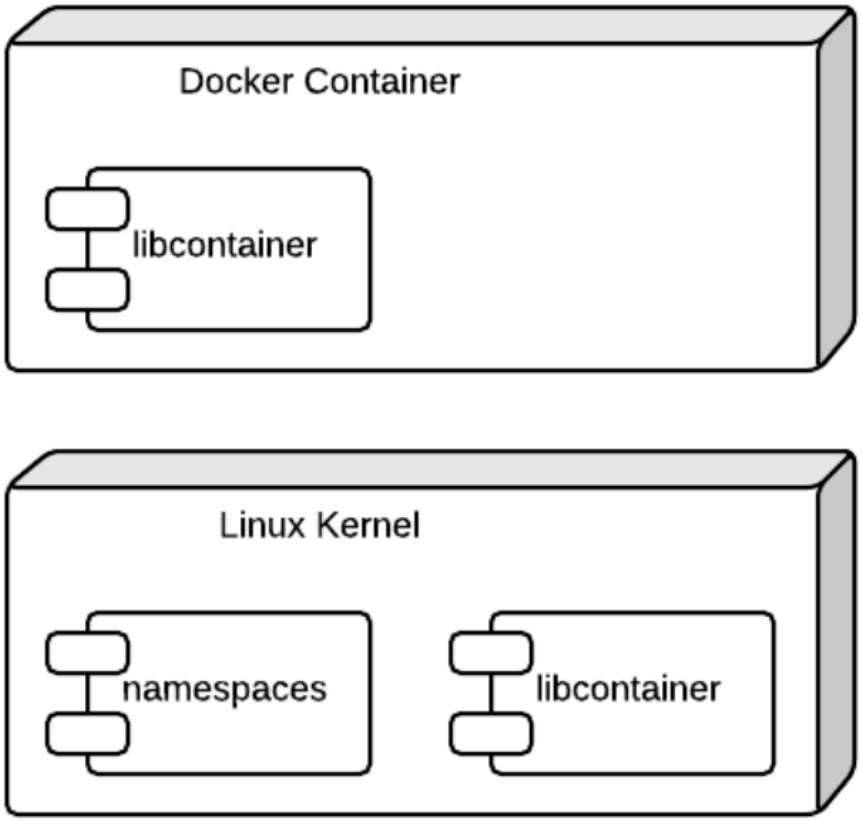
Docker

- Docker is a tool that allows developers, sys-admins etc. to easily deploy their applications in a sandbox (called containers) to run on the host operating system i.e. Linux.
- Key Benefit : Allows users to package an application with all of its dependencies into a standardized unit for software development.
- Unlike virtual machines, Containers do not have the high overhead and hence enable more efficient usage of the underlying system and resources.
- Allow extremely higher efficient sharing of resources
- Provides standard and minimizes software packaging
- Decouples software from underlying host w/ no hypervisor

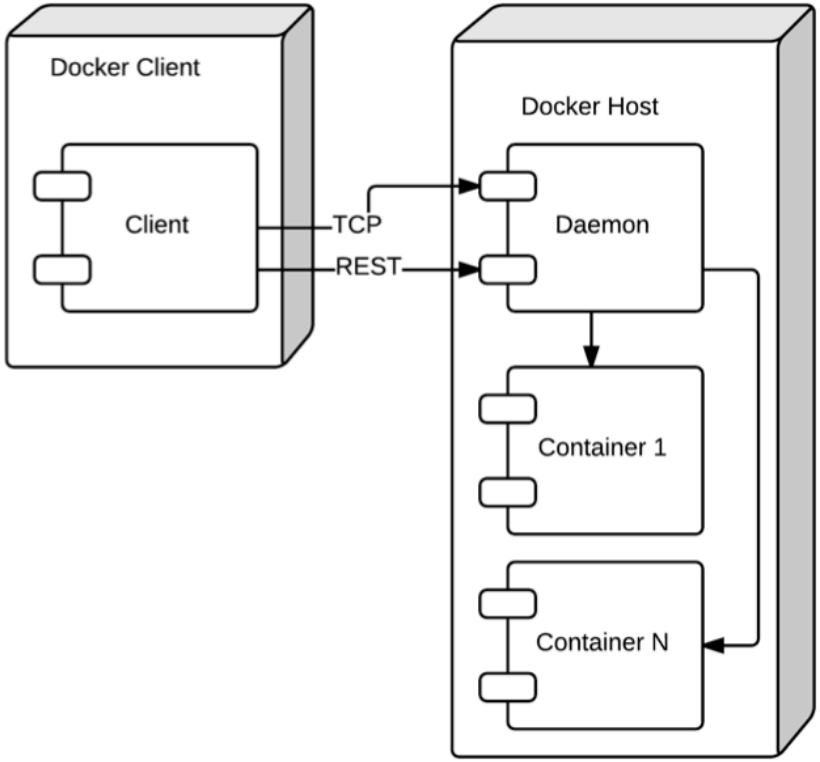
Container Issues

- Security
- Less Flexibility in Operating Systems, Networking
- Management of Docker and Container in production is challenge

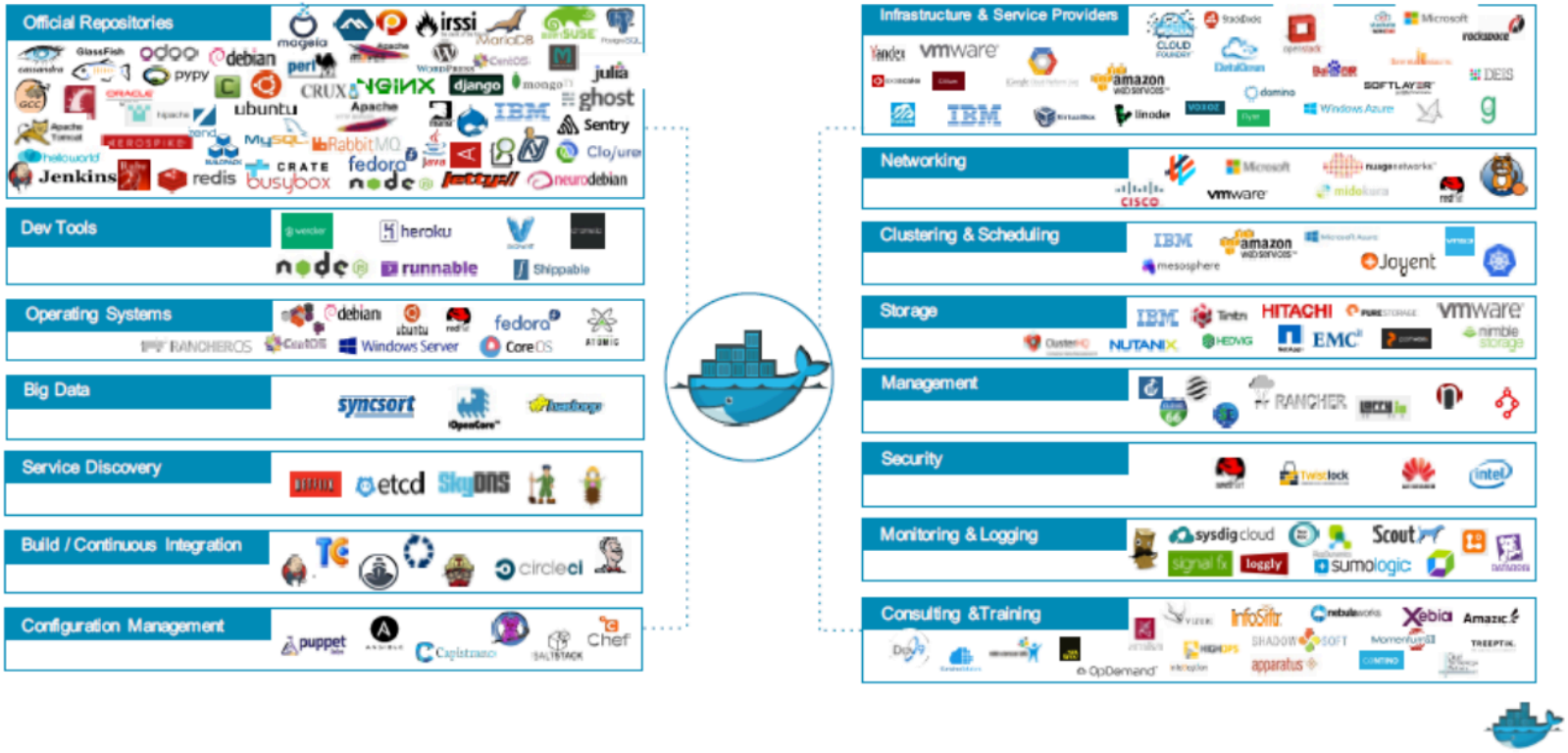
Docker Runtime Architecture



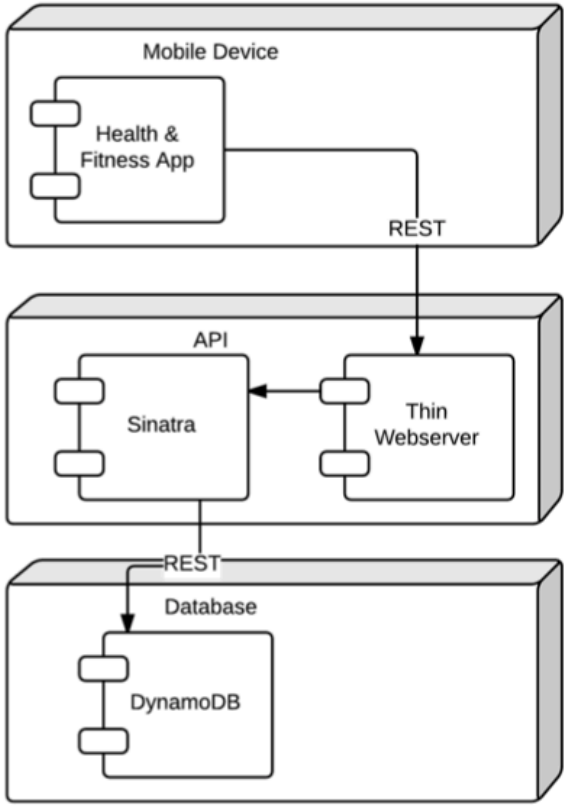
Docker – Platform Architecture



The Docker Ecosystem



Container-Based Microservice Example



Serverless – Where we go from here

- Backend-as-a-Service
  - AI
    - Fraud detection
    - Latent sematic analysis
  - Geospatial
    - Satellite imagery
    - Hyper-Locality
  - Analytics
    - Query
    - Search
    - Stream Processing
  - Database
    - Graph
  - HPC (High Performance Computing)

Serverless – Where we go from here (cont'd)

- Function-as-a-Service
  - Polyglot language support
  - Stateful endpoints (Web Sockets)
  - Remote Debugging
  - Enhanced Monitoring
  - Evolution of CI/CD Patterns
  - IDE's

Containers – Where we go from here

- Networking
  - Overlay networks between containers running across separate hosts
- Stateful Containers
  - Support for container architectures that read and write persistent data
- Monitoring and Logging
  - Evolution of design patterns for capturing telemetry and log data from running containers
- Debugging
  - Attach to running containers and debug code
- Security
  - Better isolation at the kernel level between containers running on the same host
  - Secret/Key management – Transparently pass sensitive configuration

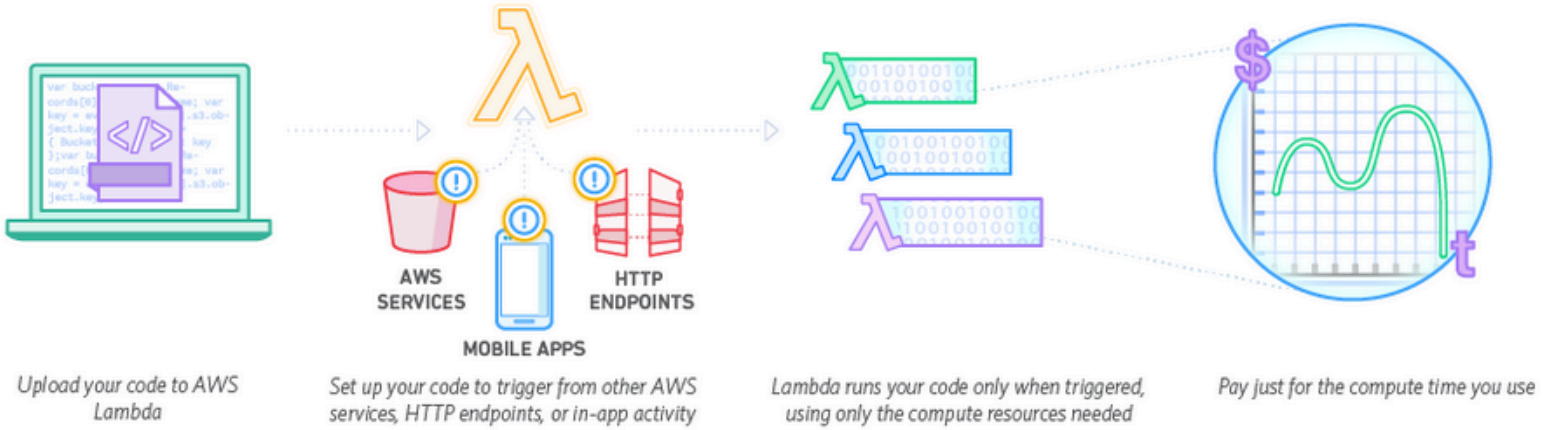
AWS Lambda

- Compute Service using Amazon's infrastructure
- Code === function
- Supported – Java, Python and Node.js
- Can say it to be Docker under the covers
- A system that uses Linux Containers
- **Pay only for the compute time you use**
- Triggered by events or called from HTTP
- It still has SERVERS but we do not care about them
- Functions are unit of deployment and scaling
- No Machines, No Vms or containers visible in Programming Model
- Never pay for idle
- AutoScaling and Always Available, adapts to rate of incoming requests

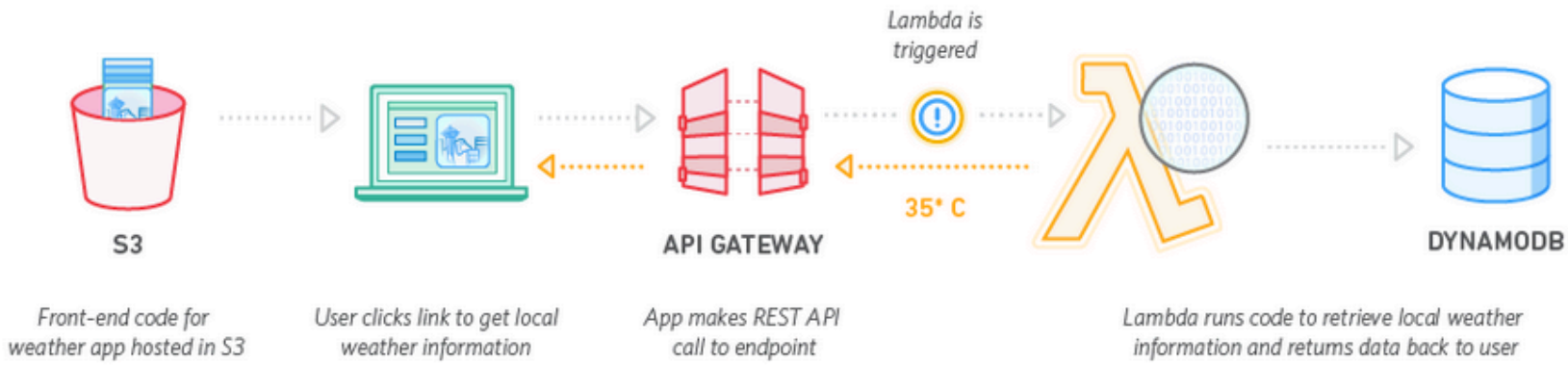
Using AWS Lambda

- No Servers to Manage
- Continuous Scaling
- Subsecond metering
- Bring your own code
- Simple resource model
- Flexible Authorization and Use
- Stateless but you can connect to others to store state
- Authoring functions
- Makes it easy to
  - Perform real time data processing
  - Build scalable backend services
  - Glue and choreograph systems

AWS Lambda - How It Works



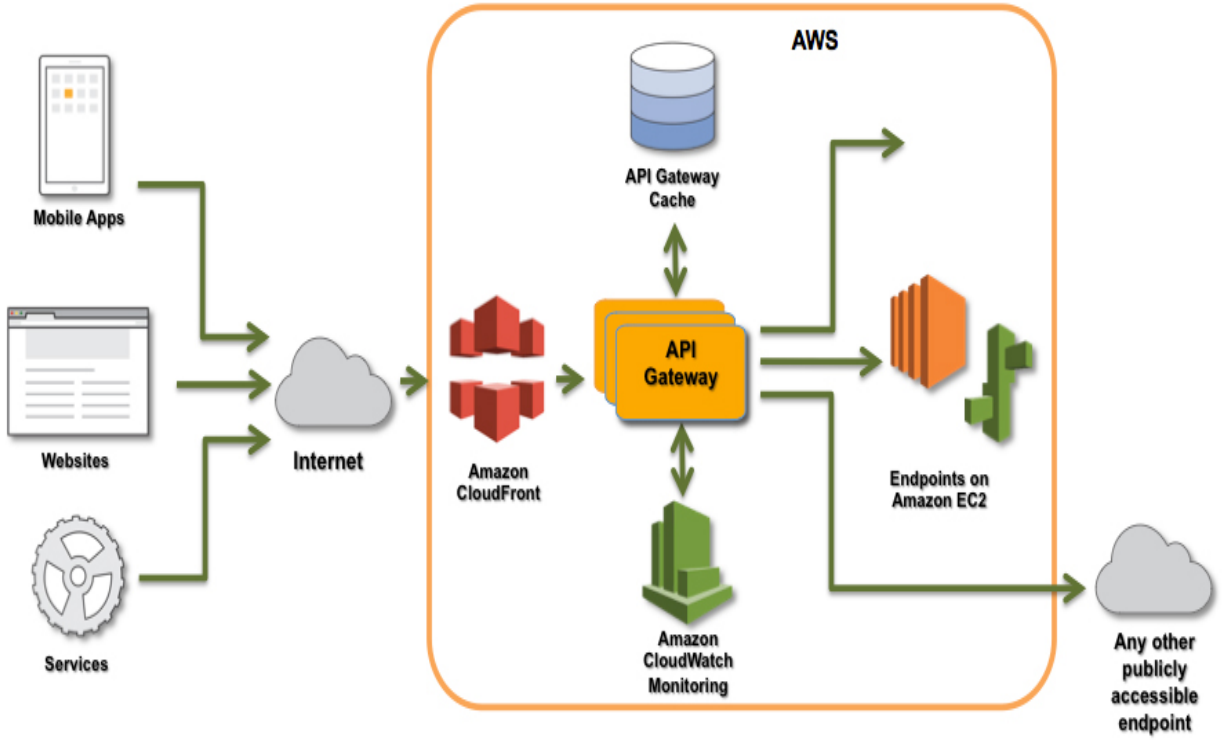
AWS Lambda - How It Works (cont'd)



Amazon API Gateway

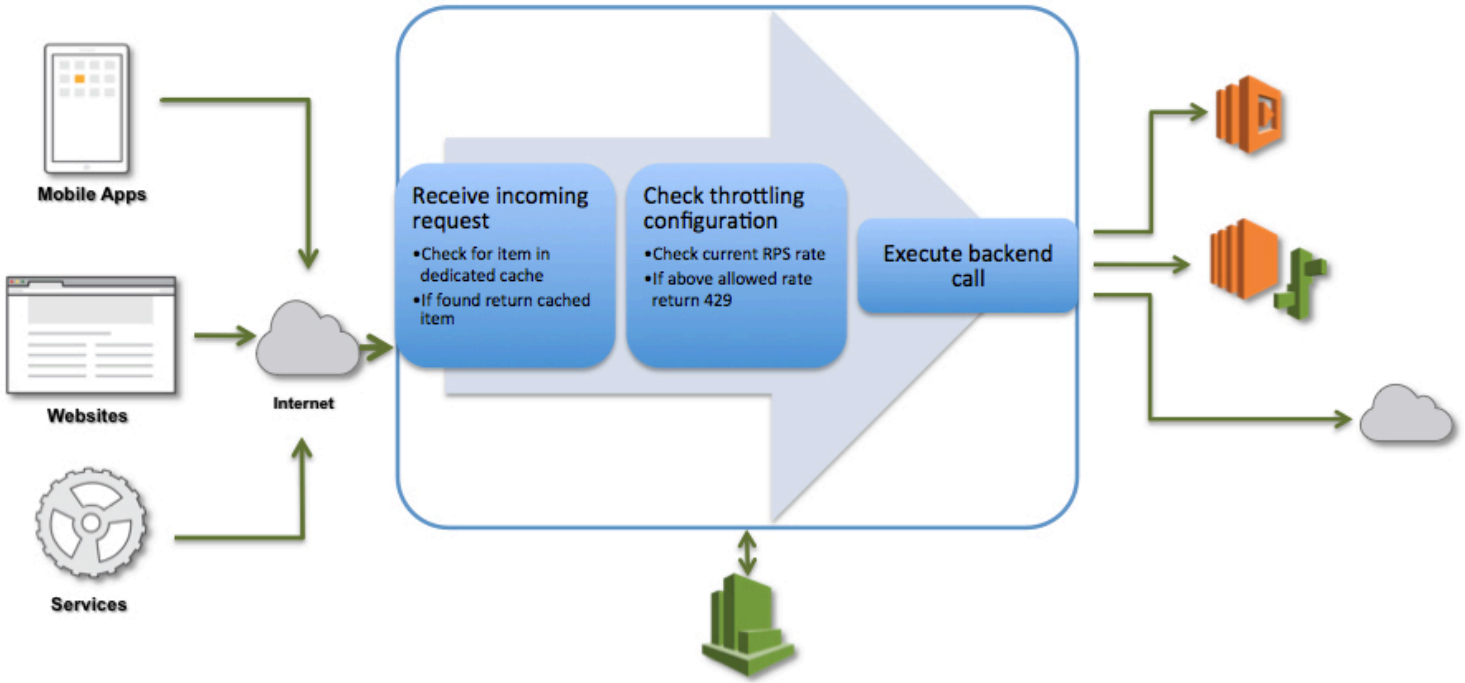
- Amazon API Gateway is a fully managed service that makes it easy for developers to create, publish, maintain, monitor, and secure APIs at any scale.
- Creates a unified API front end for multiple microservices
- DDos Protection and throttling for back end systems
- Authenticate and authorize requests

Amazon API Gateway Call Flow



An Amazon API Gateway Call Flow

Amazon API Gateway Request Processing Workflow



AWS Lambda Supported Event Sources

- Amazon S3
- Amazon DynamoDB
- Amazon Kinesis Streams
- Amazon Simple Notification Service
- Amazon Simple Email Service
- Amazon Cognito
- AWS CloudFormation
- Amazon CloudWatch Logs
- Amazon CloudWatch Events
- AWS CodeCommit
- Scheduled Events (powered by Amazon CloudWatch Events)'
- AWS Config
- Amazon Echo
- Amazon Lex
- Amazon API Gateway

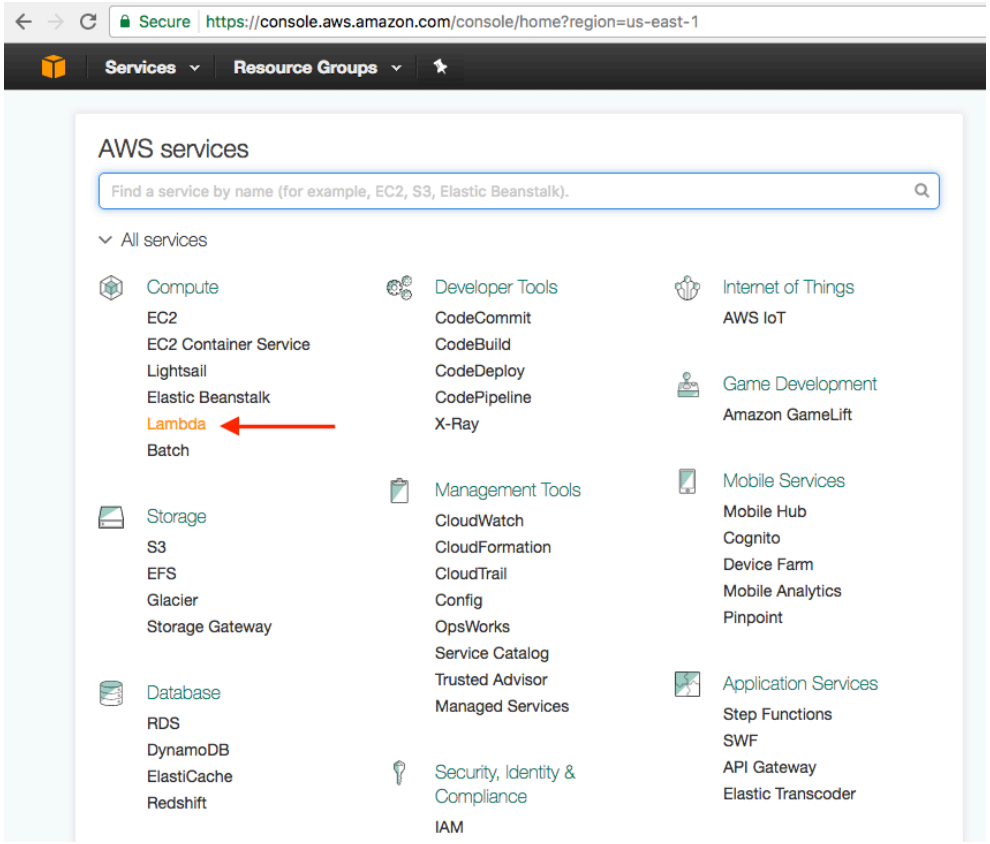
See: http://docs.aws.amazon.com/lambda/latest/dg/invoking-lambda-function.html

Create a Simple Microservice using Lambda and API Gateway

In this exercise you will use the Lambda console to create a Lambda function (MyLambdaMicroservice), and an Amazon API Gateway endpoint to trigger that function. You will be able to call the endpoint with any method (GET, POST, PATCH, etc.) to trigger your Lambda function. When the endpoint is called, the entire request will be passed through to your Lambda function. Your function action will depend on the method you call your endpoint with:

- DELETE: delete an item from a DynamoDB table
- GET: scan table and return all items
- POST: Create an item
- PUT: Update an item

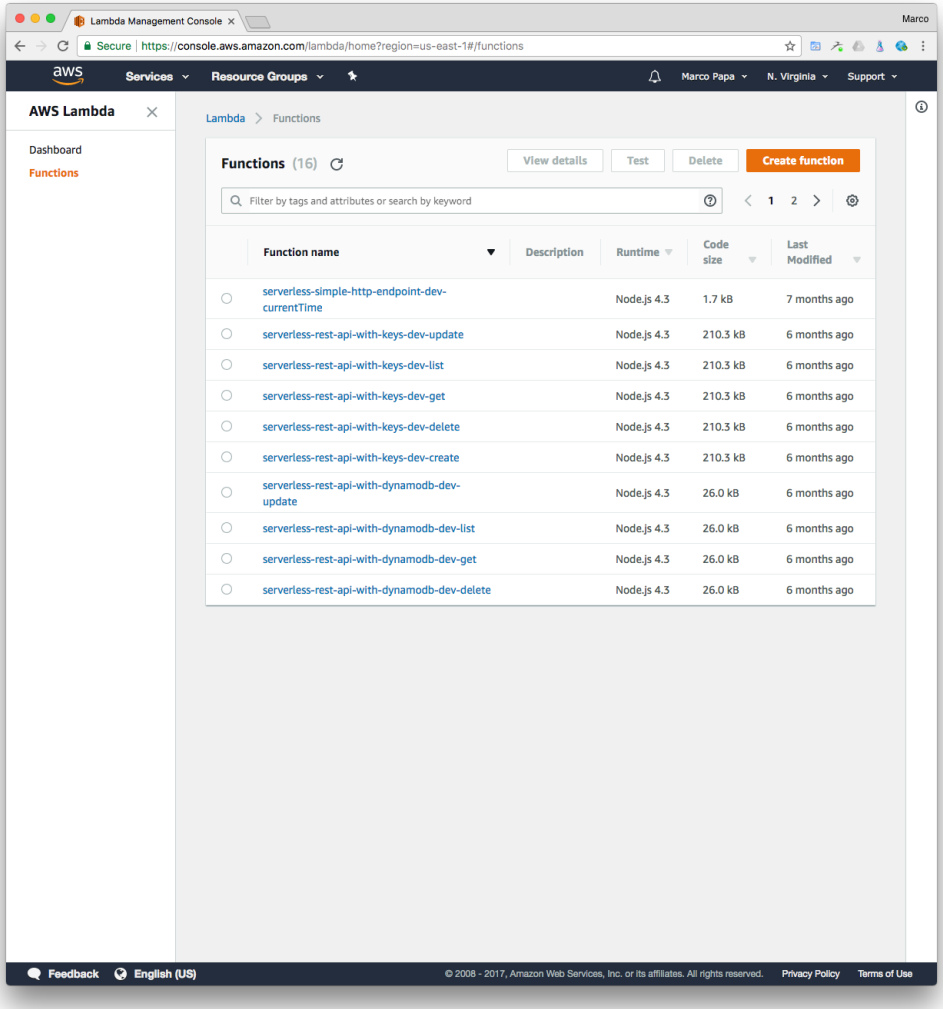
AWS Lambda (cont'd)



Follow the steps in this section to create a new Lambda function and an API Gateway endpoint to trigger it:

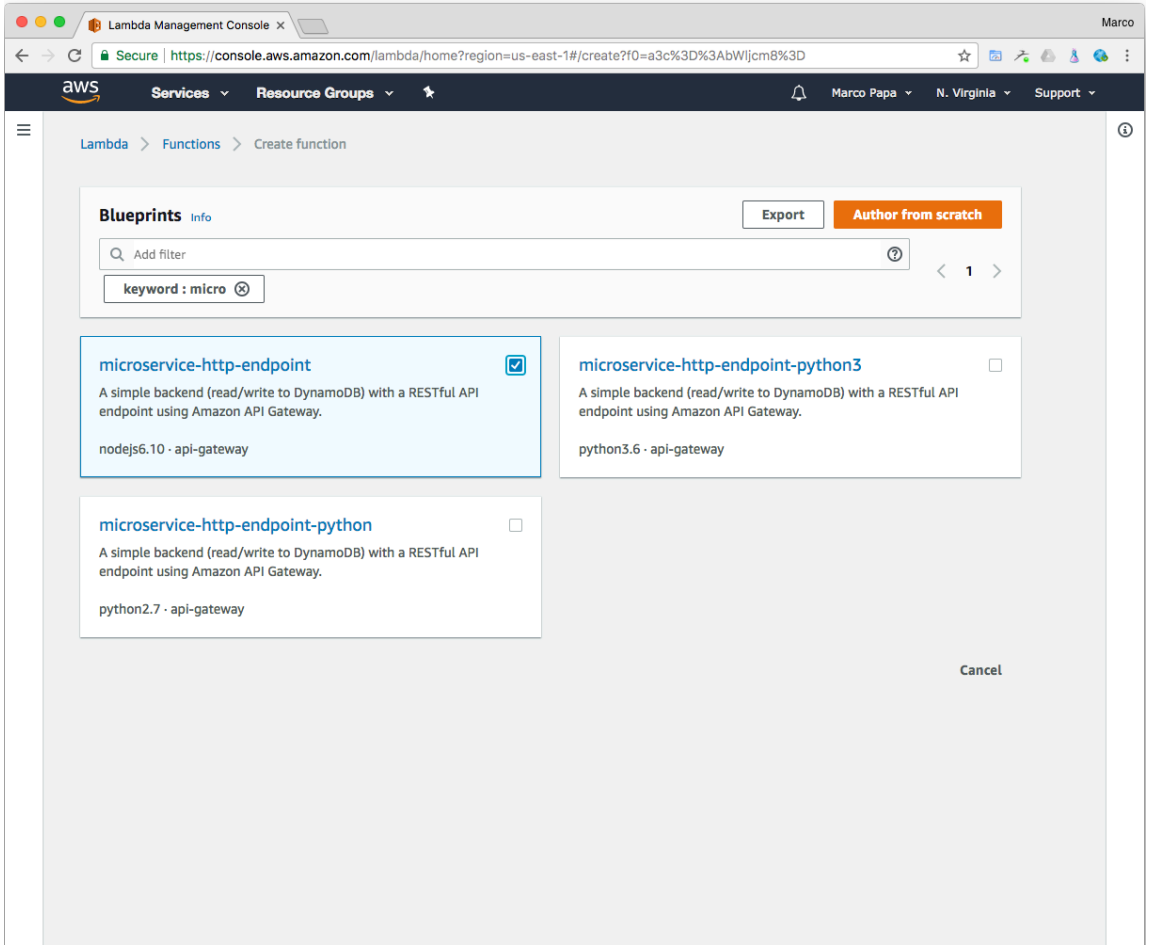
1. Sign in to the AWS Management Console and open the AWS **Lambda Management Console** under **Compute Lambda**.

AWS Lambda (cont'd)



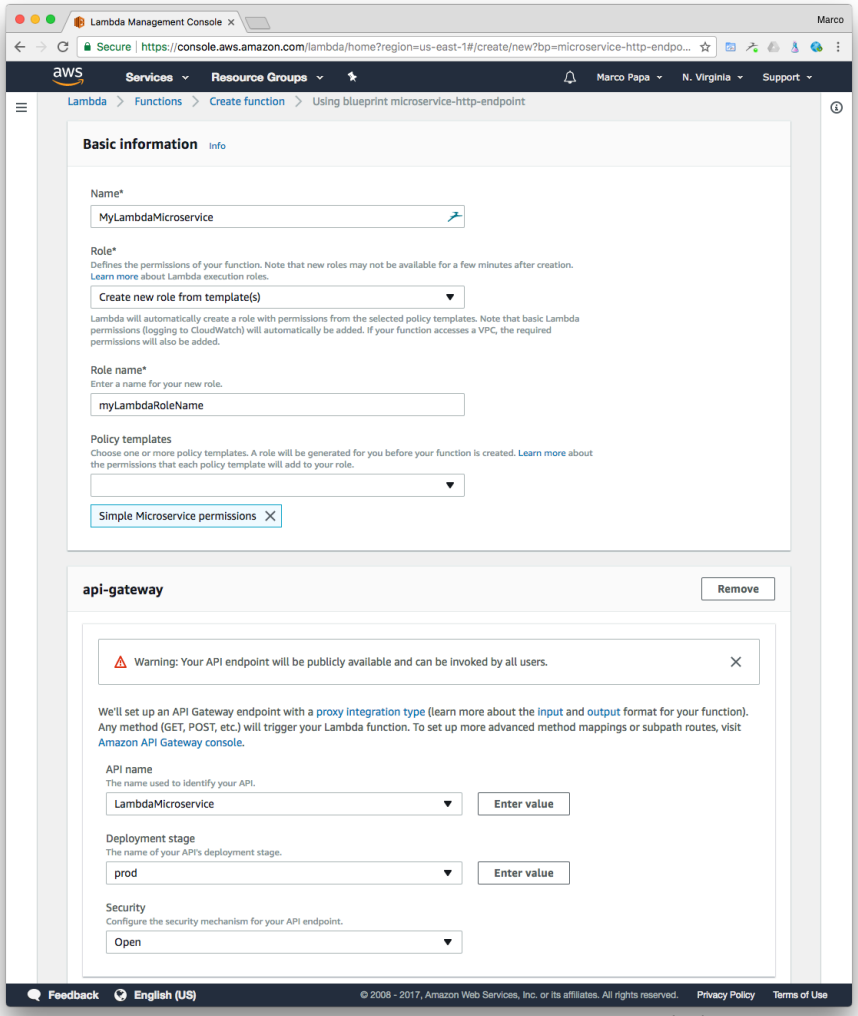
2. Choose **Create function**.

AWS Lambda (cont'd)



3. On the **Blueprints** page, choose the **microservice-http-endpoint** blueprint. You can use the Filter to find it. Just type "micro". Click the **microservice-http-endpoint** hyperlink.

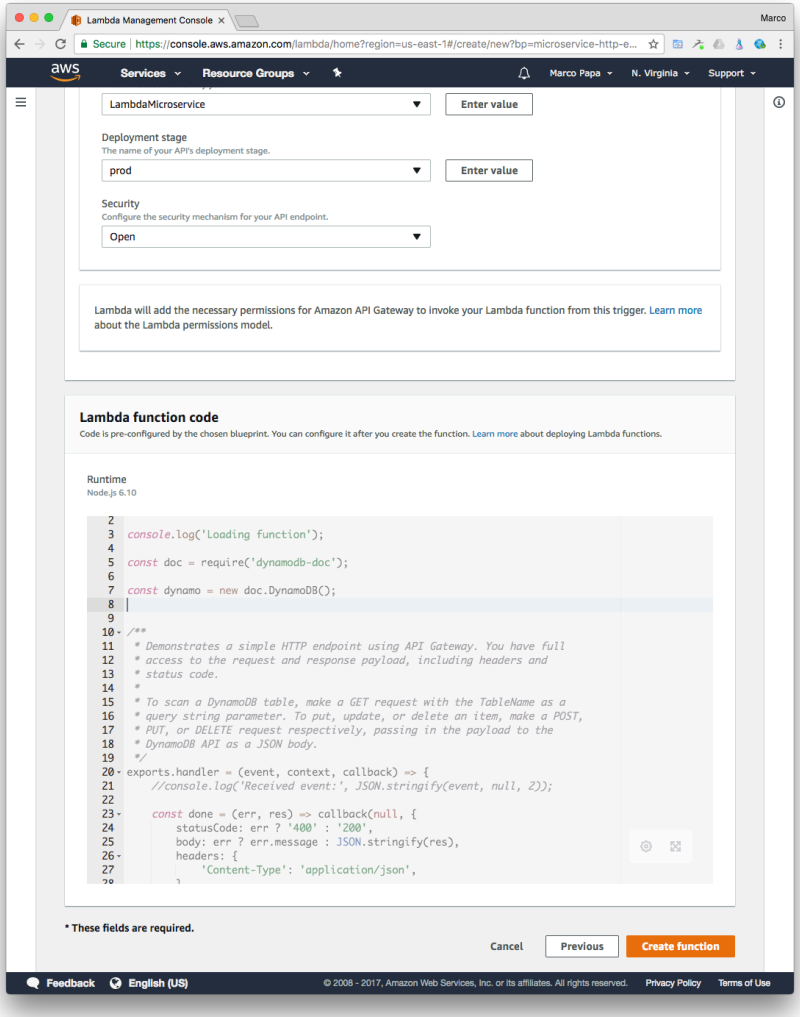
AWS Lambda (cont'd)



4. In the **Basic information** section, do the following:
   a) Enter the function name MyLambdaMicroservice in **Name**.
   b) In **Role name**, enter a role name for the new role that will be created, like myLambdaRoleName.

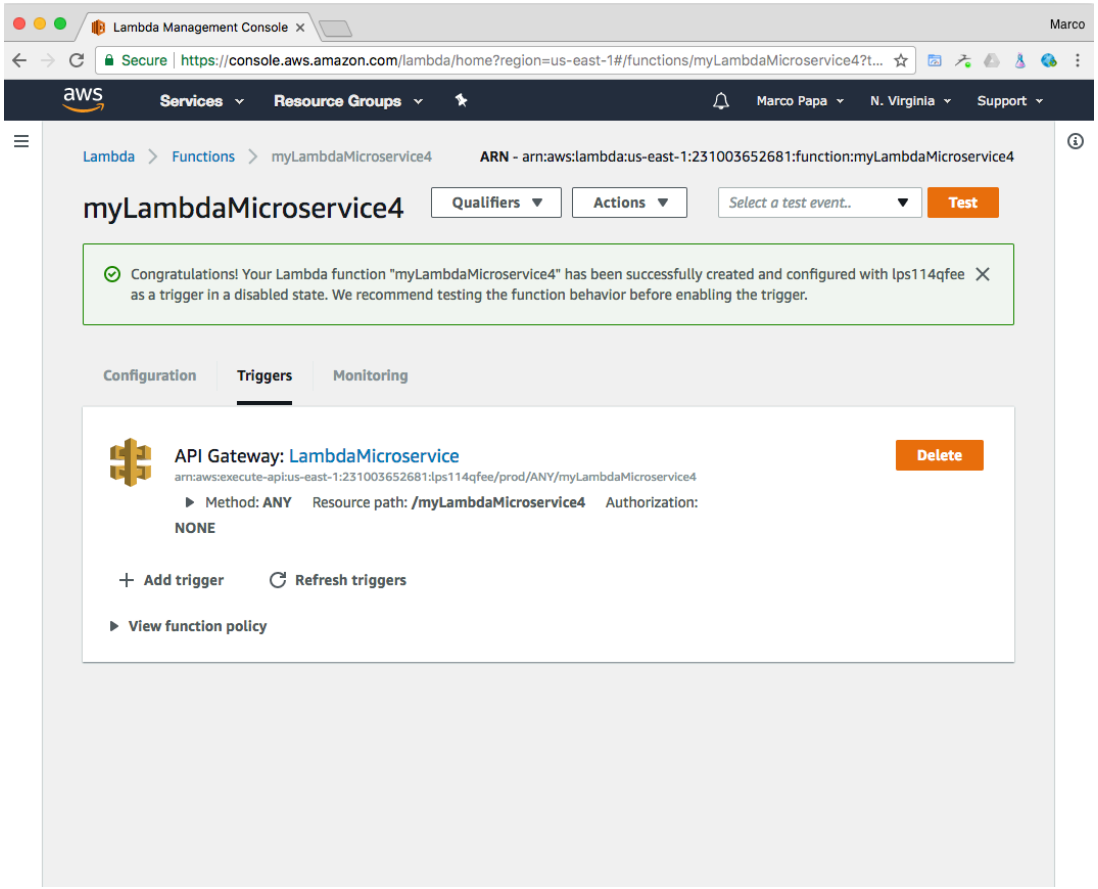
5. The **api-gateway** section will be populated with an API Gateway trigger. The default API name that will be created is `LambdaMicroservice` (You can change this name via the **API Name field** if you wish).
6. In the **Security** field, select **Open**, as we will be creating a publicly available REST API.

AWS Lambda (cont'd)



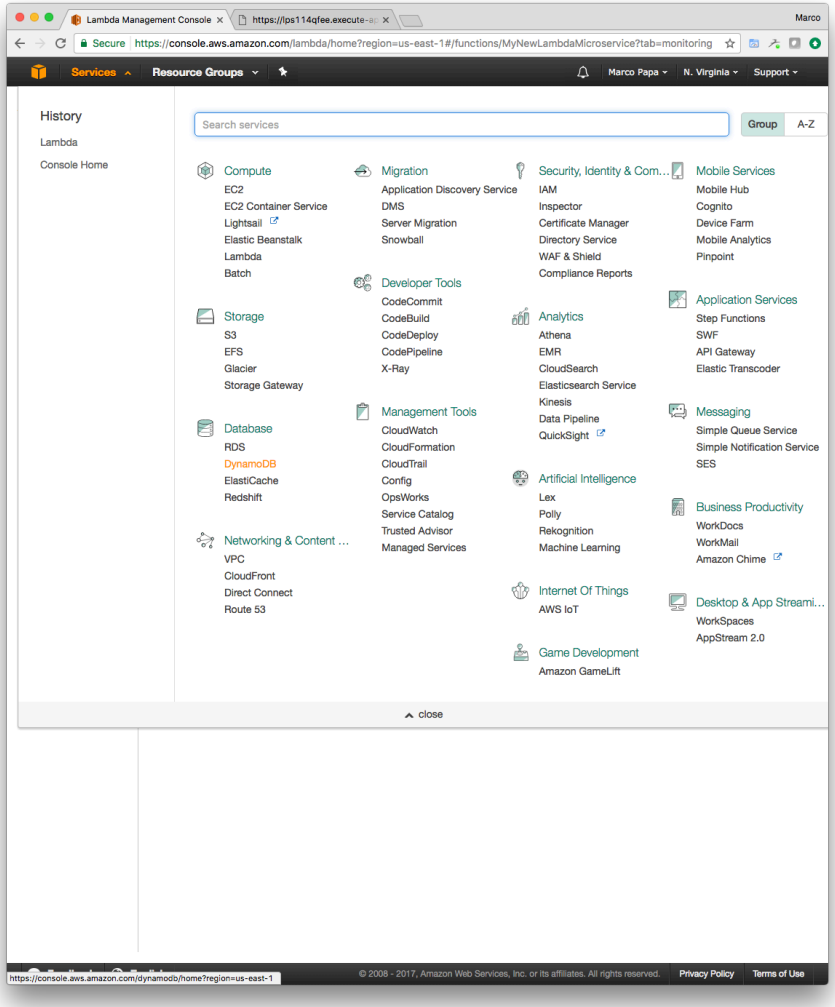
7. On the **Lambda function code** section, do the following:
   a) Review the preconfigured Lambda function configuration information, including:
      - **Runtime** is `Node.js 6.10`
      - Code authored in JavaScript is provided. The code performs DynamoDB operations based on the method called and payload provided.

AWS Lambda (cont'd)



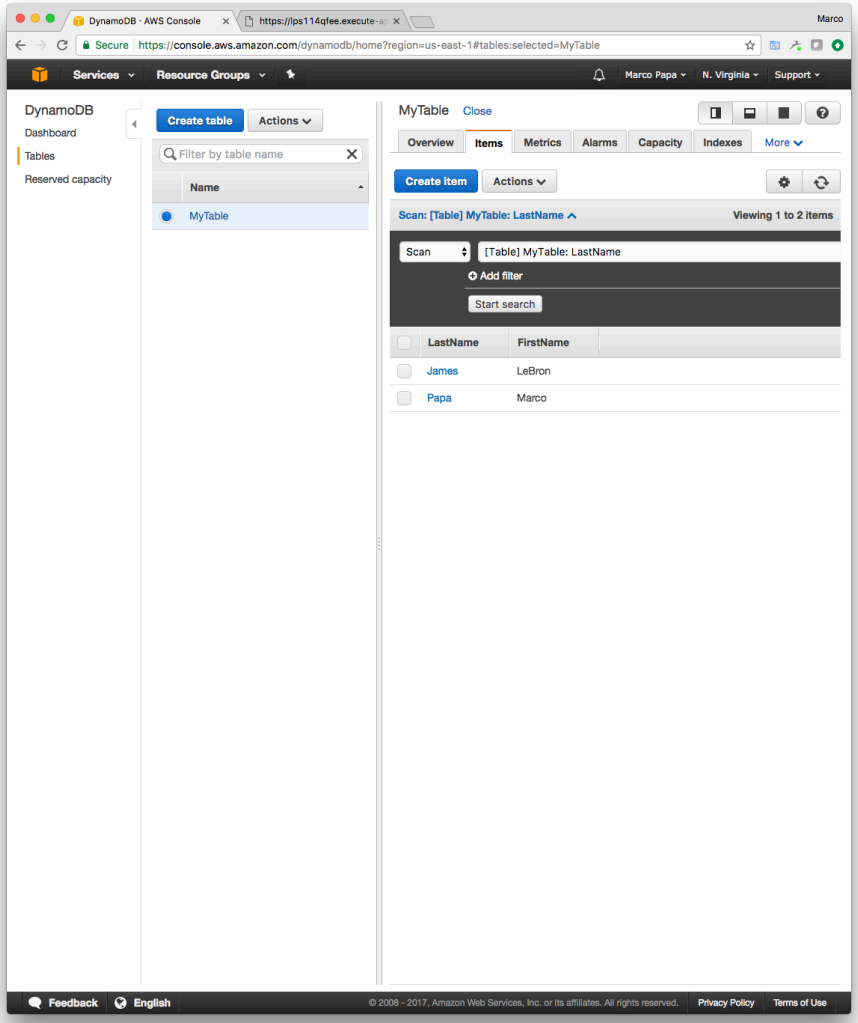
8. Chose **Create function**.
9. The "Congratulations!" page is displayed, showing the **Triggers** tab.
10. Click the **Configuration** tab. Notice **Handler** shows `index.handler`. The format is: `filename.handler-function`

AWS Lambda (cont'd)



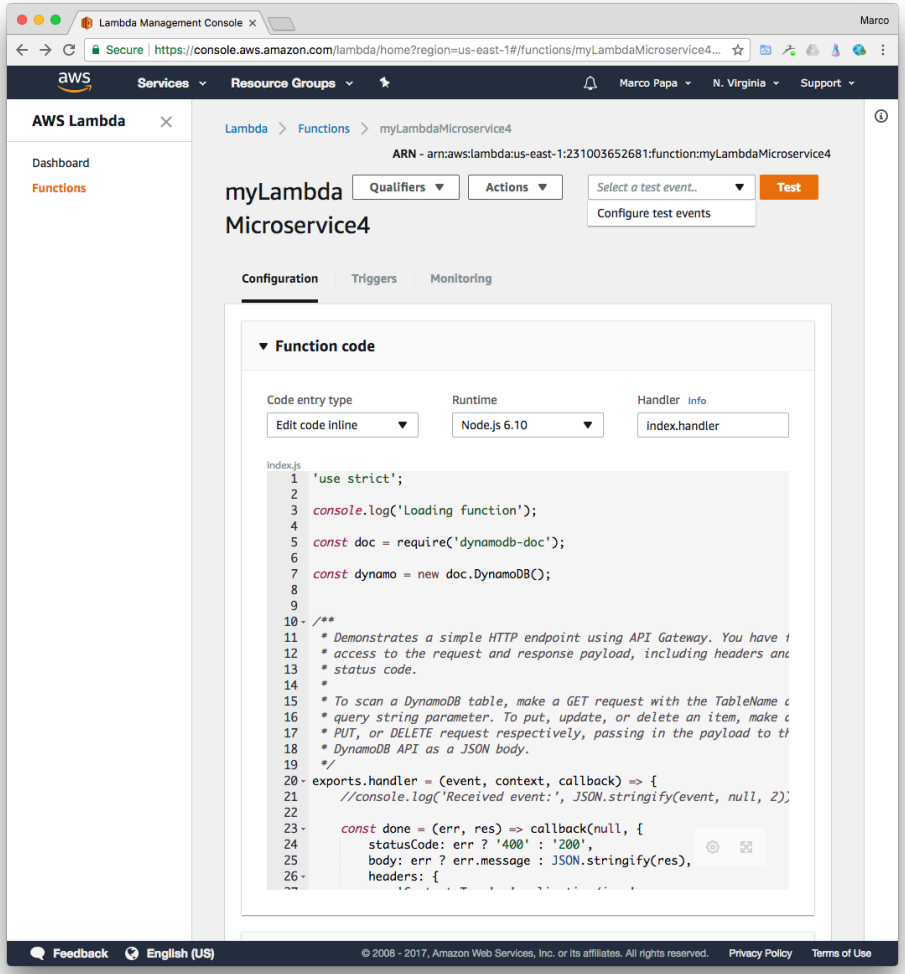
11. To test our AWS Lambda REST Service, select **Database DynamoDB** from the **Services** console.

AWS Lambda (cont'd)



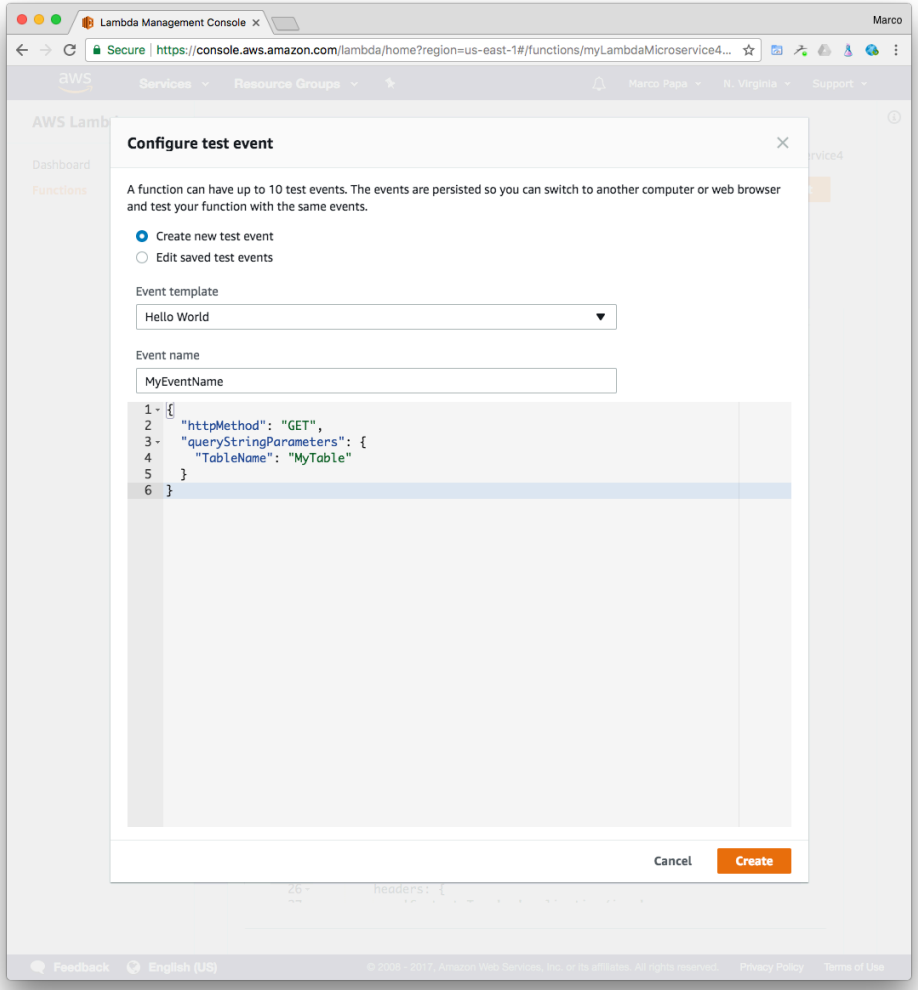
12. Create a table named **MyTable**, with two columns named **LastName** and **FirstName**, and enter a few rows.

AWS Lambda (cont'd)



13. Back to the **AWS Lambda console**. In this step, we will use the console to test the Lambda function. That is, send an HTTPS request to the API method and have Amazon API Gateway invoke your Lambda function.
14. Select **Functions** from left navigation. The With the `MyLambdaMicroService` function still open in the console, choose the **Select a test event** dropdown and then choose **Configure test events**.

AWS Lambda (cont'd)

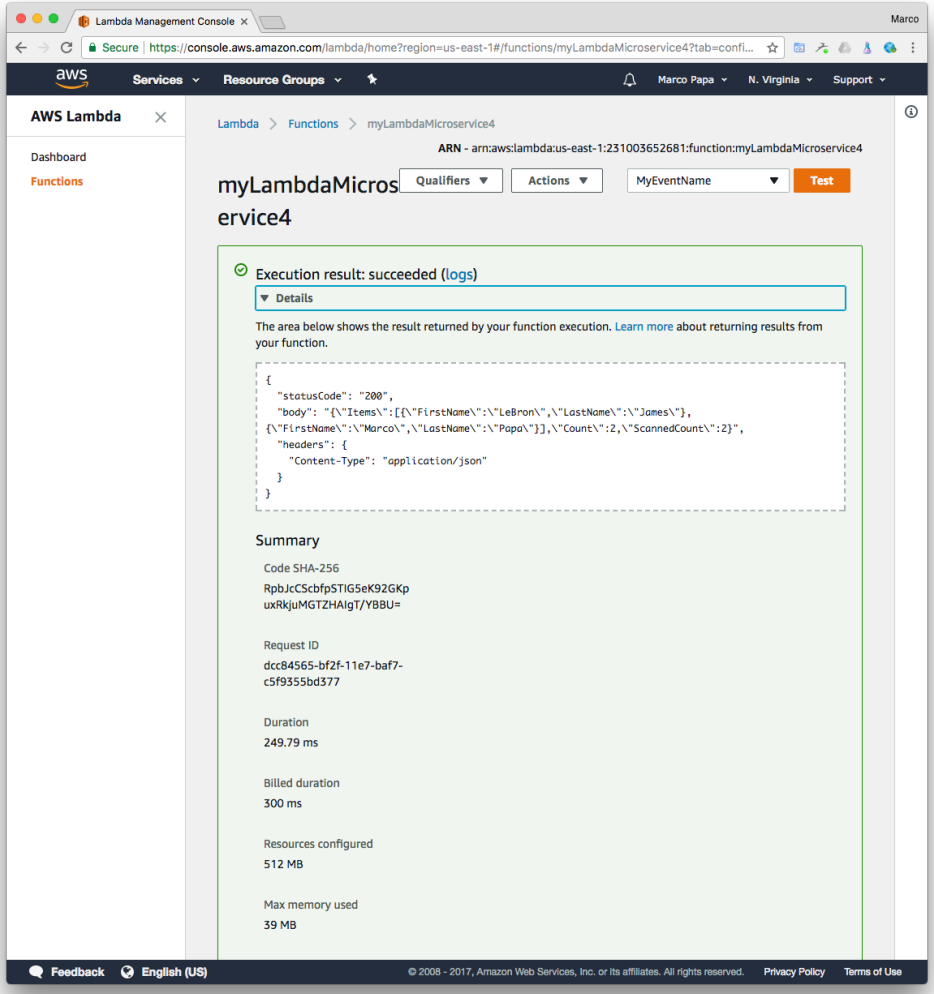


13. In the **Configure test event** page, enter an **Event Name**. Replace the existing text with the following:

```
{
  "httpMethod": "GET",
  "queryStringParameters": {
  "TableName": "MyTable"
  }
}
```

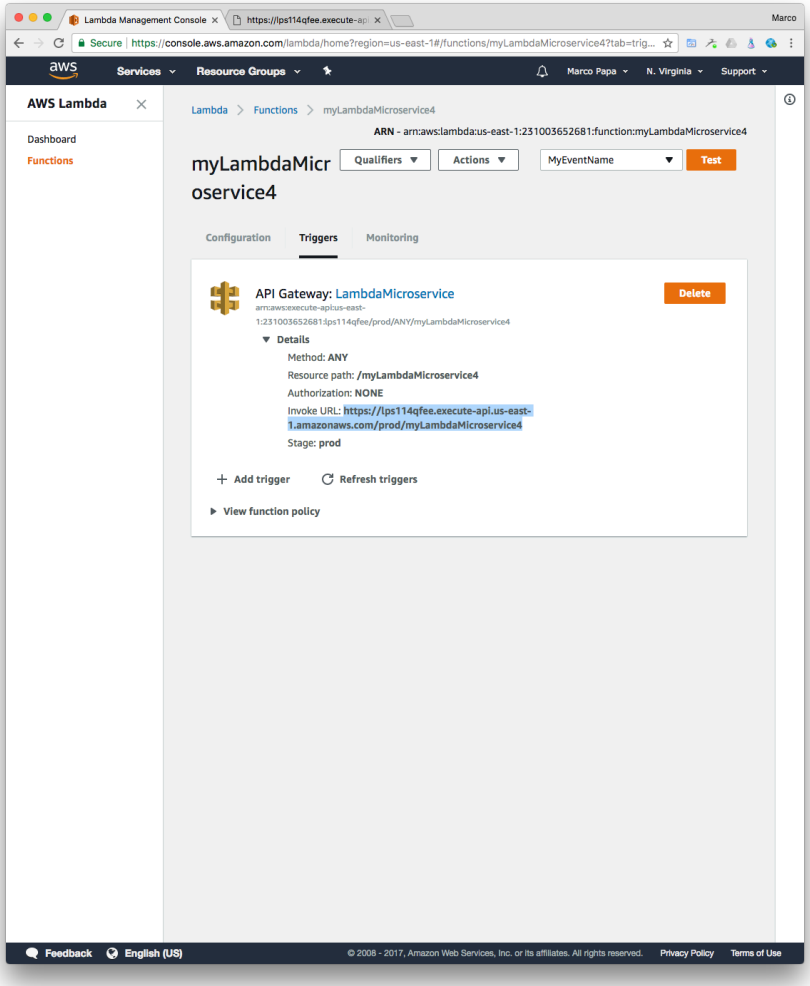
14. After entering the text above choose **Create**.

AWS Lambda (cont'd)



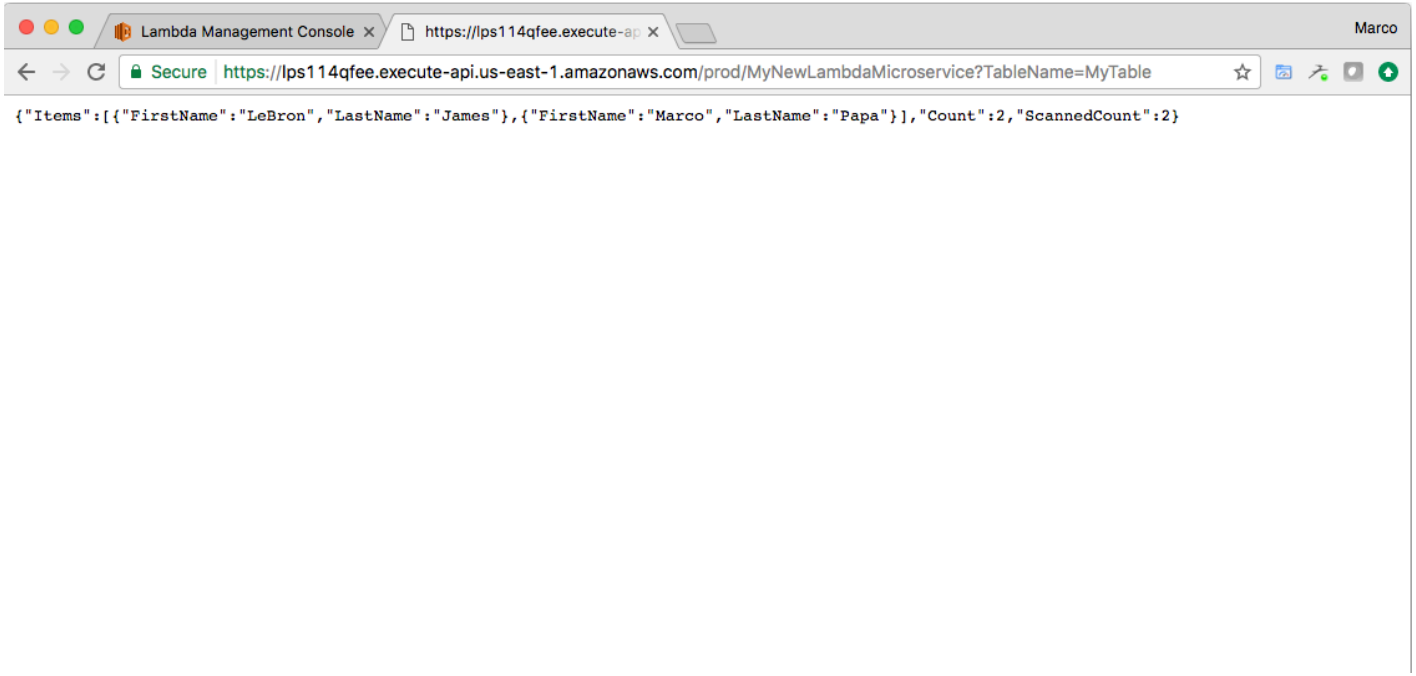
15. Click **Test**. Check the **Execution result** output, by clicking **Details**.

AWS Lambda (cont'd)



16. Click the **Triggers** tab. Click the arrow next to **Method: ANY**. Notice the value of **Invoke URL**. This is the entry point of the API Gateway for your new microservice.

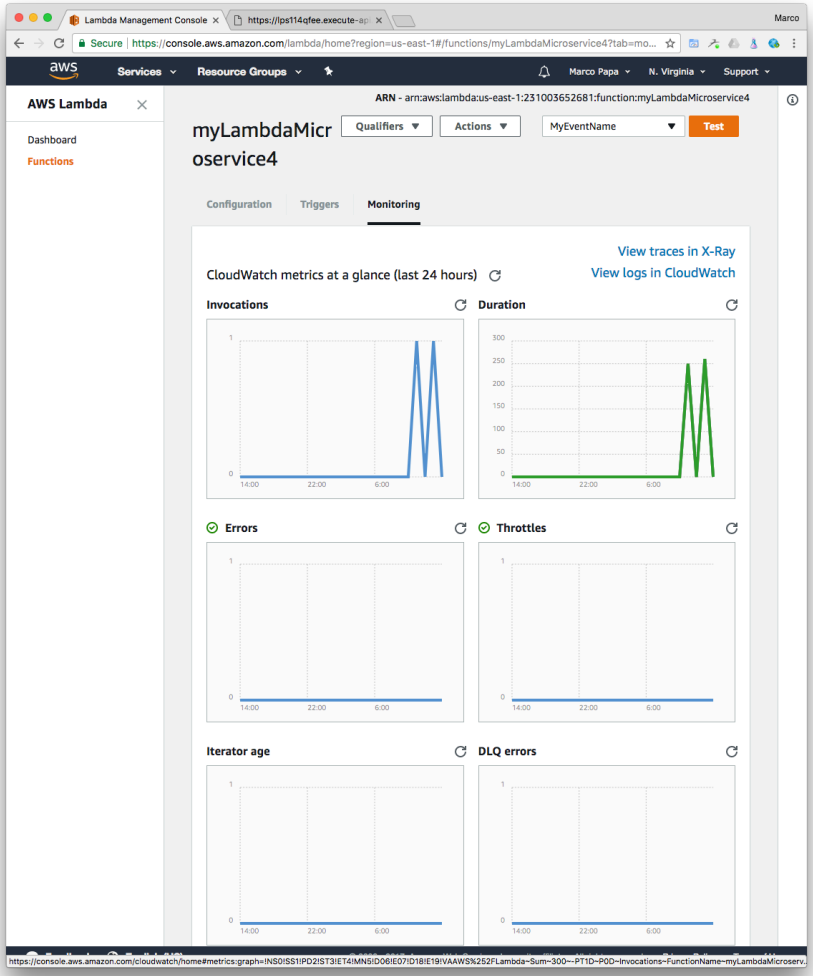
AWS Lambda (cont'd)



16. You can now execute the REST API from your browser as in:

https://lps114qfee.execute-api.us-east-1.amazonaws.com/prod/MyLambdaMicroservice?TableName=MyTable

AWS Lambda (cont'd)



17. Check out the **Monitoring** tab for CloudWatch metrics.