Getting Started with Serverless Architectures

Daniel Geske, Solutions Architect, AWS

29. März 2017
Agenda

Background
AWS Lambda
Amazon API Gateway
Serverless Architecture Patterns
Serverless Best Practices
Background

How serverless architecture patterns with AWS Lambda are the next evolution of application design
The Monolithic Architecture
The Service-Oriented Architecture
The Microservices Architecture
Tools to Help This Pattern are VAST

Web servers
Code libraries
Web service/application frameworks
Configuration management tools
API management platforms
Deployment patterns
CI/CD patterns
Containers
Etc. Etc. Etc.
AWS Has Helped Too!

Amazon EC2
Auto Scaling
Elastic Load Balancing
Amazon EC2 auto recovery
AWS Trusted Advisor
AWS Elastic Beanstalk
AWS OpsWorks
Amazon ECS
Etc. Etc. Etc.
But.....
many of these tools and innovations are still coupled to a shared dependency...
Servers (AAHHHHHHHHHHH!!)

**What size** servers are right for my budget?

How many users create **too much load** for my servers?

How much remaining **capacity** do my servers have?

How can I **detect** if a server has been **compromised**?

**How many** servers should I **budget** for?

Which **users** should have **access to** my servers?

How can I **control access from** my servers?

How will I keep my server **OS patched**?

How will new code be **deployed** to my servers?

How can I increase **utilization** of my servers?

When should I decide to **scale out** my servers?

What **size** server is right for my **performance**?

Should I **tune OS settings** to **optimize** my application?

Which **packages** should be **baked** into my **server images**?

When should I decide to **scale up** my servers?

How should I handle server **configuration changes**?

How will the **application** handle server **hardware failure**?
Architect to be Serverless

Fully managed
• No provisioning
• Zero administration
• High availability

Developer productivity
• Focus on the code that matters
• Innovate rapidly
• Reduce time to market

Continuous scaling
• Automatically
• Scale up and scale down
AWS Lambda
Serverless, event-driven compute service

Lambda = microservice without servers
Components of Lambda

- A Lambda function (that you write)
- An event source
- The AWS Lambda service
- The function networking environment
Lambda Function

- Your code (Java, NodeJS, Python)
- The IAM role that code assumes during execution
- The amount of memory allocated to your code (affects CPU and network as well)

A valid, complete Lambda function

```javascript
var AWS = require('aws-sdk');
var s3 = new AWS.S3();

exports.handler = function(event, context) {
    var params = {
        bucket: '[Input bucket name here]',
        Key: '[insert keyname here]',
        Body: '[object body]'  
    };
    s3.putObject(params, function(err, data) {
      if (err) {
        console.log(err, err.stack); // an error occurred
      } else {
        context.done();
        console.log(data);
      }
    });
};
```
Event Sources

• When should your function execute?
• Many AWS services can be an event source today:
  • Amazon S3
  • Amazon Kinesis
  • Amazon SNS
  • Amazon DynamoDB
  • Amazon CloudWatch
  • AWS Config Rules
  • Amazon Echo
  • Etc.
  • …and Amazon API Gateway (more later)
AWS Lambda

• Runs your function code without you managing or scaling servers.
• Provides an API to trigger the execution of your function.
• Ensures function is executed when triggered, in parallel, regardless of scale.
• Provides additional capabilities for your function (logging, monitoring).
Function Networking Environment

Default - a default network environment within VPC is provided for you
  • Access to the Internet always permitted to your function
  • No access to VPC-deployed assets

Customer VPC - Your function executes within the context of your own VPC.
  • Privately communicate with other resources within your VPC.
  • Familiar configuration and behavior with:
    – Subnets
    – Elastic network interfaces (ENIs)
    – EC2 security groups
    – VPC route tables
    – NAT gateway
“Hold on…” – you (maybe)
Lots of Existing Ways to Abstract Away Servers

SaaS
PaaS
MBaaS
*aaS
Application Engines/Platforms
What’s Unique About Lambda?

Abstraction at the code/function level (arbitrary, flexible, familiar)
The security model (IAM, VPC)
The pricing model
The community
Integration with the AWS service ecosystem!
  • Scale
  • Triggers
Example Serverless Architecture
PlayOn! Sports – Video Stream Processing

HLS

CloudFront

S3 Ingest

Cascading Lambda Functions

HQ Copy

480p Transcode

360p Transcode

Audio-only Transcode

Thumbnail

QOS Analytics

S3 Playback

CloudFront Streaming

Live stream

mobile client

VOD Stream

mobile client

http://www.slideshare.net/AmazonWebServices/arc308-the-serverless-company-using-aws-lambda
“But… in order to utilize Lambda, do I really need to architect event-driven applications?” – you (maybe)
SOA still works.
Amazon API Gateway
A Fully Managed Service for Your APIs

Create

Configure

Publish

Maintain

Monitor

Secure
The Serverless Architecture
Example: Mobile Backend for Social Media App

User posts status update

API GATEWAY
App makes REST API call to endpoint

Lambda is triggered

Lambda runs code to look up friends list and pushes status update notification to user's friends

SNS
Real-time Analytics Engine

Example: Analysis of Streaming Social Media Data

KINESIS
Social media stream is loaded into Kinesis in real-time.

Lambda is triggered

Lambda runs code that generates hashtag trend data and stores it in DynamoDB

DYNAMODB

Social media trend data immediately available for business users to query
Serverless Best Practices
AWS Lambda Best Practices

1. Limit your function size – especially for Java (starting the JVM takes time).
2. Node – remember execution is asynchronous.
3. Don’t assume function container reuse – but take advantage of it when it does occur.
4. Don’t forget about disk (500 MB /tmp directory provided to each function).
5. Use function aliases for release.
6. Use the included logger (include details from service-provided context).
7. Create custom metrics (operations-centric, and business-centric).
Amazon API Gateway Best Practices

1. Use mock integrations
2. Combine with Amazon Cognito for managed end user-based access control.
3. Use stage variables (inject API config values into Lambda functions for logging, behavior).
4. Use request/response mapping templates everywhere within reason, not passthrough.
5. Take ownership of HTTP response codes.
6. Use Swagger import/export for cross-account sharing.
Additional Best Practices

1. Use strategic, consumable naming conventions (Lambda function names, IAM roles, API names, API stage names, etc.).
2. Use naming conventions and versioning to create automation.
3. Externalize authorization to IAM roles whenever possible.
4. Least privilege and separate IAM roles.
5. Externalize configuration – DynamoDB is great for this.
6. Contact AWS Support before known large scaling events.
7. Be aware of service throttling, engage AWS support if so.
A Call to Action
Go build something!

Amazon API Gateway

AWS Lambda

Amazon DynamoDB
Thank you!