Serverless Development

AWS London Loft

Paul Maddox, Amazon Web Services
Specialist, Developer Technologies

September 2017

About me

Paul Maddox

Developer Technologies

Amazon Web Services

- 16 years of dev, sysadmin, and systems architecture background.
- 7 of 7 AWS certifications.
- Go/Java/C/Node.

Twitter: **@paulmaddox**
Email: **pmaddox@amazon.com**

What to expect from this session

- Overview of serverless development
- Building a serverless API
    - Development Frameworks
    - Deploying with AWS SAM
    - CI/CD with AWS CodeBuild/CodePipeline
    - Testing/debugging locally with AWS SAM Local
    - Security
        - Amazon Cognito User Pools
        - Customer Authorizers
- Q&A

A serverless world…

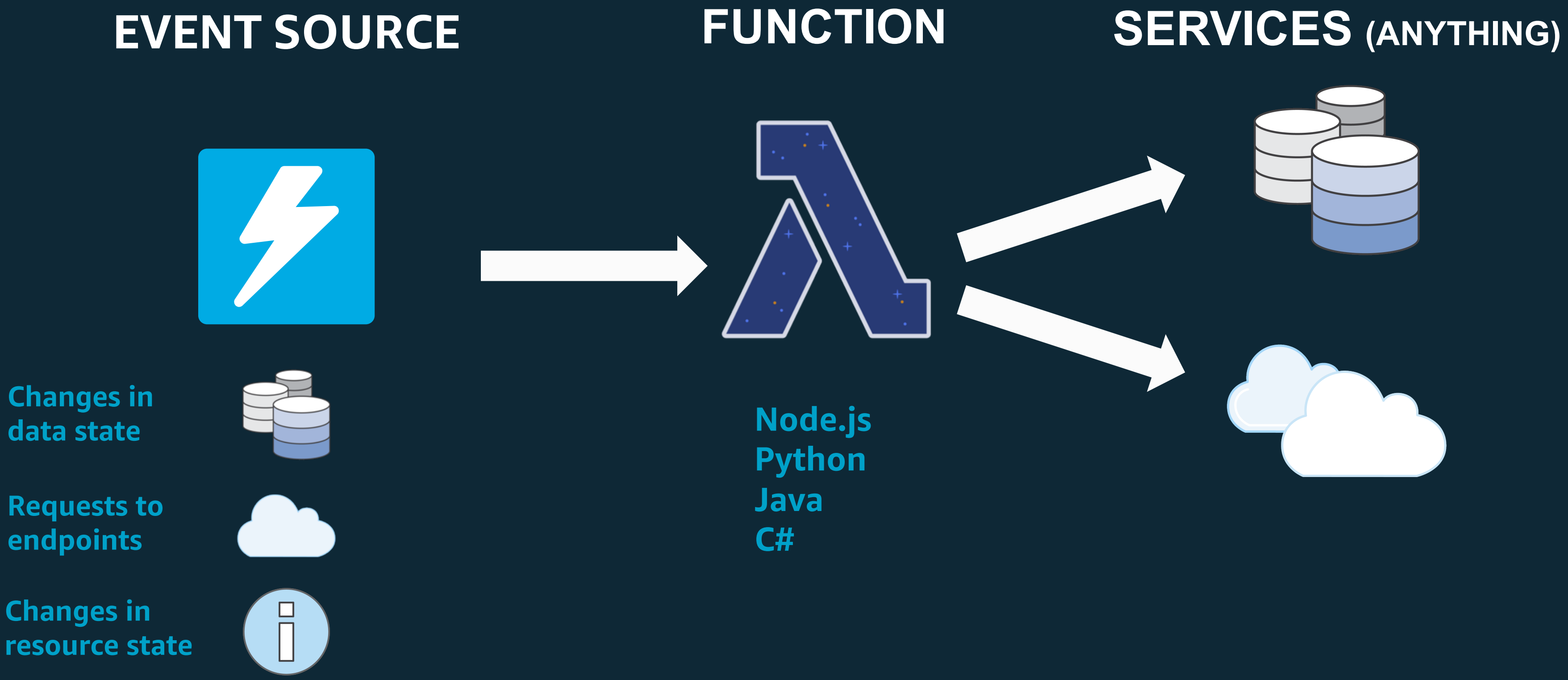
No servers to provision or manage

Scales with usage

Never pay for idle

Availability and fault tolerance built in

Event based architectures



EVENT SOURCE

FUNCTION

SERVICES (ANYTHING)

Changes in data state

Requests to endpoints

Changes in resource state

Node.js
Python
Java
C#

Building an API with Serverless

Frameworks

APEX

aws Chalice

Claudia.js

SERVERLESS

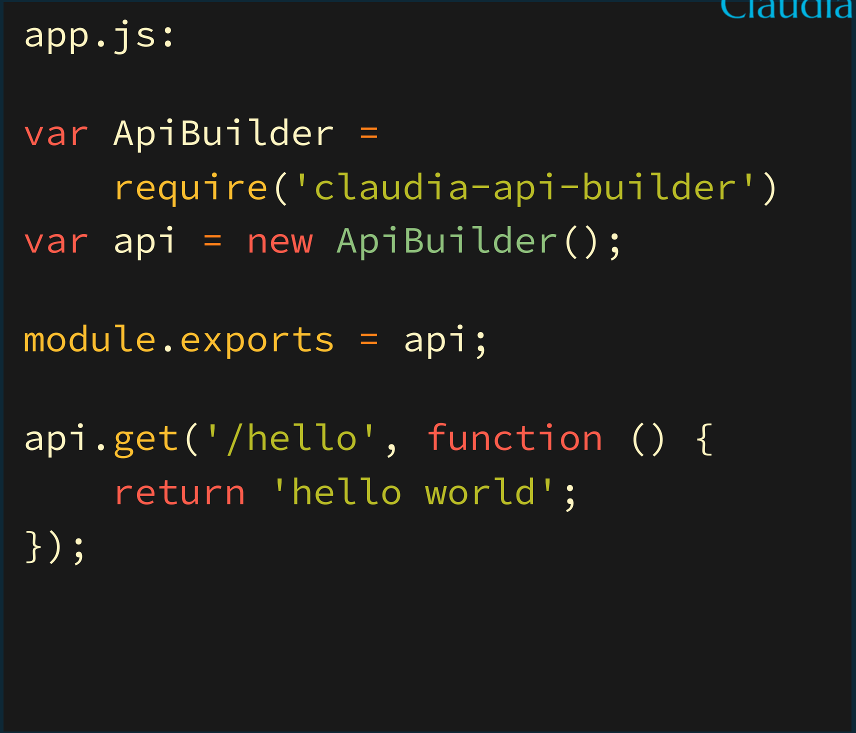
SPARTA

ClaudiaJS

Node.js framework for deploying projects to AWS Lambda and Amazon API Gateway

- Has sub projects for microservices, chat bots and APIs
- Simplified deployment with a single command
- Use standard NPM packages, no need to learn swagger
- Manage multiple versions

https://claudiajs.com
https://github.com/claudiajs/claudia



```
app.js:

var ApiBuilder =
    require('claudia-api-builder')
var api = new ApiBuilder();

module.exports = api;

api.get('/hello', function () {
    return 'hello world';
});
```

```
$ claudia create --region us-east-1 --api-module app
```

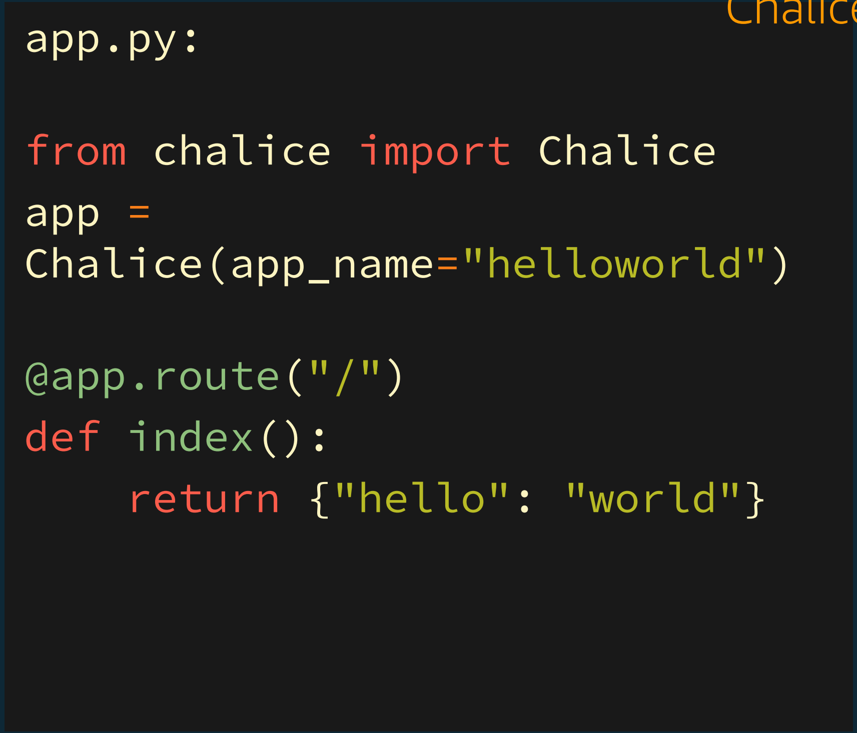
Chalice

Python serverless “microframework” for AWS Lambda and Amazon API Gateway

- A command line tool for creating, deploying, and managing your app
- A familiar and easy to use API for declaring views in python code
- Automatic Amazon IAM policy generation

https://github.com/aws/chalice

https://chalice.readthedocs.io



```
app.py:

from chalice import Chalice
app =
Chalice(app_name="helloworld")

@app.route("/")
def index():
    return {"hello": "world"}
```

$chalice deploy

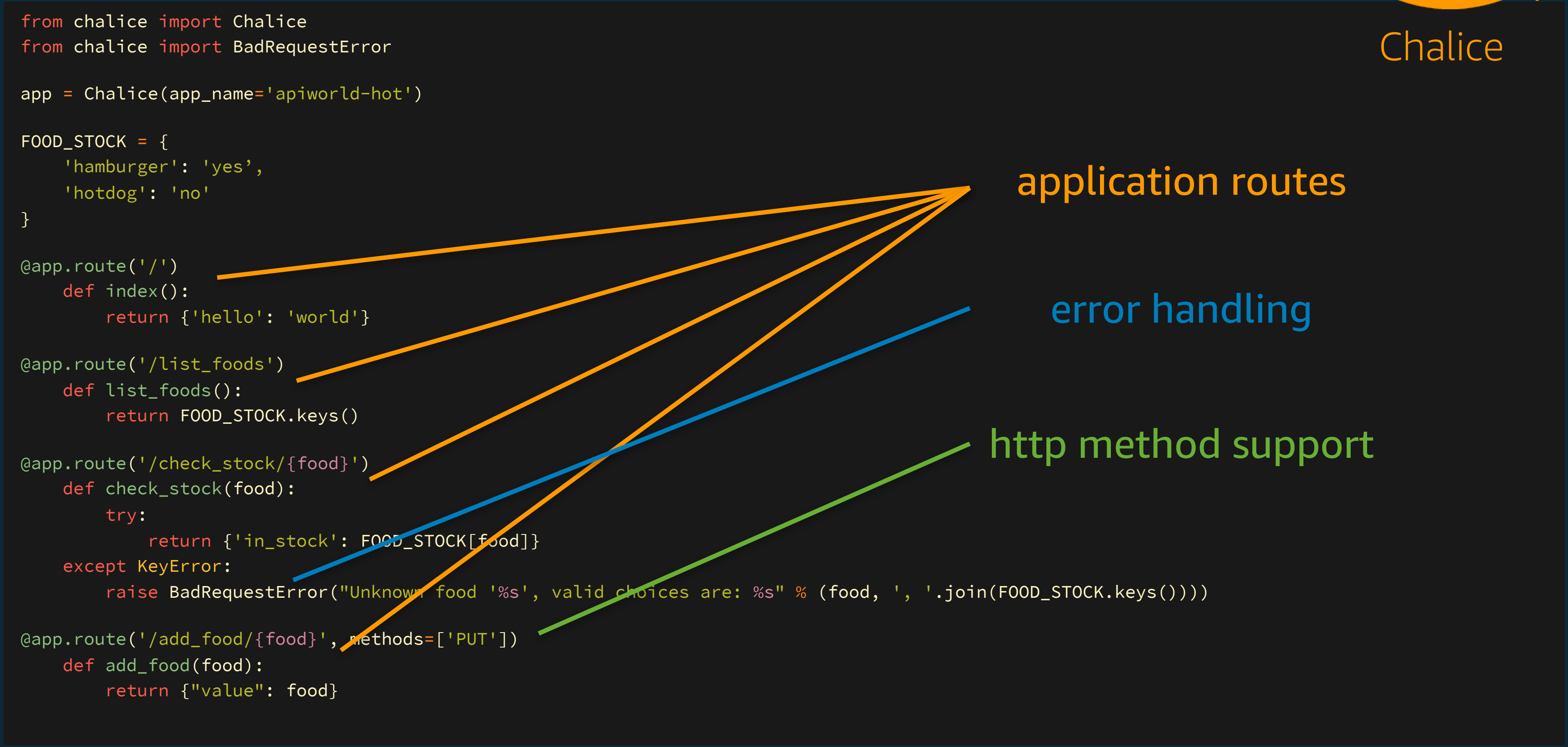
Chalice – a bit deeper

Chalice

```
from chalice import Chalice
from chalice import BadRequestError

app = Chalice(app_name='apiworld-hot')

FOOD_STOCK = {
    'hamburger': 'yes’,
    'hotdog': 'no'
}

@app.route('/')
    def index():
        return {'hello': 'world'}

@app.route('/list_foods')
    def list_foods():
        return FOOD_STOCK.keys()

@app.route('/check_stock/{food}')
    def check_stock(food):
        try:
            return {'in_stock': FOOD_STOCK[food]}
    except KeyError:
        raise BadRequestError("Unknown food '%s', valid choices are: %s" % (food, ', '.join(FOOD_STOCK.keys())))

@app.route('/add_food/{food}', methods=['PUT'])
    def add_food(food):
        return {"value": food}
```

Chalice – a bit deeper

Chalice



```
from chalice import Chalice
from chalice import BadRequestError

app = Chalice(app_name='apiworld-hot')

FOOD_STOCK = {
    'hamburger': 'yes’,
    'hotdog': 'no'
}

@app.route('/')
    def index():
        return {'hello': 'world'}

@app.route('/list_foods')
    def list_foods():
        return FOOD_STOCK.keys()

@app.route('/check_stock/{food}')
    def check_stock(food):
        try:
            return {'in_stock': FOOD_STOCK[food]}
    except KeyError:
        raise BadRequestError("Unknown food '%s', valid choices are: %s" % (food, ', '.join(FOOD_STOCK.keys())))

@app.route('/add_food/{food}', methods=['PUT'])
    def add_food(food):
        return {"value": food}
```

application routes

error handling

http method support

Meet SAM!

AWS Serverless Application Model (SAM)

CloudFormation extension optimized for serverless

New serverless resource types: functions, APIs, and tables

Supports anything CloudFormation supports

Open specification (Apache 2.0)

https://github.com/awslabs/serverless-application-model

SAM template

```
AWSTemplateFormatVersion: "2010-09-09"
Transform: AWS::Serverless-2016-10-31
Resources:
  GetHtmlFunction:
  Type: AWS::Serverless::Function
  Properties:
    CodeUri: s3://sam-demo-bucket/todo_list.zip
    Handler: index.gethtml
    Runtime: nodejs4.3
    Policies: AmazonDynamoDBReadOnlyAccess
    Events:
      GetHtml:
        Type: Api
        Properties:
          Path: /{proxy+}
          Method: ANY

ListTable:
  Type: AWS::Serverless::SimpleTable
```

SAM template

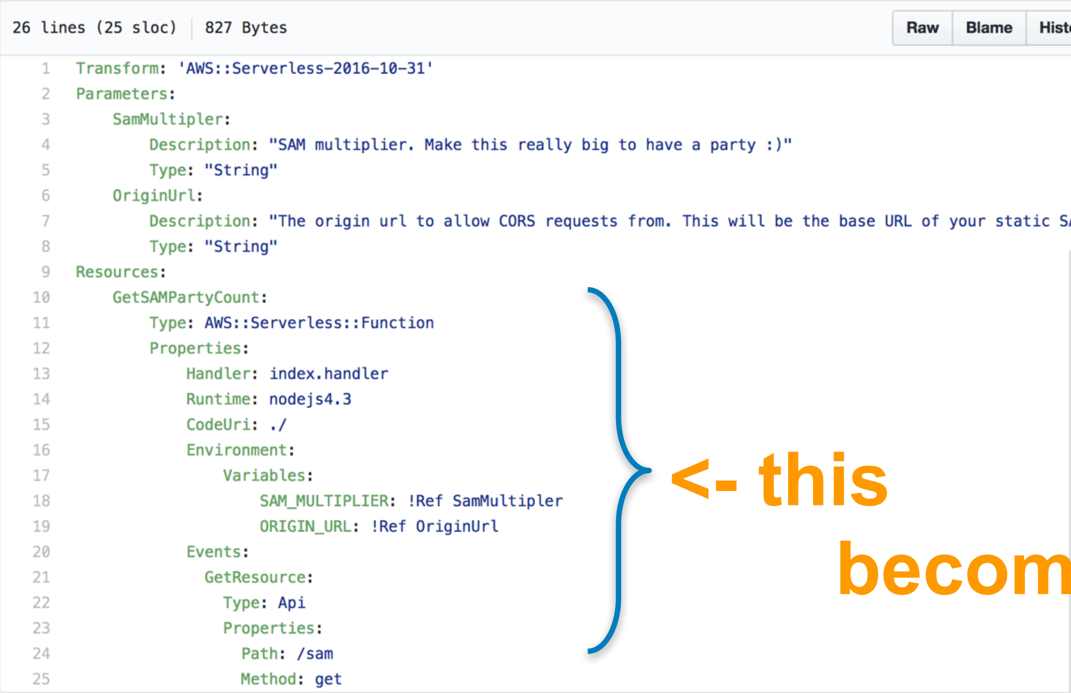
```
AWSTemplateFormatVersion: "2010-09-09"
Transform: AWS::Serverless-2016-10-31
Resources:
  GetHtmlFunction:
  Type: AWS::Serverless::Function
  Properties:
    CodeUri: s3://sam-demo-bucket/todo_list.zip
    Handler: index.gethtml
    Runtime: nodejs4.3
    Policies: AmazonDynamoDBReadOnlyAccess
    Events:
      GetHtml:
        Type: Api
        Properties:
          Path: /{proxy+}
          Method: ANY

ListTable:
  Type: AWS::Serverless::SimpleTable
```

Tells CloudFormation this is a SAM template it needs to “transform”

Creates a Lambda function with the referenced managed IAM policy, runtime, code at the referenced zip location, and handler as defined. Also creates an API Gateway and takes care of all mapping/permissions necessary

Creates a DynamoDB table with 5 Read & Write units

SAM template



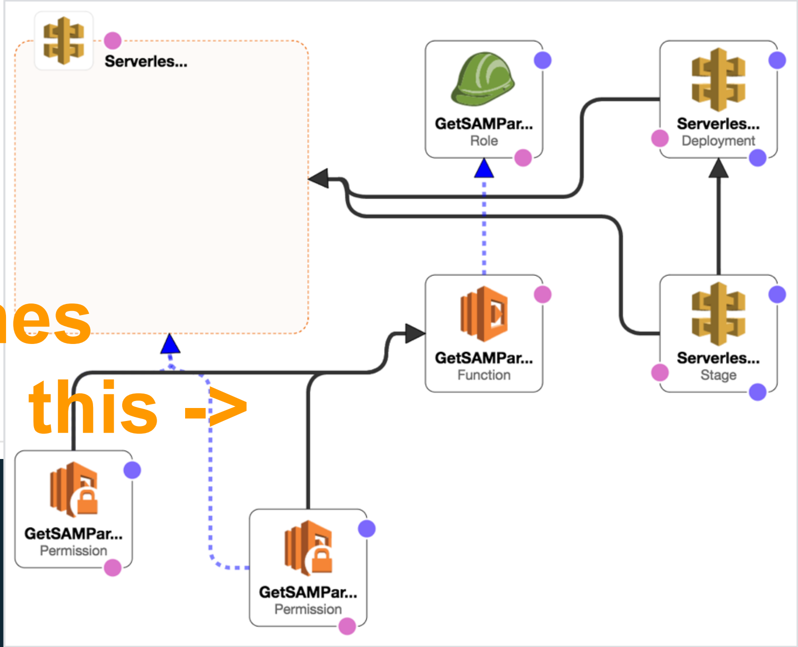
26 lines (25 sloc) | 827 Bytes

```yaml
Transform: 'AWS::Serverless-2016-10-31'
Parameters:
    SamMultipler:
        Description: "SAM multiplier. Make this really big to have a party :)"
        Type: "String"
    OriginUrl:
        Description: "The origin url to allow CORS requests from. This will be the base URL of your static SAM website."
        Type: "String"
Resources:
    GetSAMPartyCount:
        Type: AWS::Serverless::Function
        Properties:
            Handler: index.handler
            Runtime: nodejs4.3
            CodeUri: ./
            Environment:
                Variables:
                    SAM_MULTIPLIER: !Ref SamMultipler
                    ORIGIN_URL: !Ref OriginUrl
            Events:
              GetResource:
                Type: Api
                Properties:
                  Path: /sam
                  Method: get
```

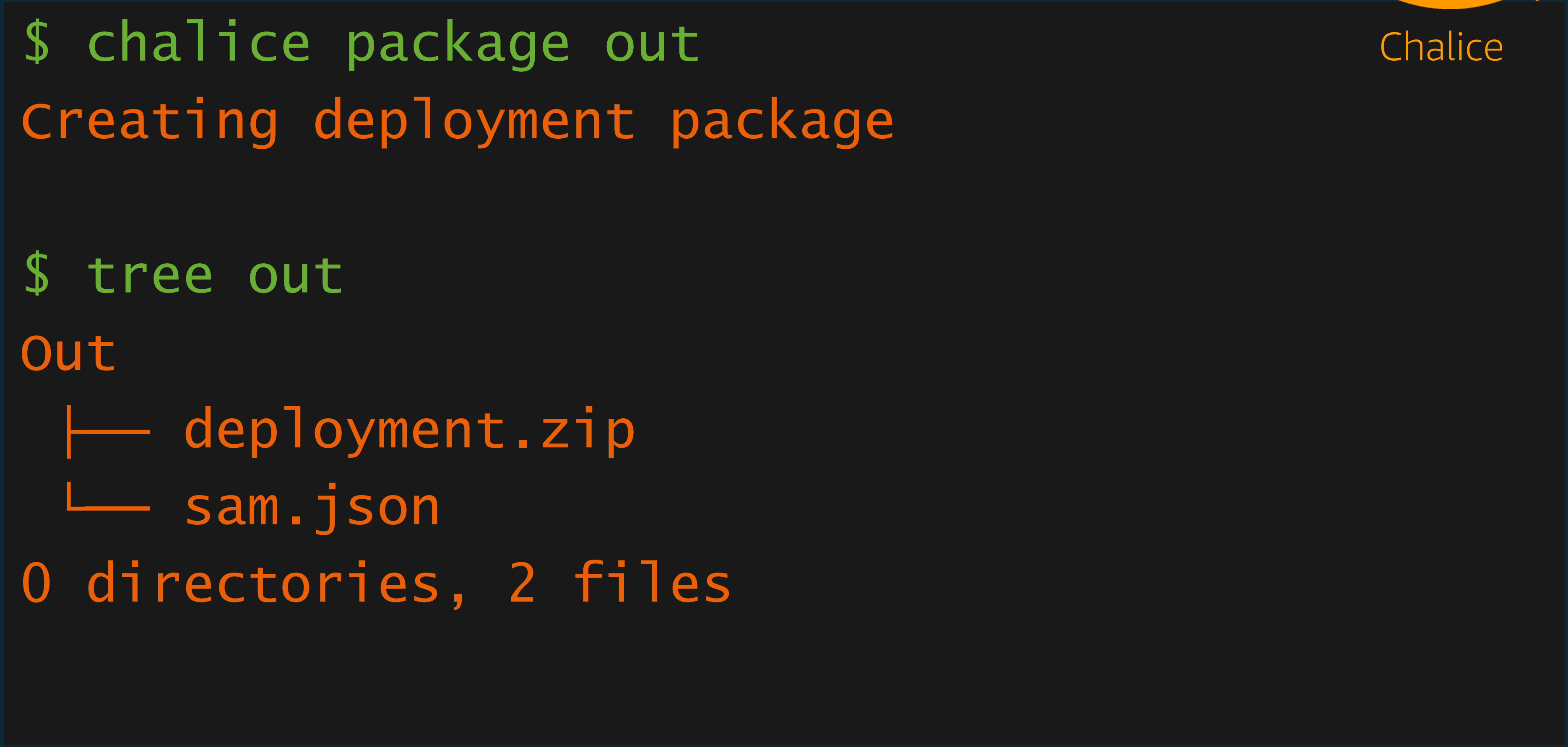
<- this

becomes

this ->



Chalice – generating a SAM template



Chalice

```
$ chalice package out
Creating deployment package


$ tree out
Out
  ├── deployment.zip
  └── sam.json
0 directories, 2 files
```

Introducing SAM Local

CLI tool for local testing of serverless apps

Works with Lambda functions and “proxy-style” APIs

Response object and function logs available on your local machine

Currently supports Java, Node.js and Python

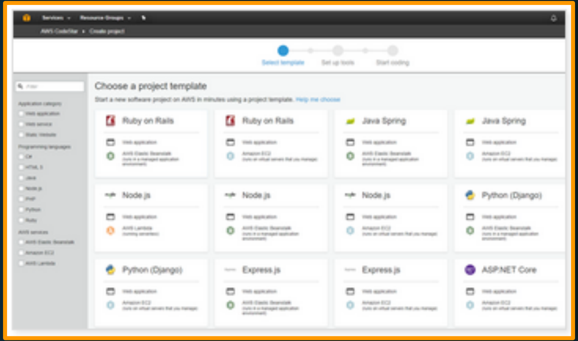
https://github.com/awslabs/aws-sam-local

```
If(!feelingLucky) {
    demogods.pray();
}
demo.start();
```

Build an App with AWS CodeStar and receive **$50 in AWS Credits**

1

Build your app in the AWS CodeStar console



Services ▾ Resource Groups ▾

AWS CodeStar › Create project

Select template — Set up tools — Start coding

Filter

Application category
- □ Web application
- □ Web service
- □ Static Website

Programming languages
- □ C#
- □ HTML 5
- □ Java
- □ Node.js
- □ PHP
- □ Python
- □ Ruby

AWS services
- □ AWS Elastic Beanstalk
- □ Amazon EC2
- □ AWS Lambda

Choose a project template

Start a new software project on AWS in minutes using a project template. Help me choose

Ruby on Rails: Web application; AWS Elastic Beanstalk (runs it in a managed application environment)

Ruby on Rails: Web application; Amazon EC2 (runs on virtual servers that you manage)

Java Spring: Web application; AWS Elastic Beanstalk (runs it in a managed application environment)

Java Spring: Web application; Amazon EC2 (runs on virtual servers that you manage)

Node.js: Web application; AWS Lambda (running serverless)

Node.js: Web application; AWS Elastic Beanstalk (runs it in a managed application environment)

Node.js: Web application; Amazon EC2 (runs on virtual servers that you manage)

Python (Django): Web application; AWS Elastic Beanstalk (runs it in a managed application environment)

Python (Django): Web application; Amazon EC2 (runs on virtual servers that you manage)

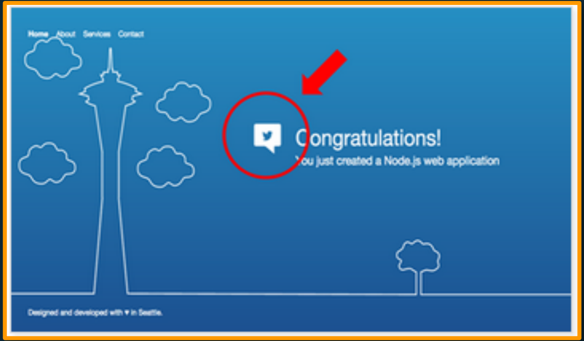
Express.js: Web application; AWS Elastic Beanstalk (runs it in a managed application environment)

Express.js: Web application; Amazon EC2 (runs on virtual servers that you manage)

ASP.NET Core: Web application; Amazon EC2 (runs on virtual servers that you manage)

2

Click the tweet icon in the console to share your app on Twitter



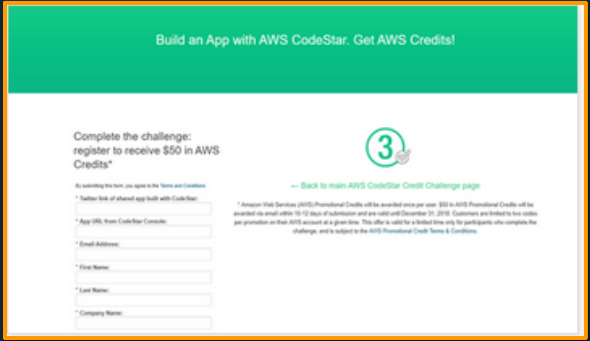
Congratulations!

You just created a Node.js web application

Designed and developed with ♥ in Seattle.

3

Register using the link below to receive AWS Credits*



Build an App with AWS CodeStar. Get AWS Credits!

Complete the challenge: register to receive $50 in AWS Credits*

By submitting this form, you agree to the Terms and Conditions

* Twitter link of shared app built with CodeStar:

* App URL from CodeStar Console:

* Email Address:

* First Name:

* Last Name:

* Company Name:

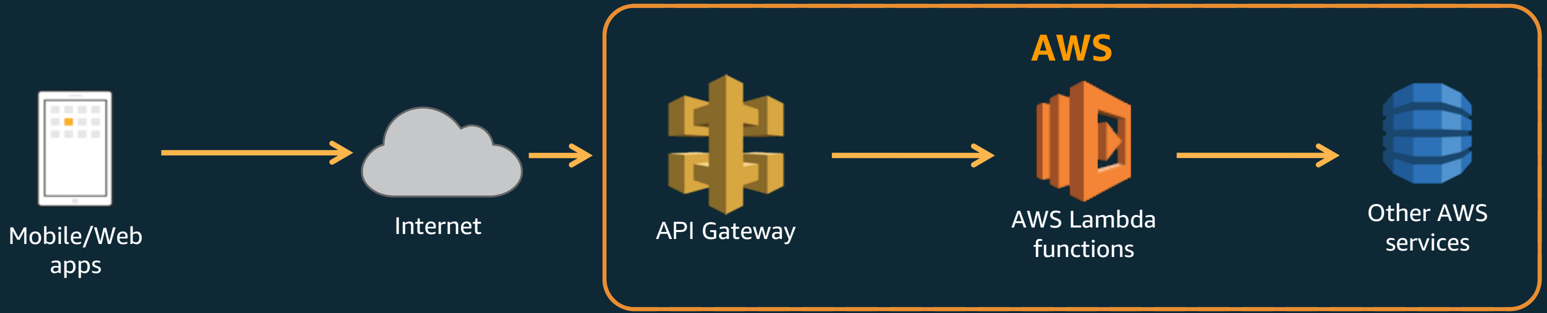
← Back to main AWS CodeStar Credit Challenge page

* Amazon Web Services (AWS) Promotional Credits will be awarded once per user. $50 in AWS Promotional Credits will be awarded via email within 10-12 days of submission and are valid until December 31, 2018. Customers are limited to two codes per promotion on their AWS account at a given time. This offer is valid for a limited time only for participants who complete the challenge, and is subject to the AWS Promotional Credit Terms & Conditions.

Go to https://aws.amazon.com/codestar/codestar-credit-challenge/ for details

* Amazon Web Services (AWS) Promotional Credits will be awarded once per user for a limited time only upon successful completion of the challenge. $50 in AWS Promotional Credits will be awarded via email within 10-12 days of submission and are valid until December 31, 2018. Customers are limited to having two promotional credits on their AWS account at a given time.

What we just deployed…



Amazon API Gateway

Create a unified API frontend for multiple micro-services

DDoS protection and throttling for your backend

Authenticate and authorize requests to a backend

Throttle, meter, and monetize API usage by 3rd party developers

AWS Lambda

Bring your own code

- Node.js, Java, Python, C#
- Bring your own libraries (even native ones)

Simple resource model

- Select power rating from 128 MB to 1.5 GB
- CPU and network allocated proportionately

Flexible use

- Synchronous or asynchronous
- Integrated with other AWS services

Flexible authorization

- Securely grant access to resources and VPCs
- Fine-grained control for invoking your functions

AWS Lambda

Authoring functions

- WYSIWYG editor or upload packaged .zip
- Third-party plugins (Eclipse, Visual Studio)

Monitoring and logging

- Metrics for requests, errors, and throttles
- Built-in logs to Amazon CloudWatch Logs

Programming model

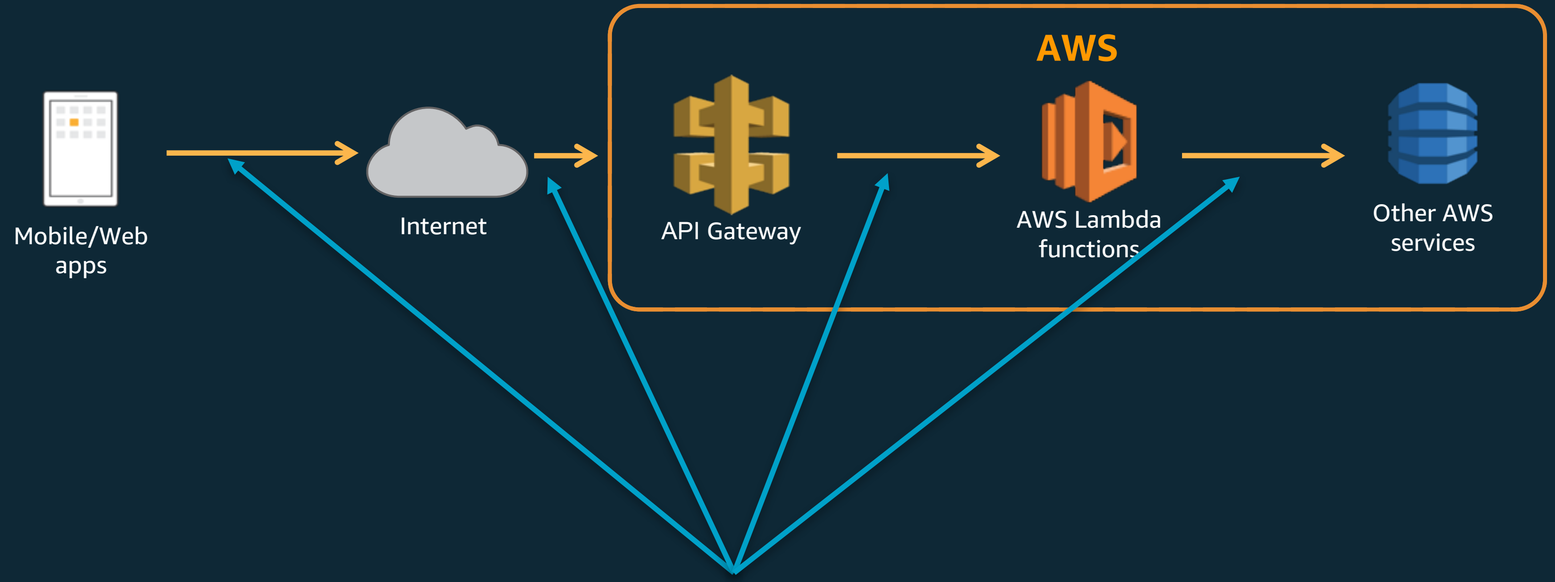
- Use processes, threads, /tmp, sockets normally
- AWS SDK built in (Python and Node.js)

Stateless

- Persist data using external storage
- No affinity or access to underlying infrastructure

Security is job zero.

The serverless API stack



places where we can secure our application

Amazon API Gateway Security

Several mechanisms for adding Authz/Authn to our API:

- IAM Permissions
    - Use IAM policies and AWS credentials to grant access
- Custom Authorizers
    - Use Lambda to validate a bearer token (OAuth or SAML as examples) or request parameters and grant access
- Cognito User Pools
    - Create a completely managed user management system

Cognito User Pools

1

Serverless Authentication and User Management

Add user sign-up and sign-in easily to your mobile and web apps without worrying about server infrastructure

2

Managed User Directory

Launch a simple, secure, low-cost, and fully managed service to create and maintain a user directory that scales to 100s of millions of users

3

Enhanced Security Features

Verify phone numbers and email addresses and offer multi-factor authentication

Cognito User Pools - User Flows

User Sign-Up and Sign-In

User Profile Data

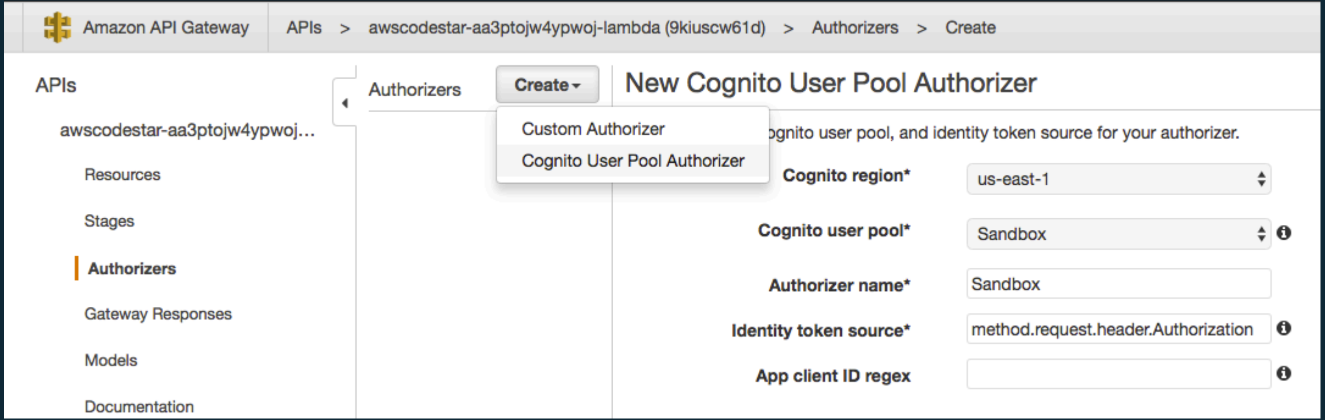
Forgot Password

Token Based Authentication

Email or Phone Number Verification

SMS Multifactor Authentication

Cognito User Pool Authorizer


Amazon API Gateway
APIs > awscodestar-aa3ptojw4ypwoj-lambda (9kiuscw61d) > Authorizers > Create
APIs
awscodestar-aa3ptojw4ypwoj...
Resources
Stages
Authorizers
Gateway Responses
Models
Documentation
Authorizers
Create
Custom Authorizer
Cognito User Pool Authorizer

New Cognito User Pool Authorizer

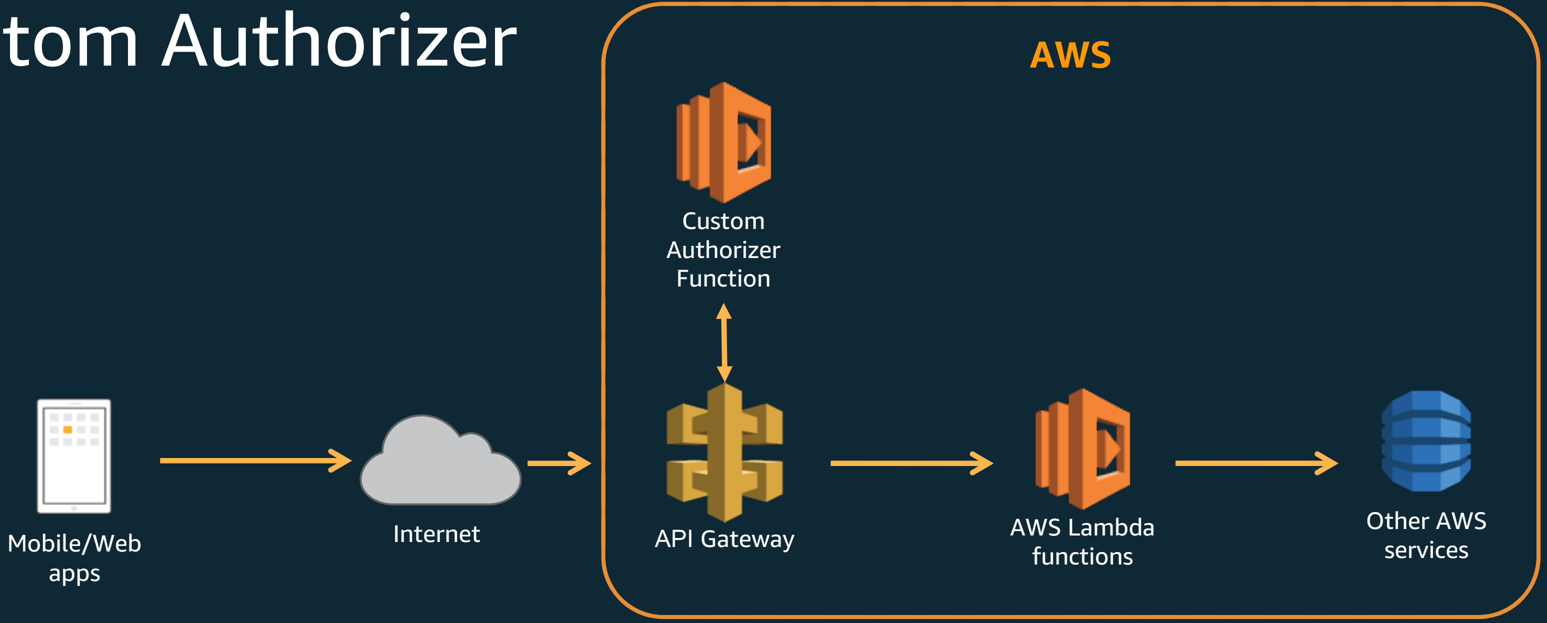
ognito user pool, and identity token source for your authorizer.

Cognito region*	us-east-1
Cognito user pool*	Sandbox
Authorizer name*	Sandbox
Identity token source*	method.request.header.Authorization
App client ID regex	

Super easy! Supports authentication, but not authorization.

E.g. you can lock down an API to Cognito User Pool users, but you don’t get fine grained control over who can access which API resources.

Custom Authorizer



Super flexible! Supports authentication and authorization.

- Function input: HTTP headers (e.g. Authorization header)
- Function output: Policy (e.g. **allow** GET /{userid}/profile, **deny** GET /admin)
- Result is cached for the input parameters (300 seconds default)

Chalice – adding Cognito User Pools

Chalice

```
from chalice import Chalice
from chalice import BadRequestError
from chalice import CognitoUserPoolAuthorizer

app = Chalice(app_name='apiworld-hot')

authorizer = CognitoUserPoolAuthorizer( 'MyPool', provider_arns=['arn:aws:cognito:...:userpool/name'])

...
...


@app.route('/list_foods')
    def list_foods():
        return FOOD_STOCK.keys()

@app.route('/check_stock/{food}’, methods=['GET'], authorizer=authorizer)
    def check_stock(food):
        try:
            return {'in_stock': FOOD_STOCK[food]}
    except KeyError:
        raise BadRequestError("Unknown food '%s', valid choices are: %s" % (food, ', '.join(FOOD_STOCK.keys())))

@app.route('/add_food/{food}', methods=['PUT'], authorizer=authorizer)
    def add_food(food):
        return {"value": food}
```

Chalice – adding Cognito User Pools

Chalice

```
from chalice import Chalice
from chalice import BadRequestError
from chalice import CognitoUserPoolAuthorizer

app = Chalice(app_name='apiworld-hot')

authorizer = CognitoUserPoolAuthorizer( 'MyPool', provider_arns=['arn:aws:cognito:...:userpool/name'])

...
...


@app.route('/list_foods')
    def list_foods():
        return FOOD_STOCK.keys()

@app.route('/check_stock/{food}’, methods=['GET'], authorizer=authorizer)
    def check_stock(food):
        try:
            return {'in_stock': FOOD_STOCK[food]}
    except KeyError:
        raise BadRequestError("Unknown food '%s', valid choices are: %s" % (food, ', '.join(FOOD_STOCK.keys()))

@app.route('/add_food/{food}', methods=['PUT'], authorizer=authorizer)
    def add_food(food):
        return {"value": food}
```

adding authorization

authorization required for certain routes/methods

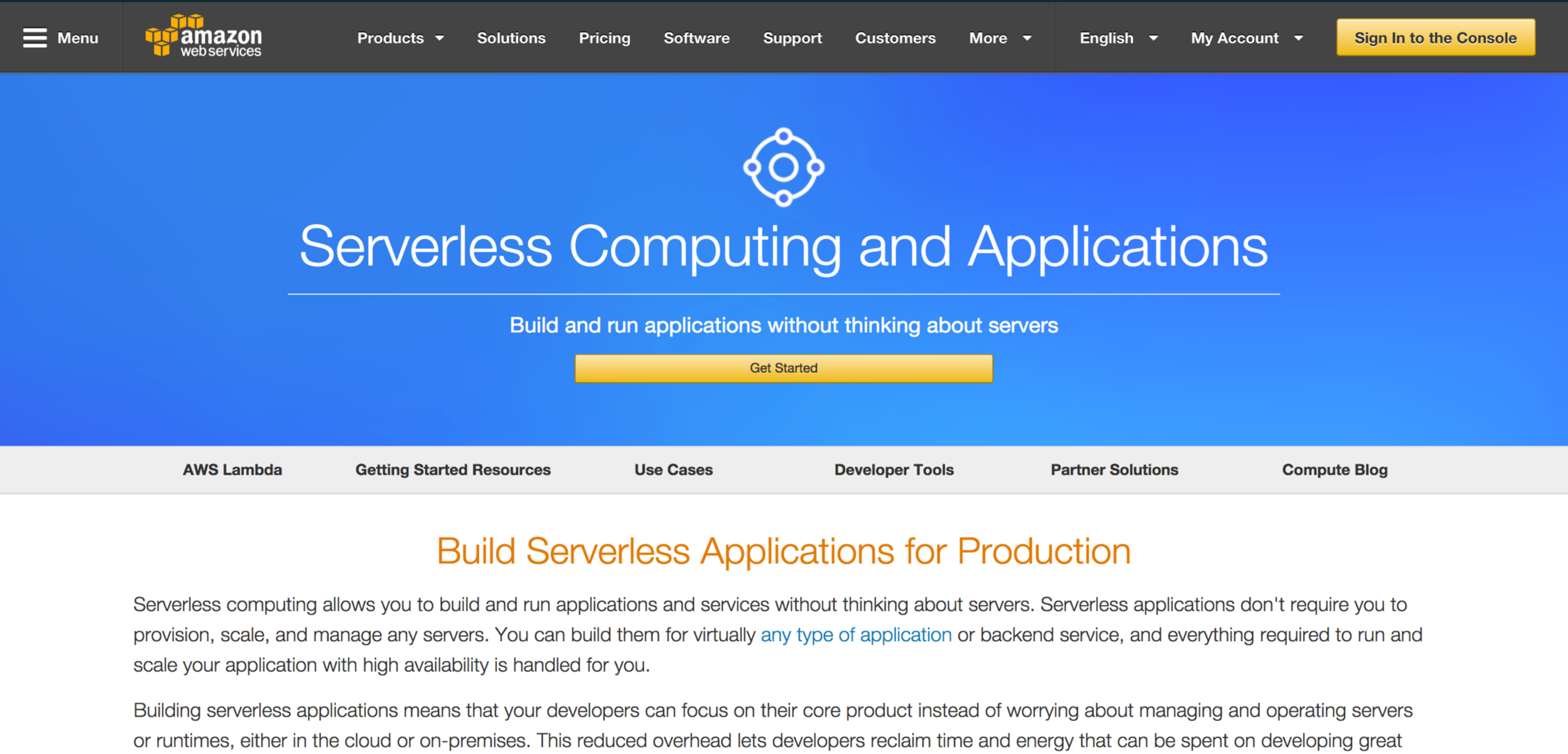
Summary

It’s never been easier to build and launch APIs!

Serverless APIs:

- **No management of servers**
- **Pay for what you use** and not for idle resources!
- **Instantly scale up** without turning any knobs or provisioning any resources
- Tooling to **get started in minutes** with incredibly minimal code needed
- **Built in high availability** built into multiple places in the application stack
- **Authentication and Authorization** built into multiple places in the application stack

aws.amazon.com/serverless



Serverless Computing and Applications

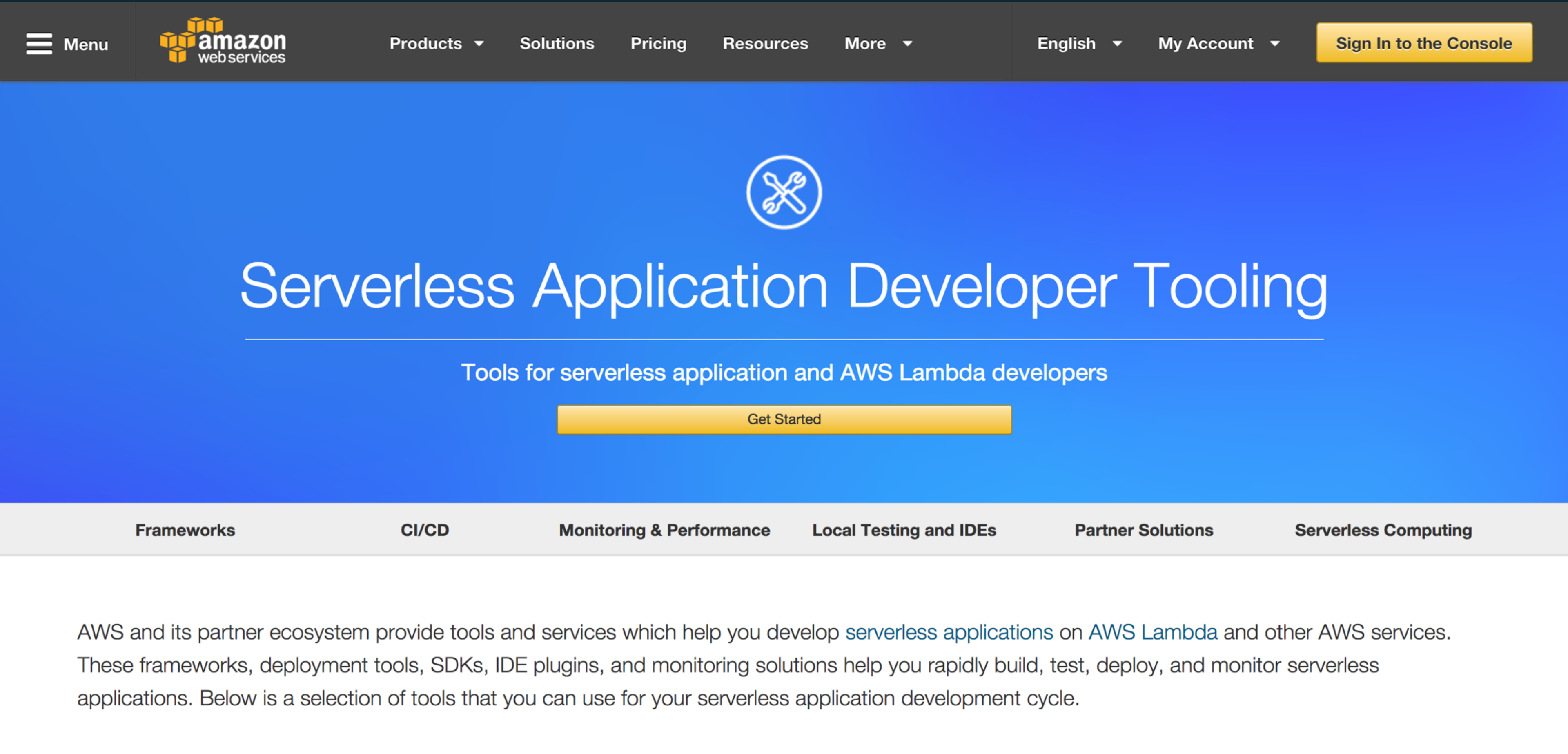
Build and run applications without thinking about servers

Build Serverless Applications for Production

Serverless computing allows you to build and run applications and services without thinking about servers. Serverless applications don't require you to provision, scale, and manage any servers. You can build them for virtually any type of application or backend service, and everything required to run and scale your application with high availability is handled for you.

Building serverless applications means that your developers can focus on their core product instead of worrying about managing and operating servers or runtimes, either in the cloud or on-premises. This reduced overhead lets developers reclaim time and energy that can be spent on developing great

aws.amazon.com/serverless/developer-tools



Serverless Application Developer Tooling

Tools for serverless application and AWS Lambda developers

AWS and its partner ecosystem provide tools and services which help you develop serverless applications on AWS Lambda and other AWS services. These frameworks, deployment tools, SDKs, IDE plugins, and monitoring solutions help you rapidly build, test, deploy, and monitor serverless applications. Below is a selection of tools that you can use for your serverless application development cycle.

Thank you