

This talk is about explaining how serverless architecture and functions as a service work, and how you can build them into your existing software infrastructure. I will also explain what microservices are, and decide whether microservices or serverless operations are well-suited for your architecture or not. We start with a couple of definitions of microservices and serverless.

Serverless is a type of architecture where the servers (physical or in the cloud) cease to exist for the developer and instead the code runs in "execution environments" that are managed by suppliers such as Amazon, Google, IBM, etc.

Serverless computing is the practice of building and **running services and applications without having to worry about provisioning and managing servers**. Serverless computing has been a popular topic the past couple years, and with respect to Python, there have been various different frameworks and tools released for developing and managing your Python serverless applications.

Microservices vs Serverless

This is graphical view from google trends where we can see the evolution of both terms. The term serverless began to be known more at the beginning of 2017. Now days, microservices it's a more used term than serverless, but in next years maybe serverless will increase, can equal, even put over microservices.

Microservices Tornado and twisted

A microservice architecture puts each functionality into a separate service and scale by distributing these services across servers.

Microservice architecture exists when your application is divided in **small responsibilities blocks**, those blocks doesn't know each other, they only have a common point of communication, generally a message queue

If you are building microservices where **increasing the number of concurrent requests** you can hold is important just use an **asynchronous** framework like **Tornado** <https://www.tornadoweb.org/en/stable/> or **Twisted** <https://twistedmatrix.com/trac/>.

Asynchronous calls with asyncio and aiohttp

In microservices architectures, asynchronous calls play a fundamental role when a process used to be performed in a single application now implicates several microservices. **Asynchronous calls can be as simple as a separate thread or process within a microservice base application.**

In this example we are using asyncio <https://docs.python.org/3/library/asyncio.html> and aiohttp <https://aiohttp.readthedocs.io/en/stable/> as the main modules for doing these tasks in our applications.

```
import asyncio
import aiohttp

@asyncio.coroutine
def fetch_page(url):
    response = yield from aiohttp.request('GET', url)
    body = yield from response.read()
    return body

content = asyncio.get_event_loop().run_until_complete(
    fetch_page('http://python.org'))
print(content)
```

REST API Development

One of the principles of this style of architecture is that the services should tend to be as cohesive as possible, minimizing the maximum coupling between them. Communication between the services must be implemented through lightweight mechanisms, preferably HTTP-based APIs, such as RESTful/JSON APIs.

These agnostic communication systems open the door for us to develop each microservice with the language and tools that suit us in each case. For example, if in a specific microservice performance is a critical aspect, we can choose low level languages that allow us to fine-tune this aspect like C, C ++ or Go.

However, in other cases we will be interested in choosing languages of high or level or that implement certain particular characteristics, such as Python, Ruby, Groovy or Scala, that allow us to move faster or adapt better to the service that should be implemented.

On the other hand, there are excellent frameworks and libraries very established and oriented to the development of REST APIs and communication systems based on HTTP, which is perfect for the construction of microservices.

In python we can find frameworks like **django rest framework** <https://www.django-rest-framework.org/> and **flask** <https://palletsprojects.com/p/flask/>. These frameworks are oriented to the **development of REST APIs** and communication systems based on HTTP, which is perfect for the construction of microservices.

django

REST

framework

These two frameworks are the kings in the Python world to create web projects. In my opinion, if we have to select one of these I think the best option select **Flask**, since **it fits perfectly with the principle of unique responsibility in microservice architecture**.

Flask also allows to be more all terrain. For example, if we want to create an **API Management**, working with **non-relational databases** or other systems such as **GraphQL**.

Performance comparison

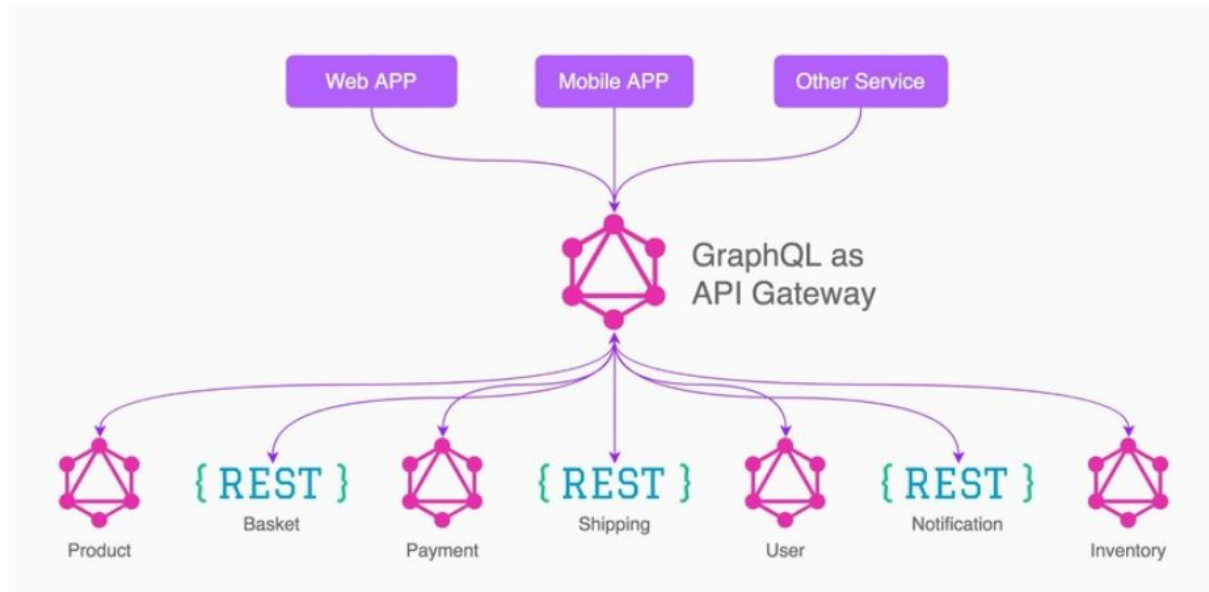
We must keep in mind that when we talk about microservices it is usually accompanied by a greater consumption of memory and resources. From a performance point of view in terms of memory,cpu and response times **Flask turned out to consume 8% less memory and have 6% response times faster than Django**.

Microservices with Graphql

Another type of architecture we find in microservices is the use of GraphQL. GraphQL <https://graphql.org> brings a microservices arrangement, such as data owner separation, granular data control, parallel execution and service caching.

Another benefit of adopting GraphQL is the fact that you can fundamentally assert **greater control** over the **data loading** process. Because the process for data loaders goes into its own endpoint, **you can control in a granular way how data is transferred**.

GraphQL provides an architecture with microservice-oriented design functions more agile, more responsively, and, more adherent to the design requirements.



Graphene-Python

Graphql is designed to replace REST APIs and we can use this architecture within python application thanks to libraries as Graphene-python <https://graphene-python.org/>.

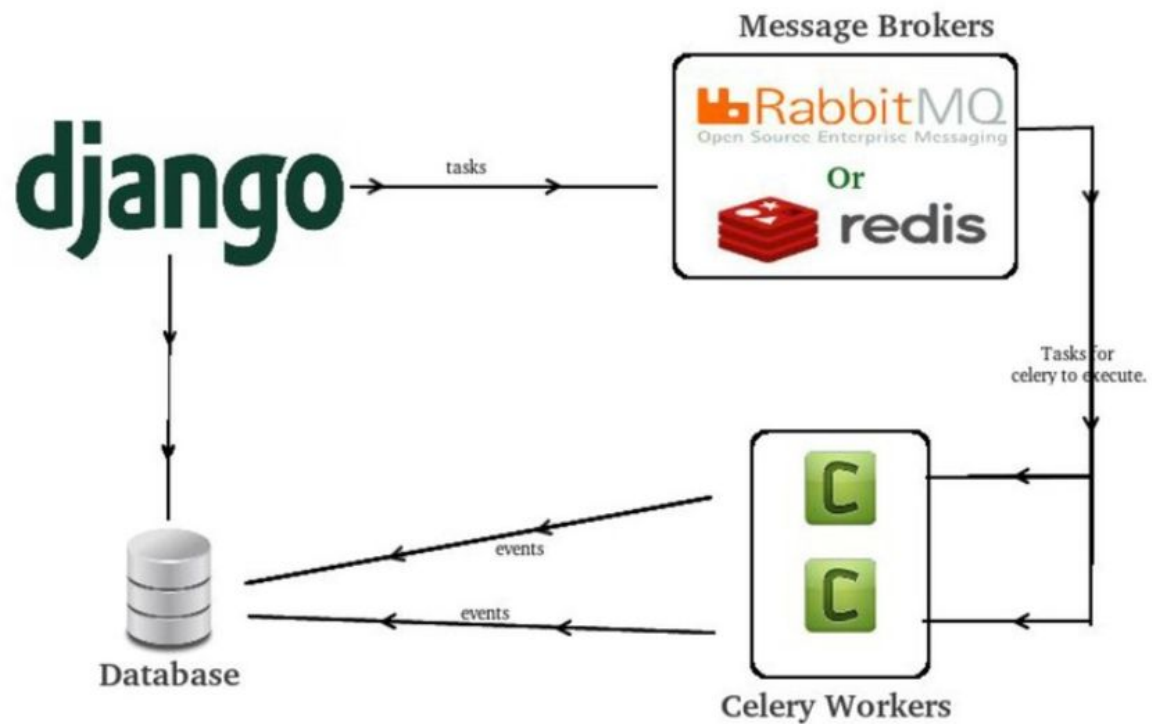
Graphene-Python is a library for building GraphQL APIs in Python easily and **it can integrate with applications based in django, flask and sqlalchemy**. Graphene allows you to define your models, define the attributes in your model, and also parse queries to get the data for those particular models.

In the same way that we have django ORM we can use graphene to define our models and diagrams and interact with a database, whether relational or not relational. We have other projects like **graphene-django** <https://docs.graphene-python.org/projects/django/en/latest/> that can help to replace your current REST API with a Graphql API.

Celery

We can also implement an architecture based in Celery <http://www.celeryproject.org/> and we can use redis or rabbitmq as a message broker. Celery is an asynchronous task queue based on distributed message passing.

Celery it is focused on real-time operation and is useful for background task processing and deferred execution in Django. **Task queues are used to distribute work across workers.**



ZeroMQ

ZeroMQ <http://zeromq.org/> provides building blocks to develop a scalable distributed systems on top of sockets. We can build a simple PUB-SUB microservice with ZeroMQ and Flask.

ZeroMQ server

In this example the response of service call is published as a message. This can be our zeromq server o publisher.

```
_context = zmq.Context()
_publisher = _context.socket(zmq.PUB)
url = 'tcp://{}:{}'.format(HOST, PORT)

def publish_message(message):
    try:
        _publisher.bind(url)
        time.sleep(1)
        myjson = json.dumps(message)
        _publisher.send(myjson)
    except Exception as e:
        print "error {}".format(e)
    finally:
        _publisher.unbind(url)
```

SERVER

ZeroMQ client

In this example the ZClient its subscribe to the channel where the server is sending messages. This can be our zeromq client o subscriber.

```

class ZClient(object):

    def __init__(self, host=HOST, port=PORT):
        """Initialize Worker"""
        self.host = host
        self.port = port
        self._context = zmq.Context()
        self._subscriber = self._context.socket(zmq.SUB)
        print "Client Initiated"

    def receive_message(self):
        """Start receiving messages"""
        print "receive_message"
        self._subscriber.connect('tcp://{}:{:}'.format(self.host, self.port))
        self._subscriber.setsockopt(zmq.SUBSCRIBE, b"")

        while True:
            print 'listening on tcp://{}:{:}'.format(self.host, self.port)
            message = self._subscriber.recv()
            print message
            logging.info('{} - {}'.format(message, time.strftime("%Y-%m-%d %H:%M")))

if __name__ == '__main__':
    zs = ZClient()
    zs.receive_message()

```

CLIENT

Microservices benefits

The idea with microservices is break the monolithic architecture and separate each relevant functionality in a specific component.

- Microservices help software teams stay agile, and improve incrementally.
- In simpler terms, as the **services are decoupled from each other**, it is very easy to upgrade and improve a service without causing the other to go down
- Microservices can be much better leveraged via containers, which provide effective and complete virtual operating system environments, processes with isolation, and dedicated access to underlying hardware resources.
- **In a microservices pattern the data is spread over multiple services**

Microservices are an architectural style in which multiple, independent processes communicate with each other. These processes are designed to be highly scalable, de-coupled and perform one small task at a time. These multiple services have their own resources and processes which communicate among themselves through the network.

This differs from a typical client-server architecture where the backend is one monolithic structure encapsulating all the server logic. A microservices architecture addresses separation of concern. This design paradigm allows easier maintenance, greater flexibility, scalability and fault tolerance.

Serverless

Cloud computing is a very elegant paradigm, providing high-level tools for distributed systems. In particular, one can easily provision and customize virtual machines according to their computing needs. However, is there a way to execute code in the cloud without spending time and effort on system administration?

Serverless is the latest phase in the evolution of cloud development. Its building blocks are functions, a bunch of stateless “nano-services”, that can scale automatically and charged only when used. It enables teams to focus more on development while having fully managed servers.

Serverless computing allows us to focus on building the application without managing infrastructure and scaling. Serverless doesn't mean no servers at all. It is still there, but we are not going to manage it. The best part is that you only pay for the amount of resources each request consumed. Each request to the serverless function is allocated and the application code runs in a stateless container.

"Serverless" denotes a special kind of software architecture in which application logic is executed in an environment without visible processes, operating systems, servers or virtual machines.

With serverless deployments, the web application only exists during the span of a single HTTP request. The benefit of this is that there's no configuration required, no server maintenance, no need for load balancers, and no cost of keeping a server online 24/7.

Obviously servers are still needed, but the term “serverless” is used because the server management and capacity planning decisions are completely hidden from the developer or operator. Serverless code can be used in conjunction with code deployed in more traditional styles, such as microservices. However, applications can be written to be purely serverless as well.

Serverless architecture

- **Function as a Service.** Serverless functions are well suited to operations that take a very short time to complete and no resource-intensive. Providers impose concurrent execution, execution time and memory limits with mean that if the workload is long-running or intensive, a conventional server model may be more suitable.
- **Scalability.** Dynamic resource utilization.
- **Deploy your code.** Just package and upload the code.
- **Pay for actual usage.** You only pay for the time your code is running in the server cloud provider.
- **Serverless computing allows developers focus on building the application without managing infrastructure.**

Serverless uses cases

The serverless functions are easy to use when it is not required to save memory status. Because there is no control over when the execution environments are created or destroyed, it can not be assumed that when saving a data in the memory of the function, it remains there when the function is invoked again.

The following are some **uses cases** of when you are using serverless technologies:

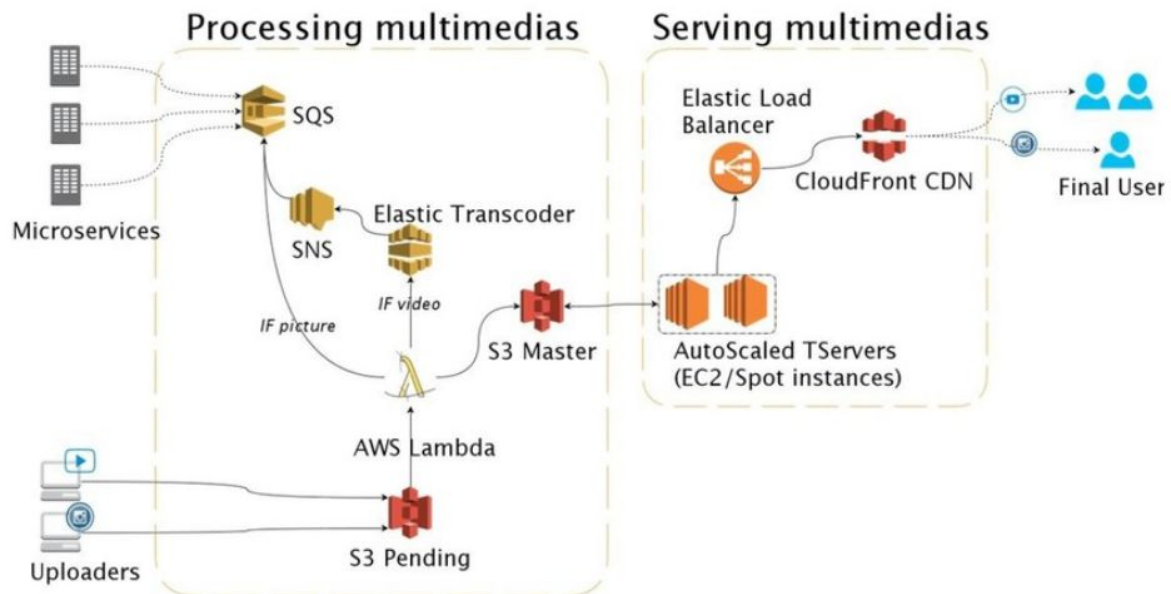
- For very variable traffic levels, for example millions of browsers requesting update.
- **Execute scheduled tasks:** execute actions periodically or at a specific time. Serverless functions are well suited to operations that take a very short time to complete and no resource-intensive.
- **Events.** Events are a key concept for serverless code. All serverless functions are triggered by an event. The serverless.com framework also includes an event gateway that enables a function to react to any event on any cloud.
- **Cloud Providers impose execution time and memory limits** with mean that if the workload is a long-running process, a conventional server model may be more appropriate.

Processing and serving multimedias with aws architecture

The most basic example, popularized by AWS Lambda is an **image processing event handler function**. The function is connected to a data store, such as Amazon S3, that emits change events. Each time a new image file is uploaded to a folder in S3, an event is generated, and forwarded to the event handler function that generates a thumbnail image that is stored in another folder.

This type of architecture executes the code that we have created based on different events. It is not necessary to have a machine with our code and an application server that executes it, **simply when an event occurs that we define, the code that we have configured will be executed.**

Each time a client uploads a multimedia to S3 (event), the process is executed. If the multimedia element is a video, the lambda function is responsible for calling **Elastic Transcoder** <https://aws.amazon.com/elastictranscoder/> as multimedia transcoding service. .If we have an image, then we call the **Amazon SQS service**.



AWS Lambda allows you to add other events for our code such as S3 bucket events. On your platform, when your users upload a file, it is stored in an S3 bucket. The event is collected by a code that creates the thumbnails and is responsible for updating the data in its database.

Serverless Advantages:

- You no longer have to worry about keeping the servers where your code runs.
- Goodbye to install software, open or close ports, run updates, ssh, etc.
- It is horizontally scalable, you do not have to worry about clusters, load balancing
- You pay only for the running time, contrary to paying a computer even for the time you are hibernating.
- Horizontally scalable
- Pay for what you use
- Infrastructure managed by service provider
- The functions can be easily integrated with other services from the same provider, which makes it easier to create HTTPS endpoints, logging, monitoring, what is necessary to have a service ready for production.

Serverless Disadvantages:

- If it is not developed with care, your code may end up quite coupled to the provider.
- Being such a recent service, the languages that can be used to implement the functions are limited by what is supported by the provider.
- Deploying and monitoring the behavior of multiple functions is much more complicated than monitoring a monolith or microservice.
- It takes extra effort to develop locally without the need to deploy the code to the execution environments whenever a change is made, as it can be time consuming and tedious.
- The languages that can be used to implement the functions are limited by what is supported by the provider.
- **The tools around the deployment automation of serverless functions are still very immature.**
- **Debugging is a little tricky, most tools include visualize logs in different develop environment, for example in amazon we have cloudwatch logs for getting metrics, monitoring and debugging**

Cloud providers

The most popular is Amazon who was the first to offer its **AWS Lambdas service**. However, other providers are already offering very similar solutions. The open source options are:

- **Kubeless** <https://kubeless.io/>, a serverless framework for Kubernetes.
- **OpenWhisk** <http://openwhisk.apache.org/> by Apache includes native support for Node.js, Python, Java, and Swift, and supports other languages and runtime via Docker containers.

These frameworks are designed to be deployed over Kubernetes clusters, Swarm or Docker. And they offer a Serverless open source environment where we can execute our portions of code, either in a public cloud or in a private one. They support many languages: Java, Python, Node.js, Ruby, PHP.

Aws lambda

Lambda <https://aws.amazon.com/lambda> is a service you can use to trigger the execution of a Lambda Function. A Lambda Function is a piece of code that you can write in Node.js, Java, C#, or Python 2.7 or 3.6, and that is deployed as a deployment package, which is a ZIP file containing your script and all its dependencies. **If you use Python, the ZIP file is usually a Virtualenv with all the dependencies needed to run the function.**

Lambda functions can replace **Celery workers**, since they can be **triggered asynchronously via some AWS events**. The benefit of running a Lambda function is that you do not have to deploy a Celery microservice that needs to run 24/7 to pick messages from a queue. Depending on the message frequency, using Lambda can reduce costs. However, again, using Lambda means you are locked in AWS services.

In AWS, **Lambda is the function as a service offering**. With this service, you can upload your Python, Node.js, Java or C# code and Lambda will store it and run it for you. To work with the Lambda service you upload your project packaged as a zip file, containing your own code plus any dependencies that are needed. **You have to designate a function in your code as the entry point, and this function will be called by AWS when a client "invokes" your Lambda function.**

There are many types of events that can trigger a Lambda function to execute, and this is one of the nicer aspects of this service. For example, you can configure your lambda to run on a schedule (cloud based cron jobs!), or when someone drops a new file on S3, which is the file storage service in AWS. Of course you can also trigger the function explicitly by using the Lambda service APIs, which makes it comparable to Celery in the sense that you can start an asynchronous task.

Aws lambda events

Events are a key concept for serverless code. All serverless functions are triggered by an event. Framework also includes an event gateway that enables a function to react to any event on any cloud.

Synchronous implies that we have a REST API and through the API gateway we invoke lambda functions behind the API with the classical request-->response

Asynchronous implies that there is a lambda function that is listening to events, for example when a file is uploading in s3 bucket or when we get a notification in Amazon SNS, then is called the lambda function, but the application doesn't know about when the event is produced.

Aws lambda functions

Lambda functions are equivalent to **Celery workers**. The idea with **lambda functions** is that they can be triggered asynchronously via some **AWS events**.

Lambda functions can replace **Celery workers**, since they can be **triggered asynchronously via some AWS events**. The benefit of running a Lambda function is that you do not have to deploy a Celery microservice that needs to run 24/7 to pick messages from a queue. Depending on the message frequency, using Lambda can reduce costs.

A lambda function basically is a handler function that AWS Lambda can invoke when the service executes your code. In Python, a lambda function handler is just a def function like this:

- **event** > AWS Lambda uses this parameter to pass in event data to the handler.
- **context** > AWS Lambda uses this parameter to provide runtime information to your handler.
- The return value can be anything: from basic values – converted to string - up to dictionaries – converted to JSON string

Aws lambda functions



```
def lambda_handler(event, context):  
    """Entry point.  
  
    event: AWS Lambda uses this parameter to pass in  
           event data to the handler.  
  
    context: AWS Lambda uses this parameter to provide  
             runtime information to your handler.  
    """  
    return
```

Awscli

<https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-install.html>

We can use **awscli** to upload and test a Lambda function. Just package the file and any dependencies into a .zip file. Upload the .zip file using either the console or AWS CLI to create a Lambda function. You specify the function name in the Python code to be used as the handler when you create a Lambda function. In this example, the handler is **hello_python.my_handler** (filename.function-name).

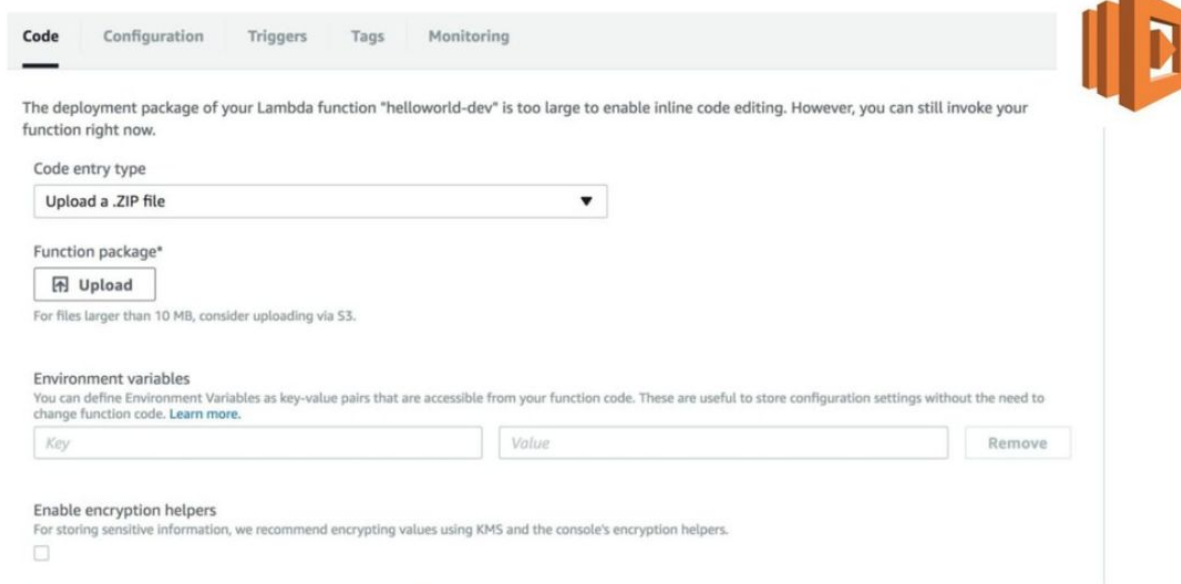
Create lambda function with awscli



```
$ aws lambda create-function \  
--region eu-west-1 \  
--function-name MyHandler \  
--zip-file fileb://handler.zip \  
--role arn:aws:iam::XXX:role/MyLambdaRole \  
--vpc-config SubnetIds=XXX,SecurityGroupIds=XXX \  
--handler handler.handler \  
--runtime python3.6 \  
--profile personal \  
--timeout 10 \  
--memory-size 512
```

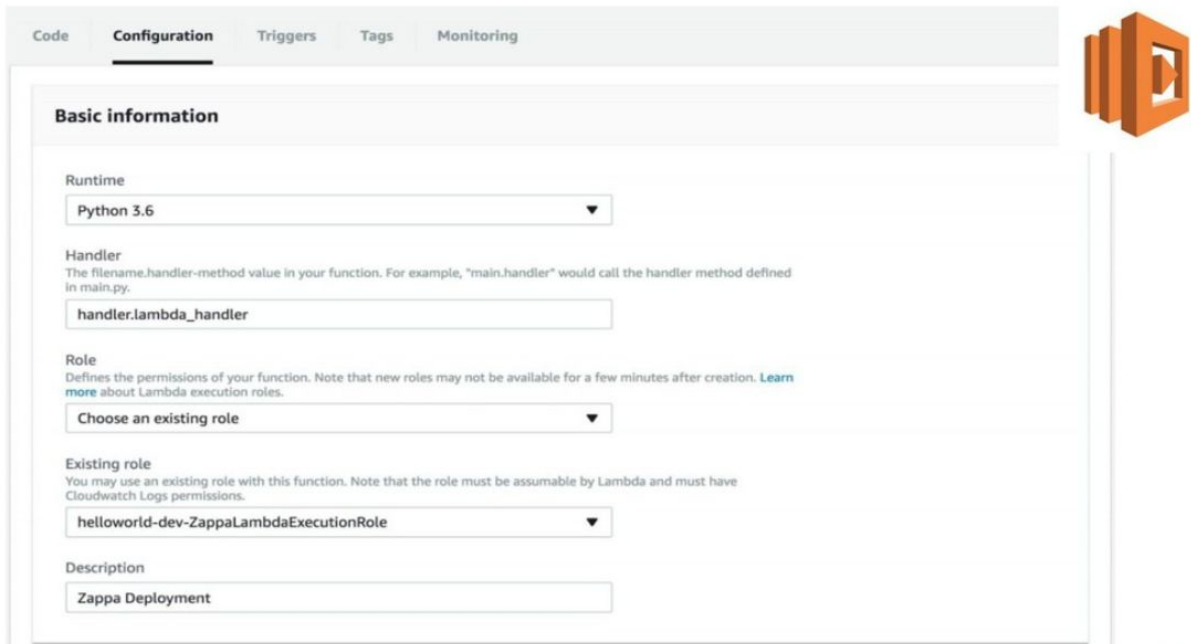
Create lambda function in aws account

Create your lambda function by supplying the zip file and some options

A screenshot of the AWS Lambda console interface. The 'Code' tab is selected, showing options to upload a ZIP file or S3 bucket. The 'Function package*' section has an 'Upload' button. The 'Environment variables' section shows a table with 'Key' and 'Value' columns. The 'Enable encryption helpers' section has a checkbox.

Code	Configuration	Triggers	Tags	Monitoring			
<p>The deployment package of your Lambda function "helloworld-dev" is too large to enable inline code editing. However, you can still invoke your function right now.</p> <p>Code entry type</p> <p>Upload a .ZIP file</p> <p>Function package*</p> <p>Upload</p> <p>For files larger than 10 MB, consider uploading via S3.</p> <p>Environment variables</p> <p>You can define Environment Variables as key-value pairs that are accessible from your function code. These are useful to store configuration settings without the need to change function code. Learn more.</p> <table><thead><tr><th>Key</th><th>Value</th><th>Remove</th></tr></thead><tbody></tbody></table> <p>Enable encryption helpers</p> <p>For storing sensitive information, we recommend encrypting values using KMS and the console's encryption helpers.</p> <p><input type="checkbox"/></p>					Key	Value	Remove
Key	Value	Remove					

The following create-function AWS CLI command creates a Lambda function. Among other parameters, it **specifies the handler parameter to specify the handler name. The runtime parameter specifies python version**



The screenshot shows the AWS Lambda console's 'Configuration' tab for a function. The 'Basic information' section is expanded, showing the following settings:

- Runtime:** A dropdown menu set to 'Python 3.6'.
- Handler:** A text input field containing 'handler.lambda_handler'.
- Role:** A dropdown menu set to 'Choose an existing role'.
- Existing role:** A dropdown menu set to 'helloworld-dev-ZappaLambdaExecutionRole'.
- Description:** A text input field containing 'Zappa Deployment'.

On the right side of the console, the AWS logo is visible.

Serverless frameworks

For simplifying the deployment process of lambda functions, there are some frameworks and tools released for developing and managing your Python serverless applications.

I will focus on developing and **managing your serverless applications with chalice and zappa solutions.**

Frameworks 



 serverless

Lambdify

Programmable AWS Lambda for Python

[View the Project on GitHub](#)
ZhukovAlexander/lambdify

Download
ZIP File

Download
TAR Ball

View On
GitHub

python- λ Chalice

Serverless Framework <https://serverless.com>. General purpose with multiple languages support. Examples in many languages in <https://github.com/serverless/examples>.

Chalice <https://github.com/aws/chalice>. Python serverless microframework from Amazon for AWS lambda.

Zappa <https://github.com/Miserlou/Zappa>. Serverless Python WSGI with AWS Lambda + API Gateway.

Python-λ <https://github.com/nficano/python-lambda>. A toolkit for developing and deploying serverless Python code in AWS Lambda.

Lambdify <http://zhukovalexander.github.io/lambdify/> a tool that turns any python callable into an AWS Lambda function.

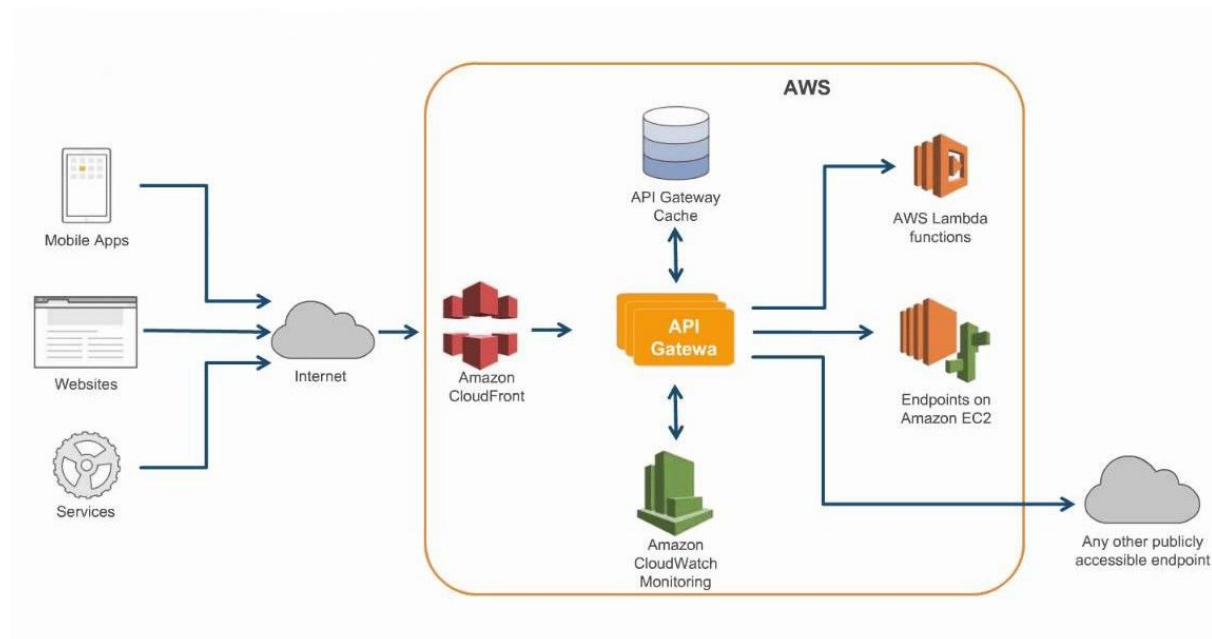
	Serverless.com	Zappa	AWS Chalice
Strengths/ Key features	Cross-cloud serverless framework General purpose- Web, REST API, IoT	Serverless Python WSGI with AWS Lambda + API Gateway. Eg Django and Flask apps	Simplify Web and REST API using Flask like URL routes. Focused on Python only at the moment Run locally via chalice local
Key observations	Wealth of extensions and community Need to get familiar with a the frameworks configuration (DSL)	Significant abstraction of AWS API Gateway and of AWS lambda Long time to deploy, difficult to revert if there are errors	Not as elegant and mature as Flask and require particular folder structure

Important: Before you begin to use these framework make sure you have a valid AWS account and your AWS credentials file is properly installed and for working with zappa and chalice we need configure first the project's virtual environment with virtualenv.

Amazon API Gateway

This Amazon service <https://aws.amazon.com/api-gateway/> lets you define an API that is fully managed by AWS, without requiring any infrastructure. Amazon API Gateway offers a variety of ways to control access to APIs: API calls can be signed with AWS credentials, you can use OAuth tokens and simply forward the token headers for verification, you can use API keys , or make an API completely public.

This Amazon service **allows us create endpoints to create an API and based on these endpoints configure what code we have in AWS Lambda will execute each one of them.**



Zappa architecture

Zappa <https://github.com/Miserlou/Zappa> is a framework that makes that easily to deploy Python WSGI applications with AWS Lambda and AWS API Gateway. **Basically, zappa is a framework for python applications that use AWS for deployment.**

Features:

- Deploy python wsgi applications
- Great for micro and macro services
- AWS lambda + AWS api gateway
- AWS event sources

Zappa provides a command line for Lambda functions in Flask and Django applications. Technically speaking, it's a framework that packages and deploys WSGI-compatible Python applications to an AWS Lambda function.

Deploy your WSGI apps on AWS Lambda

With Zappa, each request is given its own virtual HTTP "server" by Amazon API Gateway. AWS handles the horizontal scaling automatically, so no requests ever time out. After your app returns, the "server" dies.

- No more tedious web server configuration!
- No more paying for 24/7 server uptime!
- No more worrying about load balancing / scalability!
- No more worrying about web server security!

```
$ cd demo
~/demo $ ls
my_app.py      zappa_settings.json
~/demo $ source env/bin/activate
(env)~/demo $ cat my_app.py
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello():
    return 'Hello, from Zappa!\n'

if __name__ == '__main__':
    app.run()
(env)~/demo $ cat zappa_settings.json
{
  "dev": {
    "s3_bucket": "lmbda",
    "app_function": "my_app.app",
    "parameter_depth": 1
  }
}
(env)~/demo $ zappa deploy dev
Packaging project as zip...
Uploading zip (5.8MiB)...
Creating API Gateway routes...
86% | 82/95 [00:04<00:00, 17.85it/s]
```

Zappa interactive wizard

Zappa provides an **interactive wizard** that helps you **set up Zappa projects** quickly when you're first getting started. The command line tool asks you a handful of simple questions with smart defaults (e.g. **it detects if you're working on a Flask or Django project**) and it creates the Zappa project for you.

```
Welcome to Zappa!

Zappa is a system for running server-less Python web applications on AWS Lambda and AWS API Gateway.
This 'init' command will help you create and configure your new Zappa deployment.
Let's get started!

Your Zappa configuration can support multiple production stages, like 'dev', 'staging', and 'production'.
What do you want to call this environment (default 'dev'):
```

AWS Lambda and API Gateway are only available in certain regions. Let's check to make sure you have a profile set up in one that will work.

We found the following profiles: default, adsk forge2?, and adsk forge. Which would you like us to use? (default 'default'):

Your Zappa deployments will need to be uploaded to a private S3 bucket.

If you don't have a bucket yet, we'll create one for you too.

What do you want call your bucket? (default 'zappa-68fz81bc0'):

It looks like this is a Flask application.

What's the modular path to your app's function?

This will likely be something like 'your_module.app'.

We discovered: app.app

Where is your app's function? (default 'app.app'):

```
{
  "dev": {
    "app_function": "app.app",
    "aws_region": "eu-west-1",
    "profile_name": "default",
    "s3_bucket": "zappa-68fz81bc0"
  }
}
```

Does this look okay? (default 'y') [y/n]:

Done! Now you can deploy your Zappa application by executing:

```
$ zappa deploy dev
```

After that, you can update your application code with:

```
$ zappa update dev
```

To learn more, check out our project page on GitHub here: <https://github.com/Miserlou/Zappa> and stop by our Slack channel here: <https://slack.zappa.io>

Enjoy!,
~ Team Zappa!

Zappa settings.json

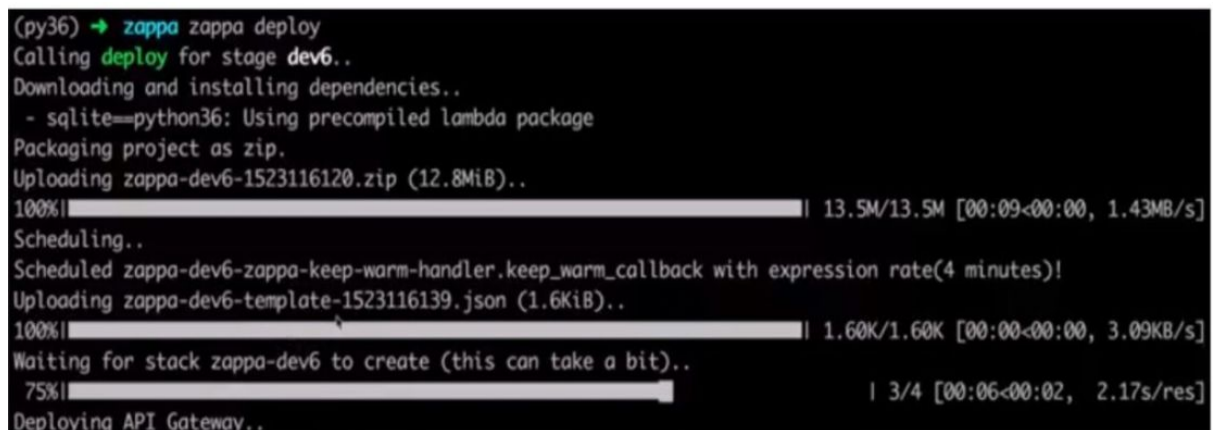
Once you create your project, you'll manage the configuration directly in the Zappa settings file. This is a simple YAML or **JSON** file that **stores your project configuration** for each stage. **it returns information about region execution in amazon, runtime execution, application settings**

zappa_settings.json

```
{
  "dev": {
    "aws_region": "us-east-1",
    "django_settings": "hello.settings",
    "profile_name": "default",
    "project_name": "hello",
    "runtime": "python3.6",
    "s3_bucket": "zappa-huyg6op0s"
  }
}
```

Zappa deploy

Zappa has the ability to automagically package and deploy Flask or Django apps to an AWS serverless app. With a single command, you can deploy, update or destroy an application for a specified stage (e.g. dev, staging, production).

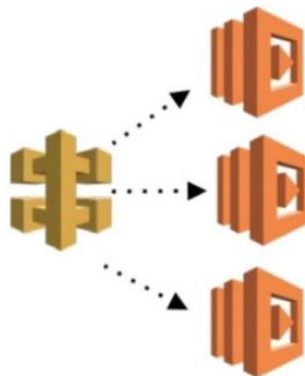


```
(py36) → zappa zappa deploy
Calling deploy for stage dev6..
Downloading and installing dependencies..
- sqlite==python36: Using precompiled lambda package
Packaging project as zip.
Uploading zappa-dev6-1523116120.zip (12.8MiB)..
100%|████████████████████████████████████████| 13.5M/13.5M [00:09<00:00, 1.43MB/s]
Scheduling..
Scheduled zappa-dev6-zappa-keep-warm-handler.keep_warm_callback with expression rate(4 minutes)!
Uploading zappa-dev6-template-1523116139.json (1.6KiB)..
100%|████████████████████████████████████████| 1.60K/1.60K [00:00<00:00, 3.09KB/s]
Waiting for stack zappa-dev6 to create (this can take a bit)..
75%|██████████████████████████████████████| 3/4 [00:06<00:02, 2.17s/res]
Deploying API Gateway..
```

Chalice

Chalice <https://chalice.readthedocs.io/en/latest> allows you to quickly create and deploy applications that use Amazon API Gateway and AWS Lambda. Chalice is a microframework for managing the lambda functions and the API Gateway. It uses the Amazon API Gateway, which must be configured to route HTTP to your Lambda function.

- Python Serverless Microframework for AWS
- Each endpoint is a separate function



It provides:

- **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 IAM policy generation**

Chalice handles all configuration process for you in an easy way and from the developer point of view just have to focus on the python specific logic of your application. The '**route decorator**' automatically builds a routing table for the lambda function to invoke functions based on the request path and the method(get,post..).

- chalice enables single-page HTTP handling, handling request routing behind the scenes
- Using Python decorators you can define an HTTP route directly on top of the handler function.

```
$ pip install chalice
$ chalice new-project helloworld && cd helloworld

$ cat app.py
from chalice import Chalice

app = Chalice(app_name="helloworld")

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

$ chalice deploy
```

Chalice deploy

Chalice provides some nice features, such as **automatically generating an IAM security policy** based on an inspection of your code.

Using the chalice logs and chalice deploy commands together, you can iterate quickly over test-diagnose-fix-deploy cycles in a live environment. **The chalice deploy command can optionally take a deployment stage name, and you can deploy different versions of your code to different stages.** Using this feature, you can leave your production stage intact while modifying your development stage. Then you can deploy to the production stage when the changes are ready to go out.

Testing AWS Lambda

We can find some Docker images and test runners that replicate the live AWS Lambda environment. **We have a collection of containers for testing aws lambda environment.**

Docker-lambda <https://github.com/lambci/docker-lambda> is a sandboxed local environment that replicates the live [AWS Lambda](#) environment almost identically – including installed software and libraries, file structure and permissions, environment variables, context objects and Behaviors

docker-lambda

A sandboxed local environment that replicates the live [AWS Lambda](#) environment almost identically – including installed software and libraries, file structure and permissions, environment variables, context objects and behaviors – even the user and running process are the same.

```
~/ docker run -v "$PWD":/var/task lambci/lambda
START RequestId: 4ce0a79f-5a50-1c09-1a8a-c0c4c6431a49 Version: $LATEST
2016-05-26T03:47:37.994Z      4ce0a79f-5a50-1c09-1a8a-c0c4c6431a49    process.execPath:
2016-05-26T03:47:37.995Z      4ce0a79f-5a50-1c09-1a8a-c0c4c6431a49    /usr/local/lib64/node-v4.3.x/bin/node
2016-05-26T03:47:37.995Z      4ce0a79f-5a50-1c09-1a8a-c0c4c6431a49    process.cwd():
2016-05-26T03:47:37.995Z      4ce0a79f-5a50-1c09-1a8a-c0c4c6431a49    /var/task
2016-05-26T03:47:37.996Z      4ce0a79f-5a50-1c09-1a8a-c0c4c6431a49    child_process.execSync('ls -la /tmp'):
2016-05-26T03:47:38.010Z      4ce0a79f-5a50-1c09-1a8a-c0c4c6431a49    total 8
drwxr-xr-x  2 sbx_user1051  495 4096 May 26 02:14 .
drwxr-xr-x 27 root         root 4096 May 26 03:47 ..
2016-05-26T03:47:38.011Z      4ce0a79f-5a50-1c09-1a8a-c0c4c6431a49    context.getRemainingTimeInMillis():
2016-05-26T03:47:38.011Z      4ce0a79f-5a50-1c09-1a8a-c0c4c6431a49    299978
END RequestId: 4ce0a79f-5a50-1c09-1a8a-c0c4c6431a49
REPORT RequestId: 4ce0a79f-5a50-1c09-1a8a-c0c4c6431a49  Duration: 24.40 ms    Billed Duration: 100 ms Memory
Size: 1536 MB    Max Memory Used: 24 MB
null~
```

Conclusions

Serverless or FaaS is a new tool that allows you to deploy functionality ready for production without investing effort in infrastructure and pay only when it is executed.

It is different from IaaS (infrastructure as a service) where you pay for empty servers and are responsible for configuring, managing and maintaining them, and it is also different from PaaS (platform as a service) where you do not manage the servers but assume the cost to have your code running like this is not being used.

Serverless is a technology that has a lot of potential, especially to deploy small or isolated functionality or for those people who do not have infrastructure knowledge but want to have applications in production.

References:

- <https://aws.amazon.com/blogs/developer/preview-the-python-serverless-microframework-for-aws>
- <https://cloudacademy.com/blog/serverless-framework-aws-lambda-api-gateway-python/>
- <https://serverless.com/framework/docs/providers/aws/examples/hello-world/python/>
- <https://github.com/serverless/examples/tree/master/aws-python-simple-http-endpoint>
- <https://randomant.net/building-serverless-apps-with-aws-lambda-python-and-zappa/>