

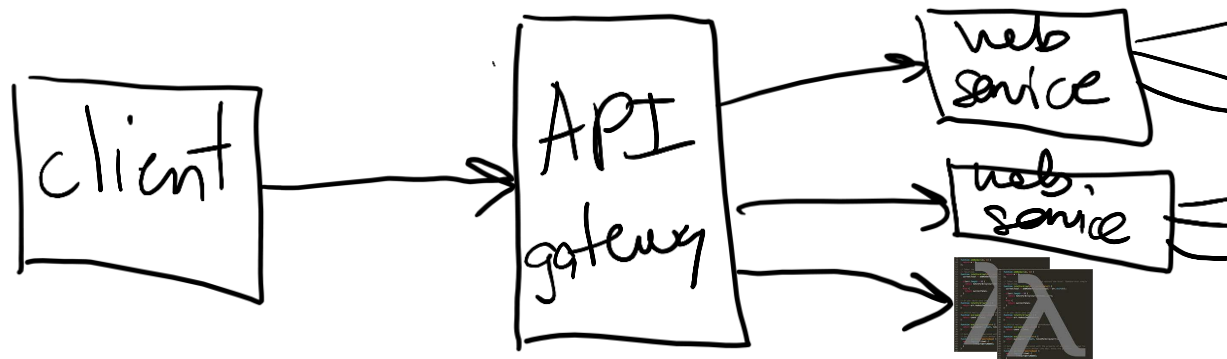
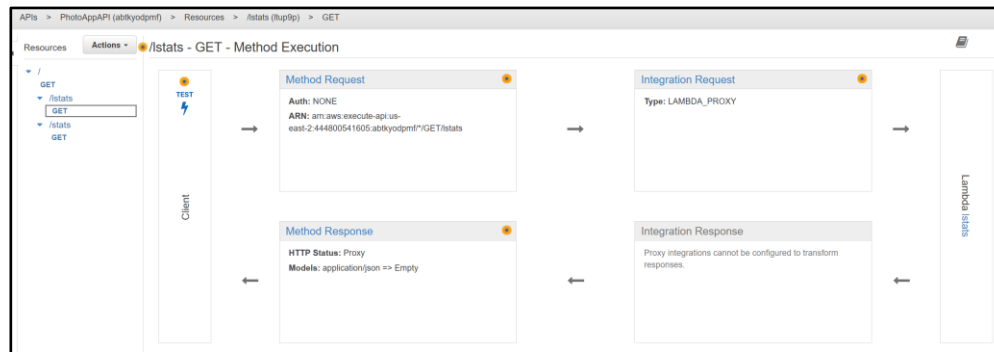
# API Gateway

- **Programming example**
  - **API Gateway + lambda**
- **Other ways to call lambda functions**



# API Gateway

- **API Gateway** allows you to define a RESTful API that forwards to other services / lambdas
  - Define HTTP verb and URL path (e.g. GET /movies)
  - Specify target...



# Programming demo

- **Let's build a simple calculator using API Gateway and lambda...**

# (1) lambda functions

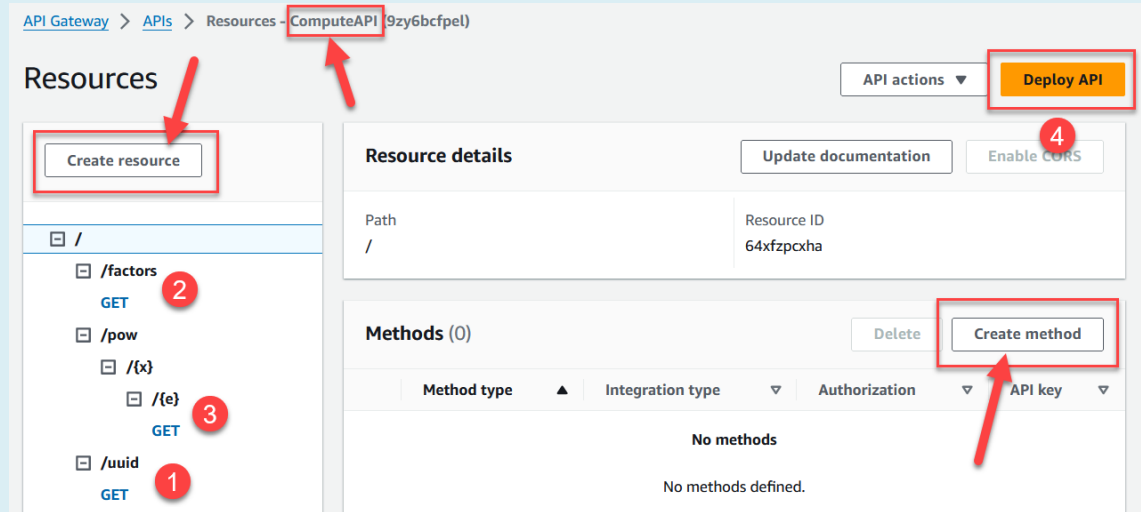
```
#  
# uuid()  
#  
import json  
import uuid  
  
def lambda_handler(event, context):  
    result = str(uuid.uuid4())  
  
    print("uuid:", result)  
  
    return {  
        'statusCode': 200,  
        'body': json.dumps(result)  
    }
```

```
#  
# pow(x, e)  
#  
import json  
  
def lambda_handler(event, context):  
    params = event["pathParameters"]  
    x = float(params["x"])  
    e = float(params["e"])  
  
    result = x ** e  
  
    print("pow:", x, e, result)  
  
    return {  
        'statusCode': 200,  
        'body': json.dumps(result)  
    }
```

```
#  
# factors(n)  
#  
import json  
  
def lambda_handler(event, context):  
    params = event["queryStringParameters"]  
    n = int(params["n"])  
  
    i = 2  
    factors = []  
    while i * i <= n:  
        if n % i:  
            i += 1  
        else:  
            n //= i  
            factors.append(i)  
  
    if n > 1:  
        factors.append(n)  
  
    print("factors:", n, factors)  
  
    return {  
        'statusCode': 200,  
        'body': json.dumps(factors)  
    }
```

## (2) Build API

- In AWS, search for API Gateway service



- **Create API**

- *Select: **REST API***
- *Name: **ComputeAPI***
- *Create resources and method as shown (steps 1-3)*
- *When you create methods:*
  - Lambda function
  - Enable Lambda proxy integration
  - Select function

**When you're done, click "Deploy" and stage if you want (e.g. test)**

## (3) Test API

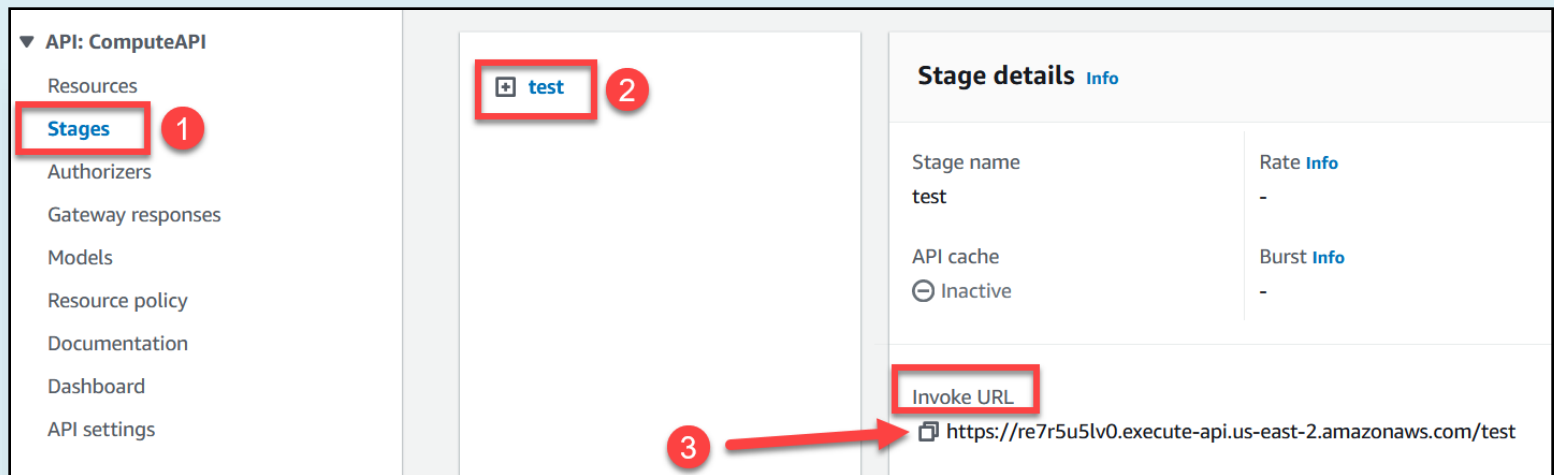
- Use the Test tab to test each function...

1. *uuid( ) has no parameters*
2. *factors(n) has a query parameter, e.g. ?n=33*
3. *pow(x, e) has path parameters, e.g. /pow/12/2*

The screenshot shows the 'Test' tab of an API Gateway interface. On the left, a sidebar lists API endpoints: `/`, `/factors` (GET), `/pow`, `/f(x)`, `/f(e)` (GET), and `/uuid` (GET). A red circle with the number '1' is next to the `/f(e)` endpoint. The main panel has tabs for 'Method request', 'Integration request', 'Integration response', 'Method response', and 'Test'. The 'Test' tab is active, indicated by a red circle with the number '2'. Below the tabs, the 'Test method' section contains fields for 'Path', 'Query strings', 'Headers', and 'Client certificate'. The 'Path' field has a value of `e` and a sub-field with the value `2`, with a red circle '4' next to it. The 'x' field has a value of `12`, with a red circle '3' next to it. The 'Query strings' field contains `param1=value1&param2=value2`. The 'Headers' field contains `header1:value1` and `header2:value2`. The 'Client certificate' field shows 'No client certificates have been generated.' with a dropdown arrow. At the bottom right, there is a 'Test' button with a red circle '5' next to it.

## (4) Deploy and copy API endpoint

- Click "Deploy API"
- Stages --- none, or e.g. "test" or "prod" (production)
- Click "Deploy"
- View stage, copy "Invoke URL"



## (5) client-side testing

- **Since all the methods are GET, use web browser**
  - *API Gateway Endpoint/uuid*
  - *API Gateway Endpoint/factors?n=33*
  - *API Gateway Endpoint/pow/2/16*
- **Alternatively:**
  - *postman.com*
  - *HTTP requests from client app*



# Python-based client-side app



```
import requests

baseurl = 'API Gateway Endpoint'

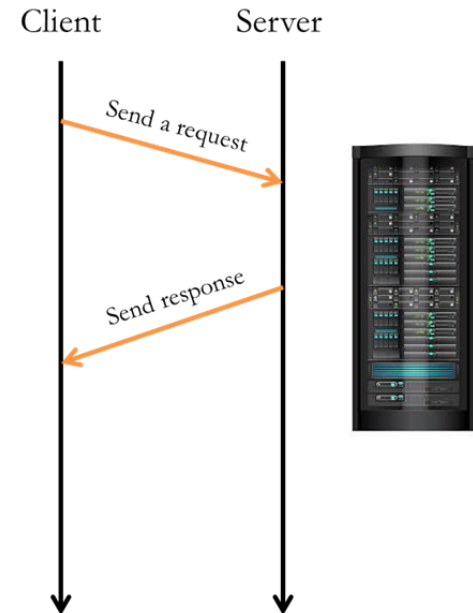
x = input('Enter base x> ')
e = input('Enter exponent e> ')

# build URL:
url = baseurl + '/pow/' + x + '/' + e

# call the web service:
response = requests.get(url)

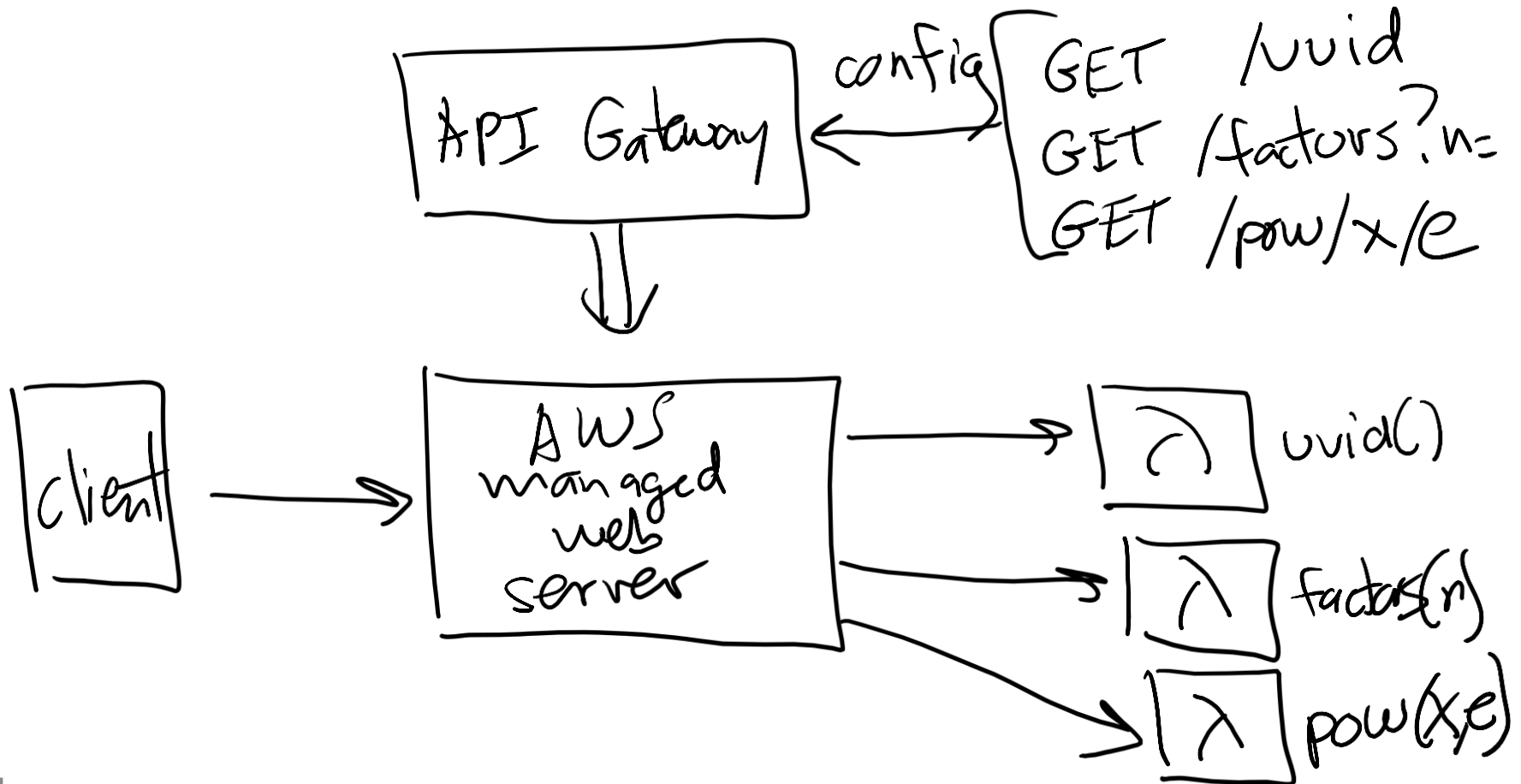
# output the result:
body = response.json()

print('status code:', response.status_code)
print('result x^e:', body)
```



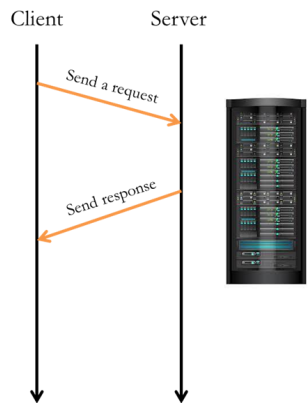
# Summary

- We have a serverless solution to running computations in the cloud



# Recall what we did earlier...

- We did a similar web service using JS and node.js



```
const express = require('express');
const app = express();

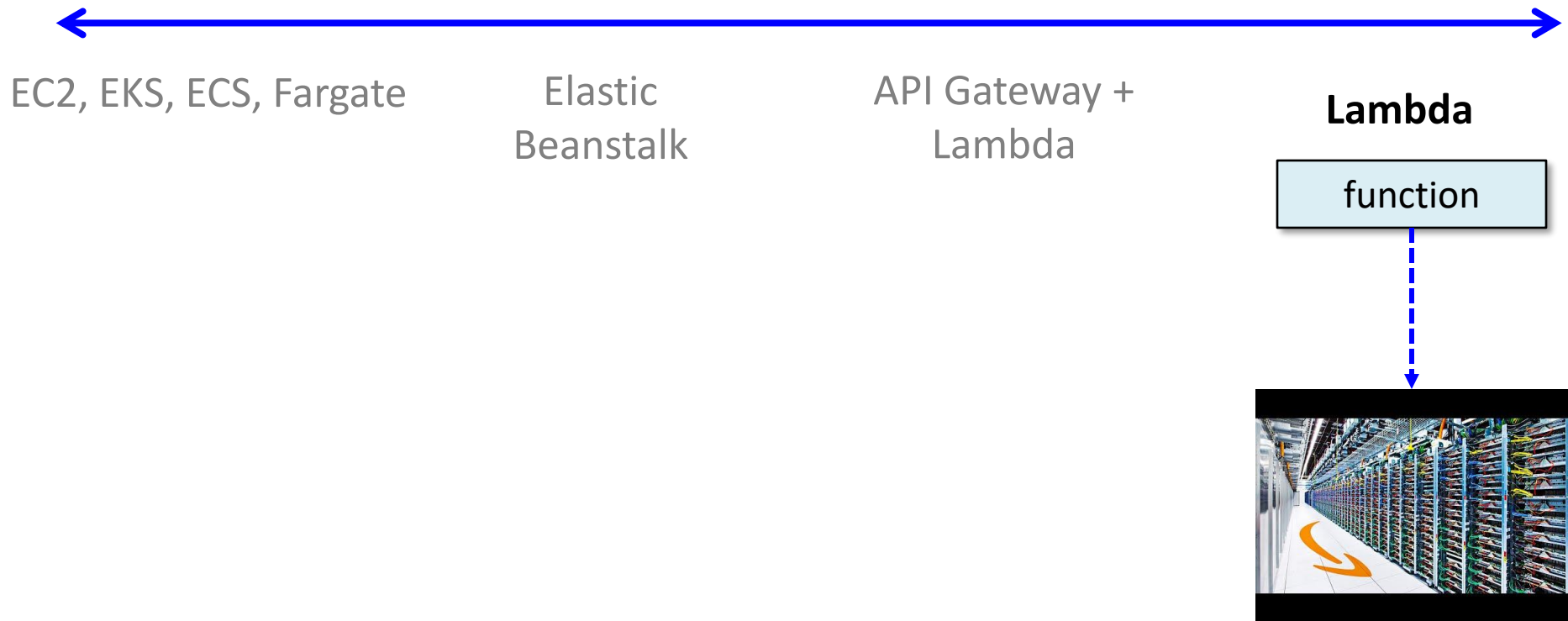
// main():
app.listen(3000, () => {
  console.log('**SERVER: web service running, listening on port 3000...');
});

// requests for default page /:
app.get('/', (req, res) => {
  console.log('**SERVER: call to /');
  res.send('<HTML><body>Home page is empty, we are a calculator service!</body></HTML>');
});

// API functions:
// raise x to the exponent e:
app.get('/pow/:x/:e', (req, res) => {...});
.
.
.
```

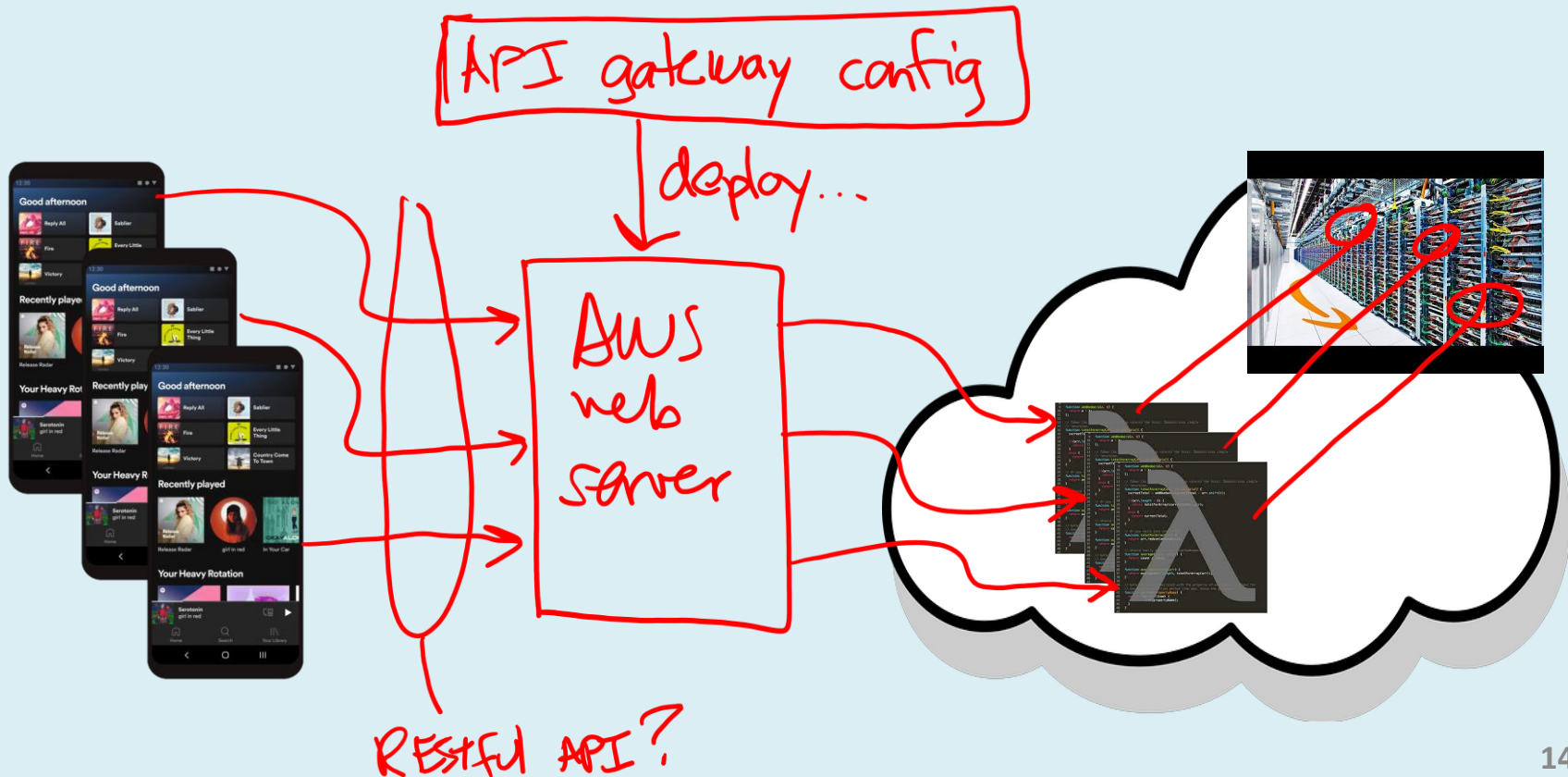
# AWS lambda

- By far the simplest, least expensive way to compute



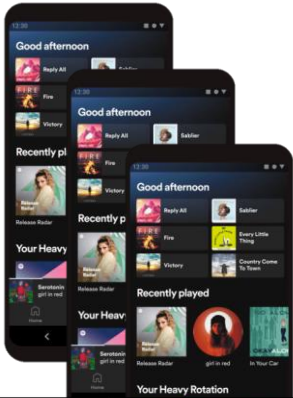
# Calling lambda functions

- We used API Gateway to "call" our lambda functions
  - *All the benefits of a traditional multi-tier design*
  - *Offers the most customization / config options*



# (1) Calling lambda via **function URL**

- Multi-tier through AWS-managed web server



սլ հղիւթ յ, ., յ, .,  
սêşrôղşê sêşûêşşşê gêtş սլ

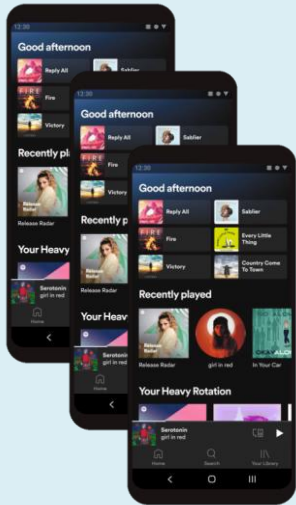


**Multi-tier design with the usual advantages. Good for simpler apps with a small API (not too many URLs)**

**Cannot map to non-lambda services to improve latency; limited configuration options (e.g. cannot support real-time WebSocket apps)**

## (2) Calling lambda directly from client

- You can use AWS libraries (e.g. boto3) to call lambda functions directly...



sêşulʽ lăncđă inhôlê  
GujctʽiônNănê găctʽôss  
Rây'lôăđ kşon đunrş ʽ ʽ

Great for small projects and prototypes...

Requires config / credentials on the client; standard installation concerns; less secure?

**That's it, thank you!**