Functions as a Service
(Serverless computing)
Motivation

- All require at least one server to be running at all times
- Want something that costs $0 if not used
Serverless computing

- A solution that costs nothing if nobody is using it
- Similar to PaaS
  - No up front provisioning
  - No management of servers
  - Pay for what you use
  - But, can go down to 0 servers and "wake-up" when needed
- Enables "event-driven" computing
  - Single-purpose function executed in response to some asynchronous event
  - Run on ephemeral run-time systems
  - Stateless
Functions as a Service

- Consists of 2 things
  - An event or trigger
  - A function to run when the event happens
  - e.g. “When an event happens, run this code”

- Treats servers and computation like electricity (i.e. a commodity consumed on-demand)
  - No machine, container, or VM to manage
  - Resources automatically scaled up based on function usage
  - Cheapest way to implement microservices with low usage

- Sometimes referred to as Internet glue or HTTP duct tape

- A functional programming approach to the cloud
  - No state stored in a function
  - Side-effects pushed out to the edge
  - Allows for greater composability
Use cases

- Recall single page application with pre-rendered pages
- Pre-render entire dynamic site as a single page and forward deploy to client or edge
  - Avoid server rendering
  - Enable search engine indexing
- Examples
  - Render an entire WordPress site
  - Render Angular, React sites
- Can be done as a cloud function
  - Render periodically to get latest changes
  - Render upon a change to content
Other use cases

- Transcode a video when uploaded by a user
- Perform a speech-to-text conversion when requested
  - Amazon Echo
- Update high-scores of an app/site when database changes
- Run fraud detection or send e-mail welcome upon new user signup
- Ingest sensor data upon new IoT device reading
- Run a function at a particular time (e.g. cron in the cloud)
- Run a Slack Bot function upon receiving a Slack Slash command (your lab)
Broader patterns

- Managed services often implemented as FaaS
  - Cloud Vision API, Cloud Natural Language Processing API, BigQuery
  - Statistically multiplex at function level versus container/VM level to drive down price
- "Extract, Transform, and Load" pattern (ETL)
  - IoT sensors
- Typically not used to implement entire app
  - Used as glue or for self-contained parts of app
Examples

- AWS Lambda (2014)
- Google Cloud Functions (2016)
- Microsoft Azure Functions (2016)
- Apache OpenWhisk
Serverless issues

- Response times not guaranteed
  - Recently executed functions cached for “hot” operation
  - Idle functions torn down to save resources
  - Cold start for idle functions ~600ms
  - Not good for real-time operations due to unpredictable performance
- Comparison
- Limited time budget
  - Often implemented on "pre-emptible" VMs
  - Maximum execution on AWS Lambda = 5 min
- Vendor lock-in
Serverless issues

- Security?
  - Typically, no persistent malware on them
  - But assumptions
    - Are the OS and libraries continually patched?
    - Are all resources destroyed when function ends?

- Assumptions often fail
  - Exploitable function exposing underlying run-time (which may have your API keys in them)
    - Azure Functions co-tenants (BSidesPDX 2017) allowing a single poorly-written function to own all the rest
  - Caching "hot" functions can allow one to steal credentials if broken
    - Rich Jones – “Gone in 60ms”
Google Cloud Functions
Google Cloud Functions

- Functions as a service running in a standardized, managed environment (mostly Node.js, some Python)
  - User supplies single file defining function and a file listing the packages it requires (e.g. `package.json`)
  - Runtime compiles function down to native modules via `npm` (e.g. Gentoo-like) for deployment
- Function can do one of two things
  - Implement a REST API that is brought up when an event hits its URL (synchronous)
  - Implement a background function that calls back to app when done (asynchronous)
Distributed messaging
Message Brokers

- Also known as publish-subscribe messaging systems
- Messaging in the cloud to sending and receive event notifications
  - Used to trigger functions or data processing pipelines
  - Must be interoperable across multiple languages and platforms to connect heterogeneous producer/consumers of data
- Must scale

- Others
  - RabbitMQ, Redis (in memory database with pub/sub)
Google Pub/Sub
Cloud Pub/Sub

- Many-to-many asynchronous messaging in GCP
  - > 1M messages per second
- Used to pipe data into App Engine, BigQuery, Dataflow
- Often used as triggers for Cloud Functions
  - IoT devices and sensors generating data
  - Push notifications for applications
Labs
Cloud Functions Lab #1

- Simple HTTP cloud function
- Enable Cloud Functions API in APIs & Services Dashboard

![Google Cloud Platform Interface](image-url)
Cloud Functions Lab #1

- Create the function
  - Create a folder on your local system called `gcf_http`.
  - Create a file called `index.js`, with the following contents

```
NODE.JS

/**
 * HTTP Cloud Function.
 *
 * @param {Object} req Cloud Function request context.
 * @param {Object} res Cloud Function response context.
 *
exports.helloGET = (req, res) => {
  res.send('Hello World!');
};
```

Cloud Functions Lab #1

- Deploy the application
  ```
gcloud functions deploy helloGET --trigger-http
  ```
- View the output to see the URL of your function
  - It will have the format
    ```
    https://[YOUR_REGION]-[YOUR_PROJECT_ID].cloudfunctions.net/helloGET
    ```

Deploying function (may take a while - up to 2 minutes)...done.
availableMemoryMb: 256
entryPoint: helloGET
httpsTrigger:
  url: https://us-central1-cs410c-wuchang-201515.cloudfunctions.net/helloGET
labels:
  deployment-tool: cli-gcloud
name: projects/cs410c-wuchang-201515/locations/us-central1/functions/helloGET
serviceAccountEmail: cs410c-wuchang-201515@appspot.gserviceaccount.com
Cloud Functions Lab #1

- Make an HTTP request to the function to trigger it via curl and web browser
  
  curl "https://[YOUR_REGION]-[YOUR_PROJECT_ID].cloudfunctions.net/helloGET"

- Delete the function
  
gcloud functions delete [NAME_OF_FUNCTION]
Cloud Functions Lab #1

- Simple HTTP cloud function (~10 min)
  - [https://cloud.google.com/functions/docs/tutorials/http](https://cloud.google.com/functions/docs/tutorials/http)
Cloud Functions Lab #2

- Blurring offensive images uploaded to storage bucket
- Clone the repository in Cloud Shell

```bash
git clone https://github.com/GoogleCloudPlatform/nodejs-docs-samples.git
cd nodejs-docs-samples/functions/imagemagick
```

- Create a Cloud Storage bucket for uploading images, with a globally unique bucket name:

```bash
gsutil mb gs://[YOUR_IMAGE_BUCKET_NAME]
```
Enable Vision API

Google Cloud Platform

Search

cloud vision

API Library

Cloud Vision API
Google
Image Content Analysis

ENABLE
TRY THIS API
View function code

- Include libraries

- Call Vision API with `filePath` of new object to do detection, then call `blurImage()` on file object if adult content or violence detected

```javascript
const storage = require('@google-cloud/storage')();
const vision = require('@google-cloud/vision').v1p1beta1;
const client = new vision.ImageAnnotatorClient();

exports.blurOffensiveImages = (event) => {
    const object = event.data;

    const file = storage.bucket(object.bucket).file(object.name);
    const filePath = `gs://${object.bucket}/${object.name}`;

    console.log(`Analyzing ${filePath}.`);

    return client.safeSearchDetection(filePath)
        .catch((err) => {
            console.error(`Failed to analyze ${filePath}.`, err);
            return Promise.reject(err);
        })
        .then(([result]) => {
            const detections = result.safeSearchAnnotation;

            if (detections.adult === 'VERY LIKELY' ||
                detections.violence === 'VERY LIKELY') {
                console.log(`The image ${filePath} has been detected as inappropriate.`);
                return blurImage(file);
            }
            console.log(`The image ${filePath} has been detected as OK.`);
        });
};
```
• blurImage()
  • Download image to a temporary file
  • Call ImageMagick's `convert` utility to blur image wrapped in a promise for error handling
• **blurImage() continued**
  • Upload back to bucket
  • Remove temporary file (good practice)

```javascript
83 .then(() => {
  console.log(`Image ${file.name} has been blurred.`);
84
  // Upload the Blurred image back into the bucket.
85  return file.bucket.upload(tempLocalFilename, { destination: file.name })
86   .catch((err) => {
87     console.error('Failed to upload blurred image.', err);
88     return Promise.reject(err);
89   });
90
91 });
92 .then(() => {
  console.log(`Blurred image has been uploaded to ${file.name}.`);
93
  // Delete the temporary file.
94  return new Promise((resolve, reject) => {
95    fs.unlink(tempLocalFilename, (err) => {
96      if (err) {
97        reject(err);
98      } else {
99        resolve();
100      }
101    });
102  });
103
104 });
105 
106 }
```
Deploy

- Register function and set trigger for its execution on storage bucket event.

```
gcloud functions deploy blurOffensiveImages --trigger-bucket [YOUR_IMAGE_BUCKET_NAME]
```
• Find an offensive image
  • e.g. a flesh-eating zombie at
    https://cdn.pixabay.com/photo/2015/09/21/14/24/zombie-949916_1280.jpg
  • Use `wget` to pull into Cloud Shell
• Upload image to bucket via console or command-line
  \texttt{gsutil cp zombie*.jpg gs://[YOUR_IMAGE_BUCKET_NAME]}
  • Function should automatically execute
• Then, upload two other images to the bucket
• View the images in the Cloud Storage bucket you created earlier for uploading images.
• Output the logs showing function execution showing at least one image that has been blurred
  \texttt{gcloud functions logs read}
Cloud Functions Lab #2

- Clean-up
  - Delete the function

```
gcloud functions delete [NAME_OF_FUNCTION]
```

- Link
  - https://cloud.google.com/functions/docs/tutorials/imagemagick (~20 min)
Cloud Functions Lab #3

- Create a Slack app that queries Google's Knowledge Graph API on demand via Cloud Functions
Application flow

1. User executes the `/kg <search_query>` Slash Command
2. Slack app sends the command payload to the Cloud Function's trigger endpoint along with its verification "token"
3. Cloud Function verifies token, then sends a request with the user's search query to the Knowledge Graph API along with an API key
4. Knowledge Graph API performs query and returns a matching result
5. Cloud Function formats the response for Slack
6. Sends it back.
7. The user sees the formatted response in the Slack channel.
Function code

- Interface definition (Javascript)

```javascript
/**
 * Receive a Slash Command request from Slack.
 *
 * Trigger this function by making a POST request with a payload to:
 * https://[YOUR_REGION].[YOUR_PROJECT_ID].cloudfunctions.net/kgsearch
 *
 * @example
 * curl -X POST "https://us-central11.your-project-id.cloudfunctions.net/kgSearch" \
 *   --data '{"token":'[YOUR_SLACK_TOKEN'],"text":"giraffe"}'
 *
 * @param {object} req Cloud Function request object.
 * @param {object} req.body The request payload.
 * @param {string} req.body.token Slack's verification token.
 * @param {string} req.body.text The user's search query.
 * @param {object} res Cloud Function response object.
 */
```
exports.kgSearch = (req, res) => {
    return Promise.resolve()
    .then(() => {
        if (req.method !== 'POST') {
            const error = new Error('Only POST requests are accepted');
            error.code = 405;
            throw error;
        }
    })
    .then((response) => {
        // Send the formatted message back to Slack
        res.json(response);
    })
    .catch((err) => {
        console.error(err);
        res.status(err.code || 500).send(err);
        return Promise.reject(err);
    });
};

// Verify that this request came from Slack
verifyWebhook(req.body);

// Make the request to the Knowledge Graph Search API
return makeSearchRequest(req.body.text);
• Slack app authenticates to Cloud Function via a shared token
  • Generated by Slack app, then included in function
  • Must be replaced with your own

```javascript
/** *
 * Verify that the webhook request came from Slack.
 *
 * @param {object} body The body of the request.
 * @param {string} body.token The Slack token to be verified.
 */

function verifyWebhook (body) {
  if (!body || body.token !== config.SLACK_TOKEN) {
    const error = new Error('Invalid credentials');
    error.code = 401;
    throw error;
  }
}
```
Call API (kgsearch)

- Cloud function authenticates to Knowledge Graph API via key
- Must be replaced with your own

Format a response to Slack based on response from API
Enable Knowledge Graph API

Dashboard

APIs & Services

Google Cloud Platform

Search

knowledge graph

Knowledge Graph Search API

Searches the Google Knowledge Graph for entities.

API Library

Knowledge Graph Search API

Google

Searches the Google Knowledge Graph for entities.

ENABLE

TRY THIS API
Create Knowledge Graph API Key

- In console, APIs & services => Credentials
- Create credentials and then select API key.
- Keep tab with API key open so you can copy to function
Create a Slack workspace

- Or use one you own

https://slack.com/create
Create a Slack app

- [https://api.slack.com/apps](https://api.slack.com/apps)
- Used to host your Slack Slash command
  - Associate it to workspace
Obtain Slack app's verification token

- Shared secret that authenticates Slack app to your Cloud Function
  - Automatically sent using the "token" field in HTTP cookie
  - In Basic Information of app
Set up Cloud Function

- In Cloud Shell, clone repository
  
  `git clone https://github.com/GoogleCloudPlatform/nodejs-docs-samples`
  
  `cd nodejs-docs-samples/functions/slack`

- Edit `index.js`
  
  - Comment out line 19 (require no longer works)
    
    ```javascript
    const config = require('./config.json');
    ```

  - Replace `config.BLLACK_TOKEN` in line 90 with verification token provided by Slack in the Basic information page of your app config (in double-quotes)
    
    ```javascript
    function verifyWebhook (body) {
      if (!body || body.token !== config.BLLACK_TOKEN)
    ```

    Verification Token
    ```
    cOKp1NXVqInw9R1uPxn
    ```
• Replace `config.KG_API_KEY` with API key you just created (in double-quotes)

```javascript
kgsearch.entities.search({
  auth: config.KG_API_KEY,
  query: query,
  limit: 1
})
```
Deploying the Function

- **Via**
  
  `gcloud functions deploy kgSearch --trigger-http`

- **Note the URL of function**
  
  ```
  entryPoint: kgSearch
  httpsTrigger:
  url: https://us-central1-cs410c-wuchang-201515.cloudfunctions.net/kgSearch
  ```
Create Slack command

- Go to Slash commands and click the Create new command button.

- Configure command
  - `/kg` as the name
  - URL listed for function in previous step as Request URL
    `https://[YOUR_REGION]-[YOUR_PROJECT_ID].cloudfunctions.net/kgSearch`
  - Then, save command
• Install the App into the workspace

• Authorize app
Using the Slash Command

• Test the command manually:

```
curl "https://[YOUR_REGION]-[YOUR_PROJECT_ID].cloudfunctions.net/kgSearch"
  -H "Content-Type: application/json" --data
  '{"token": "[YOUR_SLACK_TOKEN]","text": "giraffe"}'
```

• Use URL given in creation of the Cloud Function containing the region the function is deployed and your project ID.
  • [YOUR_SLACK_TOKEN] is the verification token provided by Slack in the Basic Information section (see earlier steps)
Using the Slash Command

- Try it out in your Slack environment!
  ```
  /kg giraffe
  ```
- Watch the logs to be sure the executions have completed:
  ```
gcloud functions logs read --limit 100
  ```
Cloud Functions Lab #3

- Clean up
  - To delete just the function, use the command:
    gcloud beta functions delete [NAME_OF_FUNCTION]

- Lab link
  - https://cloud.google.com/functions/docs/tutorials/slack
Extra
AWS Lambda Lab #1 (CS 510 only)

- Serverless 10-minute tutorial