Build a Serverless, Location-Aware, Search & Recommendations-Enabled Application

Lab Guide

Table of Contents

Overview

In this lab, we will build out a simple food delivery service using Amazon Web Services. Below is a high-level architectural diagram that will give you an idea for the components that we will create in the lab. At the end of the lab, you will undestand:

- How to use Amazon API Gateway as the front-door to various Amazon Web Services,
- How to use AWS Lambda as an abstraction layer to interface with services like Amazon Machine Learning, Elasticsearch and Amazon DynamoDB,
- How to deploy "serverless" applications using AWS,
- How to use scaffolding tools to build static web applications,
- How to build user interfaces for geo-based applications,
- The fundamentals of data science and how Amazon Machine Learning can impact your applications.

AWS Region: Virginia (US-East-1)
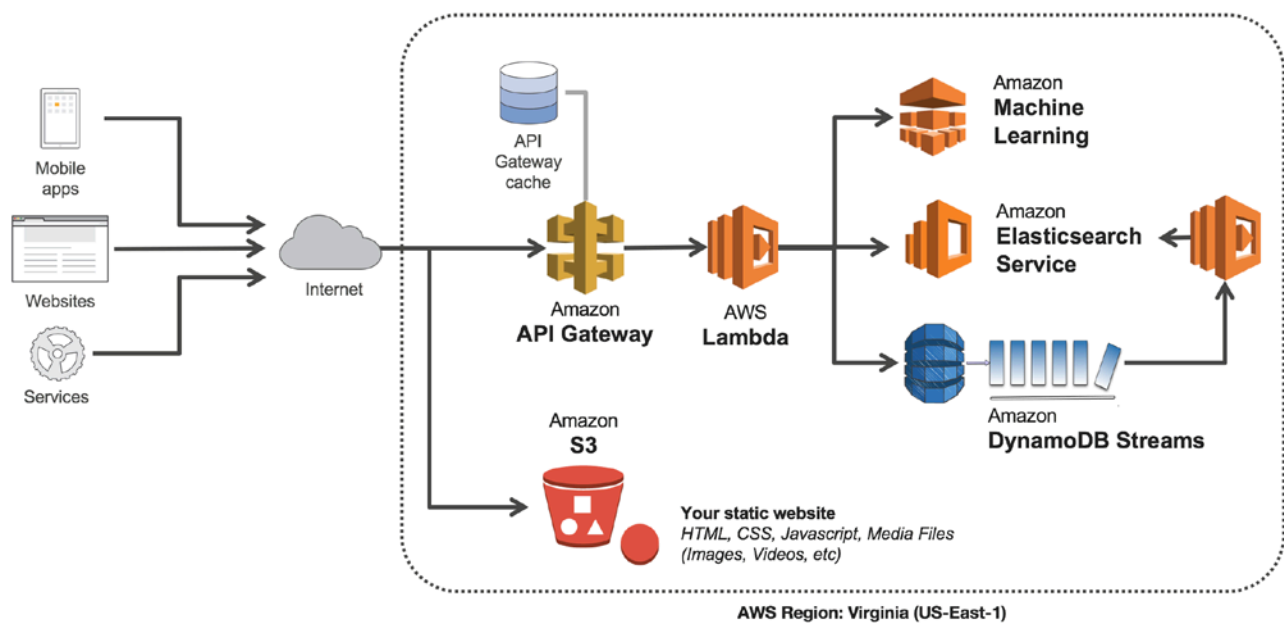
First Steps

Sign into the AWS management console

Before you begin, note the duration and access times listed next to the course title.

- **Duration**: An estimate of the time you will need to complete the course.
- **Access**: The time you have available to complete the course.

For students using Qwiklabs, follow the instructions below:

1. Click **Start Lab**.
2. If you are prompted for an access code, type the numbers in the form and click **Submit**.
3. Under **AWS Management Console**, copy the password to your Clipboard.
4. Click **Open Console**.
5. Sign in to the console with these credentials:
    a. For **User Name**, type **awsstudent**
    b. For **Password**, paste the password from your Clipboard.
6. Click **Sign In**.

For students using their own AWS account, following the instructions below:

1. Sign into the AWS console using your account credentials.

Verify your region in the AWS management console.

You can use provision resources and use services in various AWS *regions*. Regions are dispersed and located in separate geographic areas (US, EU, etc.).

The AWS region name is always listed in the upper-right corner of the AWS Management Console, in the navigation bar.

- Make a note of the AWS region name, for example, US East (Virginia), that your lab is configured for. The AWS region was set for your lab on the qwikLABS launch page.
- For this lab, we will be using the **eu-west-1 (Dublin, Ireland)** region.

Install the Required Development Tools

node.js

The code for this lab is written almost entirely in node.js. To build and run the website and the various scripts supplied with the lab, you will need to have the node.js programming environment installed on your workstation. This open-source software is available at:

https://nodejs.org/

Follow the instructions at that link for downloading and installing node.js for the operating system you are running on you local workstation.

Once node.js has been installed, we will need to install some additional packages and tools. Once tool we will be using extensively for this lab is grunt (http://gruntjs.com/getting-started). Grunt is a build tool for node.js. While node.js isn't a compiled language, "building" a node.js application typically involves running a number of tasks to import library dependencies, minimize javascript code, and package and deploy the code. The grunt tool is essentially a tool for defining and running tasks such as this. We will be using grunt to package, deploy,and configure the application code used by this lab.

To install grunt, execute the following from a command prompt:

```
npm install -g grunt-cli bower yo
```

Text Editor

To edit the source code and configuration files we will be using in this lab, you will need a good text editor. We will be editing a combination of JavaScript and JSON files. Atom is an excellent choice. It has syntax highlighting for all the languages we will be using (and then some), and it runs on all major OS's. It's also available for free:

https://atom.io/

Mapbox API Key

This lab uses a map component to display geo-filtered search results in the client web browser. This component is from Mapbox. It is free to sign up for a developer account, which will include an API key that you'll use when building the web application in this lab. Sign up for a "starter" plan using the link below:

https://www.mapbox.com/studio/signup/

Configure the AWS CLI

7. Download and install the AWS CLI following the instructions found here:
8. http://aws.amazon.com/cli/
9. Configure a new profile to use with the CLI:
    a. For students using Qwiklabs, use the AccessKeyID and SecretAccessKey provided on the QwikLabs launch page by typing this command in a terminal window and following the prompts. The region we will be using is **eu-west-1**.
    b. For students using their own AWS account, use a set of API credentials linked to an IAM user with super-user privileges. If you have a set of credentials already, feel free to reuse this. However, we strongly recommend creating a new CLI profile with these credentials. The region we will be using is eu-west-1.

    ```
    aws configure --profile bootcamp
    ```

10. Download the ZIP file containing the lab source code from the URL provided by your lab instructor. Extract this ZIP file to the directory of your choosing, but choose a path that is simple to remember and to type. This path is going to be referenced repeatedly in this lab, and we will refer to it as *<app_code_path>* from here on out.
11. Let's examine the structure of the lab source code:

File/Path	Description
Gruntfile.js	This file contains the grunt task definitions. All the tasks required to build and deploy this lab are defined in this file. We won't need to edit this file for the lab.
aws.json	This file contains some properties that we will set to run the grunt tasks required to build the lab.

6

bootcamp.json	This file contains some additional properties that need to be set in order to build the lab. The properties in this file will be set automatically with values from the CloudFormation stack we will be running to build the AWS infrastructure for the lab.
es.json	This file contains some additional properties required to build the lab. Like bootcamp.json, the properties in this file will be set automatically once the Elasticsearch cluster is provisioned in AWS.
bower.json	Bower is a package manager for javascript websites. It is used to package all the library dependencies for a javascript-based website into a form suitable for deployment to a web server. We will not need to edit this file for this lab.
package.json	Node.js packages are managed through a tool called the "Node Package Manager" (npm). This file is use to specify the packages required by your node.js application so that they can easily be downloaded and installed using the npm tool. This file defines the packages required by grunt to build the application. We wil not be editing this file for the lab.
src/api	This contains the source code and config files used to build the API that the client website will used to communicate with the backend. We will be going into more detail later in this lab about each file contained in this location.
src/cfn	This contains the Cloudformation template used to build the lab's AWS backend infrastructure. We won't be editing this file for the lab, but we will explore the resources this template defines later in the lab.
src/demo	This contains the source code for several node.js applications that will be used to load and update data in the lab's backend database. We will be executing this code via grunt later in the lab.
src/lambda	This contains the source code for the lambda functions that define the entire server-side logic for the application. We won't need to edit these for this lab, but we recommend you spend some time understanding each of these functions, as this is code that makes the entire lab work.
src/web	This contains the source code for the front-end website that will be used to execute geospatial searches, display recommendations, and show search results on a map. This application is written entirely in javascript and does not require any server side code to run. We will be editing a couple of properties in a config file in this directory before deploying the web site to S3.

Build the Lab's AWS Infrastructure

Now that you've installed node.js, grunt, and downloaded the application source code, it's time to execute the first set of grunt tasks that build the AWS infrastructure used by the lab.

12. We need to configure grunt to run the first set of tasks. We will do this by editing the file *<app_code_path>/aws.json* in the Atom editor. We only need to change two properties in the file, as shown below:

```
{
  "cliProfile": "bootcamp",
  "region": "eu-west-1",
  "accountId": "<your aws account id>",
  "cloudformation": {
    "stackName": "geospatial-rec-engine"
  },
  "es": {
    "domainName": "geospatial-rec-engine"
  },
  "api": {
    "jarKey": "api/aws-apigateway-importer-1.0.3-SNAPSHOT-jar-with-
dependencies.jar",
    "swaggerKey": "api/restaurantDeliveryApi.swagger.json"
  },
  "lambda": {
    "bucket": "<bucket name>"
  }
}
```

2. Set the `accountId` property with the AWS account you're using for this lab. You can see the account Id in the upper right of the AWS console, just to the left of the region name. This is the value you will set for the `accountId` property. The value should be in double quotes, i.e. "accountId": "1233456".

3. The second property we need to set is the `bucket` property in the `lambda` section at the bottom of the file. This property specifies the name of the S3 bucket that will be created to store all the lambda function code, as well as a few configuration files used when deploying the lab. This can be any string value you choose, but it must conform to the S3 naming conventions (http://docs.aws.amazon.com/AmazonS3/latest/dev/BucketRestrictions.html). Here is an example, but keep in mind that bucket names need to be globally unique, so don't use this example literally because there's a good chance somebody else is already using it:

   "bucket": "my-bootcamp-bucket-sko2016"

4. Save the config file in Atom, and open a command prompt. Navigate to `<app_code_path>` and type the following command to install all the node.js packages required by grunt:

   `npm install`

5. From the same command prompt used in the previosu step, type the following commands:

   ```
   grunt bootcamp-init
   grunt bootcamp
   ```

   These commands create the S3 bucket and the CloudFormation stack. Even though the commands will execute quickly, it will take a few minutes for the CloudFormation stack to complete. You can see the status of the CloudFormation stack by navigating to the AWS console and selecting the CloudFormation service. If all goes well, you should eventually see the following:

6. Now we need to generate the bootcamp.json config file. We do this by executing another grunt command from the command prompt used in the previous step:

```
grunt bootcamp-config
```

7. Now we need to create the AWS Elasticsearch cluster. From the command prompt you used in the previous step, type the following command:

```
grunt es
```

8. The Elasticsearch cluster will take some time to provision. Be patient; this can take up to 20 minutes. (If you haven't signed up for a Mapbox account, now would be a good time to do so!) You can check on the status by visiting the Elasticsearch service in the AWS console. You will see something like the following:
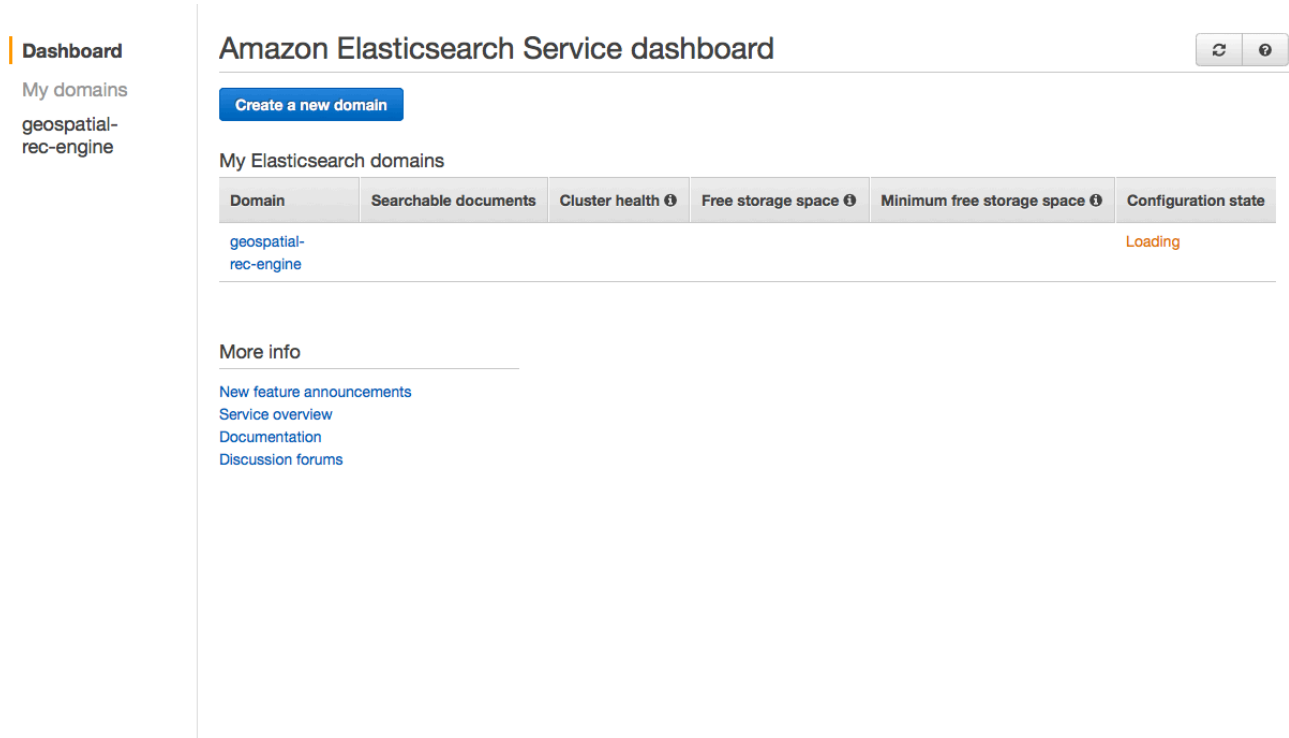
9. Once the configuration state of the Elasticseach cluster shows "active" in the AWS console, run the following commands from the command prompt used in the previous steps. These commands perform some "bootstrapping" of the AWS infrastructure we created using Cloudformation and Elasticsearch services in the previous steps:

```
grunt es-config
grunt bootstrap
```

One of the critical portions of these bootstrapping routines is to configure the Elasticsearch cluster to recognize the geo-location properties of the documents we'll be indexing in this lab. This is required in order to execute geo-filter queries in Elasticsearch. Navigate to the Elasticsearch service in the AWS console and click on the search domain link. From the page that loads, select the "Index" tab, and you should see something like this:

geospatial-rec-engine

Configure cluster **Modify access policy** **Manage tags**

Domain status	Active
Endpoint	search-geospatial-rec-engine-ehae4ds7nl6ejsppzhwgpsbwzm.eu-west-1.es.amazonaws.com
Domain ARN	arn:aws:es:eu-west-1:127436723527:domain/geospatial-rec-engine
Kibana	search-geospatial-rec-engine-ehae4ds7nl6ejsppzhwgpsbwzm.eu-west-1.es.amazonaws.com/_plugin/kibana/

Cluster health **Indices** **Monitoring**

▼ locations

Count	0
Size in bytes	575 B
Query total	0
Mappings	▼ driver
	location
	coordinates *geo_point*
	▼ restaurant
	location
	geocoordinate *geo_point*

10. You should now have a fully functional AWS backend, consistinig of an Elasticsearch cluster, a number of Lambda functions, three S3 buckets, and several DynamoDB tables. Take some time to visit each of these services in the AWS console and explore the resources that were created. As the lab continues, you'll learn more about what each of these do.

Part One: Design and build the API backend.

Overview

In part one, you will:

- Set up an API using Amazon API Gateway
- Connect the API to an AWS Lambda function
- Update indices and searching data from Elasticsearch
- Update and read data from DynamoDB

Topics Covered

By the end of this lab, you will be able to:

- Create all the APIs required by the user interface,
- Transform the request and response,

- Write Lambda functions to execute custom business logic,
- Integrate Lambda functions with Amazon DynamoDB and Elasticsearch

Technical Knowledge Prerequisites

To successfully complete this part of the lab, you should be familiar with the AWS Console and CLI. You should understand the concepts of REST APIs and have basic familiarity with node.js code.

You should also have an understanding of the IAM policy language, JSON format, and familiarity with node.js for writing AWS Lambda functions.

What are RESTful APIs?

REST (Representional State Transfer) is a architectural style that provides a simple yet powerful mechanism to expose APIs (Application Programming Interfaces) and build highly scalable and decoupled applications. Amazon API Gateway makes it easy to author scalable APIs on top of existing AWS or external resources. It also allows transformation of request and response format to match a format suitable for consumers of the APIs.

In this part of the lab, we will walk you though steps to write an API and connect it to AWS Lambda function and Amazon DynamoDB.

Create a Lambda function

1. Confirm that the region in AWS Console is **Ireland** and navigate to Lambda.

2. Click create a Lambda function.

3. Click Skip at the bottom on blueprint page.

4. Enter the name of the function as "saveUserProfile" and copy the code from the *<app_code_path>/src/lambda/users/users.js* file into the Lambda function code editor window.

5. Select the role as "Basic with DynamoDB" and use default value for all other options. This will open a popup window that creates a new IAM role for your Lambda function.

6. Review the policy document and then press "Allow" to create the role.
7. Review the amount of memory the Lambda function has access to and specify the timeout period. **Note**: the amount of memory you allocate to your lambda function will impact the cost of running that function.

8. Press **Next**. Review the configuration and press **Create function**.

Test the Lambda Function

Now that you have created your Lambda function, it is important to test it before putting it into production. Follow the steps below to run a quick test.

1. Click on the Lambda function you just created from the Lambda console page.

2. Click **Test**.

3. A window should display where you can copy and paste JSON based test arguments. Copy and paste the test json shown below from the file *<app_code_path>/src/api/users.test.event.json* Click **Submit.**

```
{
    "method": "POST",
```

```
    "user": {
        "email": "test-user@test.com",
        "gender": 0,
        "age": 2,
        "budgetPreference": 2,
        "deliveryAddress": {
            "address_1": "475 Sansome Street",
            "address_2": "21st Floor",
            "city": "San Francisco",
            "state": "CA",
            "zip": "94133"
        }
    }
}
```

Input test event

It looks like you have not configured a test event for this function yet. Use the editor below to enter an event to test your function with (please remember that this will actually execute the code!). You can always edit the event later by choosing **Configure test event** in the Actions list. Note that changes to the event will only be saved locally.

Sample event template   Hello World   ▼

```json
1 ▾ {
2       "method": "POST",
3 ▾     "user": {
4           "email": "test-user@test.com",
5           "gender": 0,
6           "age": 2,
7           "budgetPreference": 2,
8 ▾         "deliveryAddress": {
9               "address_1": "475 Sansome Street",
10              "address_2": "21st Floor",
11              "city": "San Francisco",
12              "state": "CA",
13              "zip": "94133"
14          }
15      }
16 }
```

4. Your Lambda function will now be invoked, using the JSON text as the sample event. You can use this functionalty to test and debug your Lambda function code.
5. There are 2 possible outcomes, either success and fail.

6. You'll see the following messege is the Test succeds.

Execution result: succeeded (logs)

The area below shows the result returned by your function execution using the context methods. Learn more about returning results from your function.

```
"email": "test-user@test.com",
"gender": 0,
"age": 2,
"budgetPreference": 2,
"deliveryAddress": {
    "address_1": "475 Sansome Street",
    "address_2": "21st Floor",
    "city": "San Francisco",
    "state": "CA",
    "zip": "94133"
},
"userId": "test-user@test.com"
```

7. You'll see the following messege is the Test fails.

Execution result: failed (logs)

The area below shows the result returned by your function execution using the contex

```
{
    "errorMessage": "Process exited before completing request"
}
```

Summary

Request ID	1573ccdc-5e3b-11e5-a235-a57faf238bd5
Duration	233.42 ms
Billed duration	300 ms
Resources	128 MB

8. If your test function fails you can debug by looking at the Cloudwatch Logs generated by it. Scroll down below the inline function textbox to find the results of test run. And click on "**logs**" link.

9. The Cloudwatch Logs dashboard shows all the logs for a given log group.

Dashboard	Log Groups > Streams for /aws/lambda/saveUserProfile		
Alarms	Search Events	Create Log Stream	Delete Log Stream
ALARM 0			
INSUFFICIENT 0			
OK 0	Filter: Log Stream Name Prefix ✕		Log Streams 1-1
Billing	Log Streams		Last Event Time
Logs	2015/09/18/[HEAD]eb304a7c1e934b0a95aca21ad98f0722		2015-09-18 12:25 UTC-7

10. The Cloudwatch Logs dashboard shows all the files generated by AWS Lambda. Click on the relevant file to look into the logs generated by Lambda function.

A common error is that the test JSON was not set in the Lambda function, as seen above. If you see "key3", "key2", "key1", that means you did not change the test JSON in step 3.

Create an API

Now, we will use Amazon API Gateway to create our API set.

1. By the end of this section you will have the following API
    /users/PUT
    /users/POST
2. Access the API Gateway service. If this is the first API you create for the selected region, choose "Getting Started", otherwise choose "Create API".
3. Type "Food Delivery Service" in the "API Name" field and click "Create API".

4. Click **Create Resource** on the top right of the screen

/ Methods   Create Method   Create Resource   Delete API

No methods defined for the resource.

5. Next, type "user" in the "Resource Name" and "Resource Path" fields and click **Create Resource**.

New Child Resource

Use this page to create a new child resource for your resource.

Resource Name*   users

Parent Resource

Resource Path*   / users

You can add path parameters using brackets. For example, the resource path **{username}** represents a path parameter called 'username'.

* Required   Cancel   **Create Resource**

6. Select the /users resource in the "Resources" tree (left). Select "Create Method", choose "PUT" and click the **checkmark** icon.

   ▼ /
      /users
         POST

7. Select the /users resource in the "Resources" tree again and "Create Method", choose "POST" and click the **checkmark** icon.
8. Select the POST function on the left panel and select "Lambda Function" and choose the region "eu-west-1".

9. Select Lambda function created in previous Lambda exercise.

10. You'll see a Method execution screen showing the flow of calls.
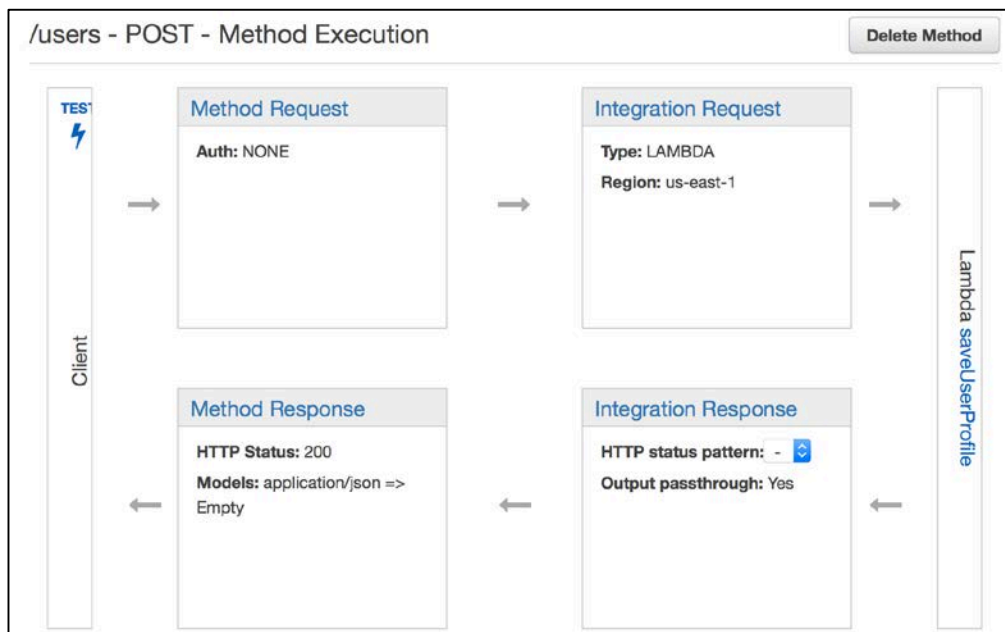
11. Click **Integration Request**. Expand Mapping Templates. Add a content type and enter "application/json" in the text field. Press the **checkmark**.

12. Click the pencil icon right next to "**Input passthrough**", choose "**mapping template**" from the combobox. Copy and paste the mapping template from *<app_code_path>/src/api/users.post.request.template*:

```
#set($inputRoot = $input.path('$'))
{
    "method": "POST",
    "user": {
        "email": "$inputRoot.email",
        "budgetPreference": "$inputRoot.budgetPreference",
        "gender": "$inputRoot.gender",
        "age": "$inputRoot.age",
        "deliveryAddress": {
            "address_1": "$inputRoot.deliveryAddress.address_1",
            "address_2": "$inputRoot.deliveryAddress.address_2",
            "city": "$inputRoot.deliveryAddress.city",
            "state": "$inputRoot.deliveryAddress.state",
            "zip": "$inputRoot.deliveryAddress.zip"
        }
    }
}
```

▼ Mapping Templates

Content-Type	
application/json	application/json Mapping template ◌ ✓ ✗
	Template ↗ Select a model to generate a template ◌
	1	
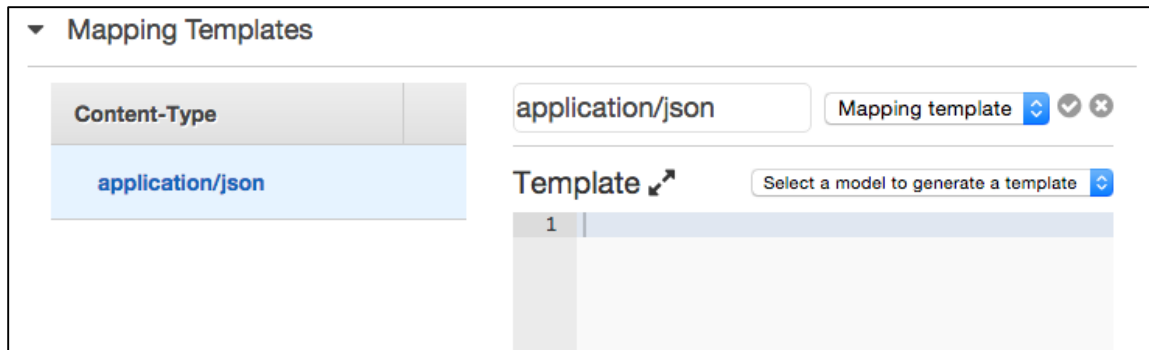
13. Press the **checkmark** to save your changes.

14. You can test the API by sending a JSON from the Test screen. Copy the sample JSON show below from the *<app_code_path>/src/api/users.post.request.test.json* file to try this API:

```
{
    "email": "test-user@test.com",
    "gender": 0,
    "age": 2,
    "budgetPreference": 2,
    "deliveryAddress": {
        "address_1": "475 Sansome Street",
        "address_2": "21st Floor",
        "city": "San Francisco",
        "state": "CA",
        "zip": "94133"
    }
}
```

← **Method Execution** /users - POST - Method Test   **Delete Method**

Make a test call to your **POST** method. Provide the value of the input parameters for your API call.

🟢 **POST** » **/users**

⚡ **Test**

▼ Request Body

```
1 ▾ {
2       "email": "test-user@test.com",
3       "gender": 0,
4       "age": 2,
5       "budgetPreference": 2,
6 ▾     "deliveryAddress": {
7           "address_1": "475 Sansome Street",
8           "address_2": "21st Floor",
9           "city": "San Francisco",
10          "state": "CA",
11          "zip": "94133"
12      }
13  }
```

The expected output is below:

```
Request: /user
Status: 200
Latency: 3128 ms
```

▼ Response Body

```
        {
  "email": "test-user@test.com",
  "budgetPreference": "2",
  "gender": "0",
  "age": "2",
  "deliveryAddress": {
    "address_1": "475 Sansome Street",
    "address_2": "21st Floor",
    "city": "San Francisco",
    "state": "CA",
    "zip": "94133"
  },
  "userId": "test-user@test.com"
}
```

15. Repeat the Steps above to associate the same Lambda function with the PUT method. Note the "method" key in the response mapping template. For the PUT operation, change the value of this key to "PUT."

Congratulations, you have created an API Gateway that will invoke lambda functions as HTTP requests are received.

Now we are going to deploy the entire API. For an API consisting of numerous resources and methods, performing the steps we just walked through manually would be inefficient and error prone. Instead, we can use an API definition language called "swagger" (http://swagger.io) to define all the resources, methods, and integrations we'll be using for the rest of the lab.

To do this, we will run the AWS API Gateway Swagger Importer from a Lambda function. Open up a command prompt and navigate to *<app_code_path>*. Execute the following command:

```
grunt create-api
```

Once this command has completed, return to the API Gateway service in the AWS console. If you refresh the main page, you should see a new API called Restaurant Delivery Service. This API contains all the resources and methods required by the web client to make REST calls to the backend. The REST calls made through API Gateway endpoints call Lambda functions. We'll explore these in detail in the sections that follow.

Now let's load some data so we can make it available for searches in the application. From the same command prompt used above to create the API, run the following commands:

```
cd <app_code_path>/src/demo/drivers
npm install
cd <app_code_path>
grunt load-drivers
cd <app_code_path>/src/demo/restaurants
npm install
cd <app_code_path>
grunt load-restaurants
```

In this next section, we are going to complete the integrations between the API Gateway resources and the Lambda functions. The swagger import created all the resources, methods, and integrations required by the application API. But we still need to map the JSON formats used by the REST API to those used by the Lambda functions. We saw in the previous step how to do this for the `users` resource. We will need to do the same for the other resources in our API. The following table lists the resources and methods in the API that are required in order for the front-end web application to work properly. It also shows the path (relative to *<app_code_path>*) where the mapping template files can be found (where applicable). All of these must be completed before moving onto the section that describes how to build and deploy the web application:

Resource	Method	Request Template File	Response Template File
drivers/search	GET	None required; this mapping was in-lined within the swagger JSON file.	src/api/drivers.search.get.response.template
users	POST	src/api/users.post.request.template	None required. This response uses a "pass through", which means the JSON from the Lambda function is retured as-is to the client calling the API method.
users/{userId}	GET	None required; this mapping was in-lined within the swagger JSON file.	None required; This response uses a "pass through."
restaurants/search	GET	src/api/restaurants.search.get.request.template	src/api/restaurants.search.get.response.template
predictions	GET	None required; this mapping was in-lined within the swagger JSON file.	None required; This response uses a "pass through."

Now let's bring up the API for searching for restaurants on keywords and location:

1. From the API Gateway service page, click the API called "Restaurant Delivery Service".

2. From the left-hand tree, click the GET method under /restaurant/search.
3. Click the Integration Request link from the Method Execution screen.
4. Expand "Mapping Templates" and then click the "application/json" content type.
5. Edit the mapping template by clicking the pencil icon in the upper right. Cut-and-paste the request mapping template shown below from the file *<app_code_path>/src/api/restaurants.search.get.request.template*. This template is mapping the URL query string parameters into the event object that will be passed into the Lambda function.

```
{
    "searchTerms":"$input.params('searchTerms')",
    "locations": "$input.params('locations')"
}
```

6. When done editing the template, click the checkmark icon to save the changes.
7. Click on "Method Execution" at the top of the screen.
8. Click on "Integration Response."
9. Expand the "Method Response Status 200."
10. Expand the "Mapping Templates" and then click the "application/json" content type.
11. Edit the mapping template by clicking the pencil icon in the upper right. Cut-and-paste the response mapping template shown below from *<app_code_path>/src/api/restaurants.search.get.response.template*. This template is mapping the response generated by the Lambda function to the form required by the client. In this case, we are transforming raw search results returned from an Elasticsearch query in the Lambda function to a form that the mapping API can understand.

```
#set($inputRoot = $input.path('$'))
{
  "type": "FeatureCollection",
  "features":[
 #foreach($elem in $inputRoot.restaurants)
     {
         "type": "Feature",
         "geometry": {
             "type": "Point",
             "coordinates": ["$elem.location.geocoordinate.lon",
"$elem.location.geocoordinate.lat"]
         },
         "properties": {
             "title": "$elem.name",
             "description": "$elem.cuisine_type",
             "marker-color": "#FDCA00",
             "marker-size": "medium",
             "marker-symbol": "restaurant",
             "price": "$elem.price",
             "restaurantId": "$elem.restaurantId"
         }
     }
     #if($foreach.hasNext),#end
   #end
  ]
}
```

12. Click the checkbox to exit out of the editor and then click the blue "Save" button to save the changes.

13. Click on "Method Execution" at the top of the screen.
14. Click the "Test" icon.
15. Type "burgers" for the searchTerms parameter, and type a space in the locations parameter. The Lambda function will execute an Elasticsearch query with the term "burgers."

If all goes well, you will see some search results. Play around with different search terms. The search can also filter results based on geo-location, and we will cover how that works in the next section, when we bring up the UI.

You will need to repeat a similar set of steps to enable the methods for each of the other resources, such as driver, prediction, and user. We covered the user resource in the last section, and the process for assigning the request mapping template is identical.

For the /drivers/search resource, the process of enabling this is identical to the steps we just completed for the /restaurants/search resource. The only difference will be the request and response mapping templates used by this resource. For the /drivers POST method, you can use this template in
<app_code_path>/src/api/drivers.post.request.template for the integration request input:

```
#set($inputRoot = $input.path('$'))
{
    "operation": "POST",
    "driver": {
        "driverId": "$inputRoot.driverId",
        "car": {
            "make": "$inputRoot.car.make",
            "color": "$inputRoot.car.color"
        },
        "location": {
            "coordinates": $inputRoot.location.coordinates
        }
    }
}
```

For the /drivers/search GET method, you can use the template in
<app_code_path>/src/api/drivers.search.get.response.template for the integration response output:

```
#set($inputRoot = $input.path('$'))
{
  "type": "FeatureCollection",
  "features":[
    #foreach($elem in $inputRoot.drivers)
        {
            "type": "Feature",
            "geometry": {
                "type": "Point",
                "coordinates":
"[$elem.location.coordinates[0],$elem.location.coordinates[1]]"
            },
            "properties": {
                "driverId": "$elem.driverId",
                "title": "$elem.car.color $elem.car.make",
                "description": "$elem.car.color $elem.car.make",
                "marker-color": "#FDCA00",
                "marker-size": "medium",
```

```
                    "marker-symbol": "driver"
                }
            }
        #if($foreach.hasNext),#end
    #end
  ]
}
```

This next section is optional. You do not need to be able to insert or update drivers or restaurants through the API for the web application to work properly. However, this is a good exercise if you want to get more experience working with API Gateway.

Once you have the /restaurant and /driver POST methods working, go ahead and create some new restaurants and drivers. The JSON format is defined in the restaurant and driver models, which you can view by selecting the Resources->Models menu item from the top of the screen. You can also see examples of the JSON for restaurants and drivers in the files drivers.json and restaurants.cuisine.map.final.json in the directory containing the artifacts loaded from S3.

When you create a new restaurant or driver, the Lambda function will insert the data into DynamoDB. You can view the data in the DynamoDB tables directly from the DynamoDB console. The Lambda function will also index the data using Elasticsearch. Check the commands.txt file for some useful command lines for viewing the index data directly from Elasticsearch.

Deploy to "dev" stage

Select the "Deploy API", choose "New Stage" and type "dev" in the "Stage name" field.

Configuring the deployed API
Configure the deployed API according to the following:

Congratulations! You have successfully deployed your API gateway. Now, let's get the web application up and running and test the API's we just deployed.

Part Two: Build a Serverless Web Application

Overview

This lab will lead you through the steps to:

- Download and customize web application files for mapping data.
- Setup a Node.JS development and testing environment.
- Modify config.js to include your API Gateway URL.
- Upload web application files to an S3 bucket.
- Create and apply a bucket policy to allow public access to the S3 bucket.

Topics Covered

By the end of this lab, you will be able to:

- Create and deploy static web applications to S3.
- Understand and implement the AWS SDK for Javascript.
- Understand how to implement
  Configure an S3 bucket to host a static website.

Technical Knowledge Prerequisites

To successfully complete this lab, you should be familiar with the Amazon S3 service. You should understand the concepts of bucket and object, and how to perform put and get operations on objects in an S3 bucket using the S3 console or AWS CLI.

You should also have an understanding of the IAM policy language and how to use bucket policies to provide access to your S3 resources.

Finally, you should have practical experience building web applications with common web programming languages. In this boot camp, you will use a Node.JS scaffolding tool to build out an HTML5 website with Angular.JS. Experience using the AWS SDK for Javascript is also recommended.

What is a Static S3 Website?

The web application you will build out in this lab will run entirely on S3 and in the client's browser. By contrast, a dynamic website relies on server-side processing, including server-side scripts such as PHP, JSP, or ASP.NET. Amazon S3 does not support server-side scripting.

Build the Web Application

Open a command prompt and navigate to *<app_code_path>* and run the following command:

```
bower install
```

Developing with Yeoman

The first step will be to modify the **main.js** file with the details of your stack.

1. Next, navigate to the javascript file *<app_code_path>/src/web/scripts/main.js.*

2. Update the values at the top of the file with your values from above.

    MAPBOX_API_TOKEN: Your Mapbox API Access Token.
    Description: There will be a default access token in this location. You can use this key, or create your own token on Mapbox.

    AWS_API_GATEWAY_ENDPOINT: Your API Gateway Endpoint
    Description: This is where you will copy and paste the base endpoint for your API that you deployed in the previous exercise. To determine this, go to the API Gateway console and click the "Restaurant Delivery Service" link. Then select the "Resources" menu and select "Stages." From the left-hand menu, select the stage you created in the previous section. The URL will be displayed on the main part of the screen.

    Yeoman comes packaged with a task manager called **grunt.** In the web application files, we have preconfigured a few grunt tasks. To run these tasks, you will need to open your favourite shell. On a mac, this could be **Terminal**, on a PC this could be **PowerShell**.
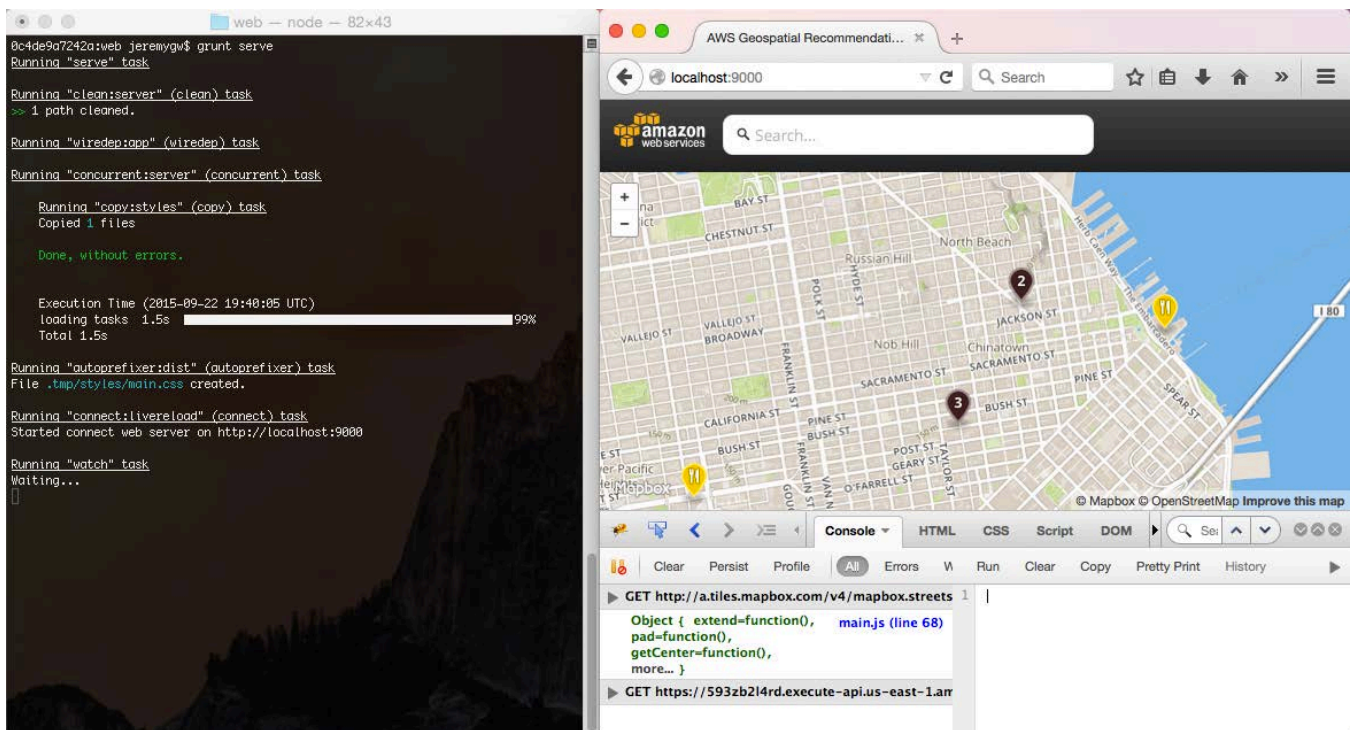
Figure 1: Running grunt serve to create a local webserver at http://localhost:9000/ with Firefox and Firebug.

grunt serve
This will run a local development version of your application. Once you run this command, a local webserver will start and your website will be available at: http://localhost:9000/#/

grunt test
This is your test harness. In the future, you could create unit tests and include them. An example test is included in the web application code.

grunt build
*This command will package and minify your application source code to be most efficient in the browser. Once completed, your web files can be found in the **dist** folder.*

grunt publish
*This command will run the build command from above and then copy all of the files in the **dist** folder into your specified S3 bucket. Once the files have been copied to S3, your S3 website will be deployed and ready to accept incoming traffic.*

To quickly get started, simply type **grunt publish** and navigate to your Amazon S3 Static Website URL in your browser. It will follow this structure: http://[BUCKET_NAME].s3-website-eu-west-1.amazonaws.com

Congratulations! You have successfully launched your static S3 website that will make RESTful API calls to your API Gateway as well as plot mapping data using Mapbox. You should be able to search for restaurants on keyword, and see those search results filtered for the geocoordinates of the map displayed in the browser.

Now, let's move on to build out the Amazon Machine Learning components, which will enable the "Will I like this restaurant?" functionality to work. The API gateway endpoint is available for doing this prediction, but we first need to train the Machine Learning model before the predictions will work.

Part Three: An Introduction to Machine Learning

Overview

In this section, we will:

- Download a data set used to train a model in the Amazon Machine Learning service
- Examine and understand the attributes used to train the model
- Train and evaluate the model using Amazon Machine Learning
- Make batch predictions using the model

Topics Covered

By the end of this lab, you will be able to:

- Define a new data source,
- Create a new ML model from the data source,
- Evaluate the performance of the ML model,
- Use the model to make batch predictions

Technical Knowledge Prerequisites

To successfully complete this lab, you should be familiar with the Amazon S3 service. You should understand the concepts of bucket and object, and how to perform put and get operations on objects in an S3 bucket using the S3 console or AWS CLI.

You should also have an understanding of the IAM policy language and how to use bucket policies to provide secure access to your S3 bucket from the machine learning service.

What is Amazon Machine Learning?

Amazon Machine Learning (Amazon ML) is a robust machine learning platform in the cloud that allows software developers to train predictive models and use them to create powerful predictive applications.   Amazon ML allows business domain experts and software developers to focus on the problems they are trying to solve using predictive models rather than running and maintaining the compute and storage infrastructure required to train a supervised machine learning model.

Amazon ML can be used to make predictions for a variety of purposes. For example, you could build a model in Amazon ML that will predict whether a given customer is likely to respond to a marketing offer. Amazon ML creates models from "supervised" data sets. This means that the model is based on a set of previous observations. This set of observations consists of features or attributes as well as the target outcome. In our marketing offer example, the features might include the age, profession, and gender of the customer. The target outcome (also called the target variable) would be whether that particular customer responded to the marketing offer or not.

The process of creating a model from a set of known observations is called "training." Once you have trained a model in Amazon ML, you can then use the model to predict outcomes from a set of attributes that matches the attributes used to train the model. Amazon ML scales so that you can make thousands of predictions concurrently. This is important, as today machine learning is often used to provide predictions in near real-time. In this lab, we will be using a machine learning model to predict which restaurants a customer is likely to favor based on the results of a search query.

Components of Amazon ML

Amazon ML contains the following components:

Data Set
A delimited text file containing the attributes and target variable used to train the model.

Create a Datasource

1. The name of the file containing the data that will be used to create the dataset is **<app_code_path>/data/restaurants.data**.
2. In the **AWS Console**, select the *CloudFormation* service from the list of available services.
3. From the list of stacks, select the stack with the name **splXX-geospatial-rec-engine**.
4. Select the **Outputs** tab and note the value of the S3 bucket for the key called *"DataSourceBucket."* Cut and paste this value into a text editor. This is the name of an S3 bucket that will be used in subsequent steps.
5. Upload the data file to S3 using the following command (replace [BUCKET_NAME] with the name of the S3 bucket noted in the previous step):

```
aws s3 cp <app_code_path>/data/restaurants.data s3://[BUCKET_NAME]/data/restaurants.data
```

6. Open the Machine Learning service from the list of services in the AWS Console.
7. Click the blue "Get Started" button.
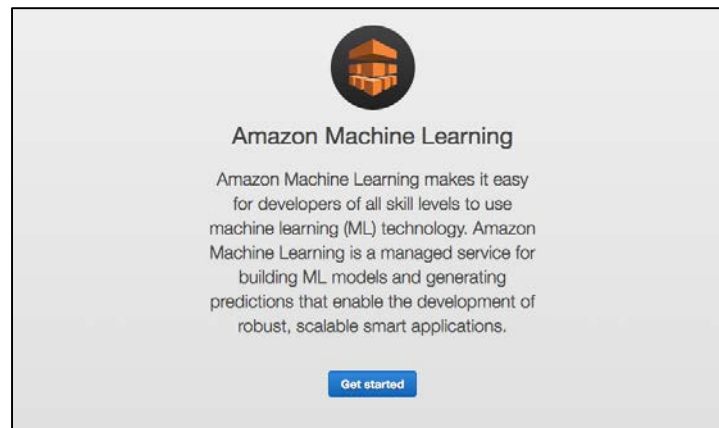
11. Click the grey "View Dashboard" button.
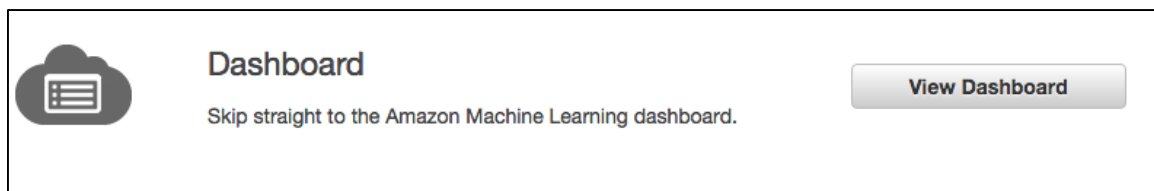
12. Click the "Create New…" button and select "Datasource."

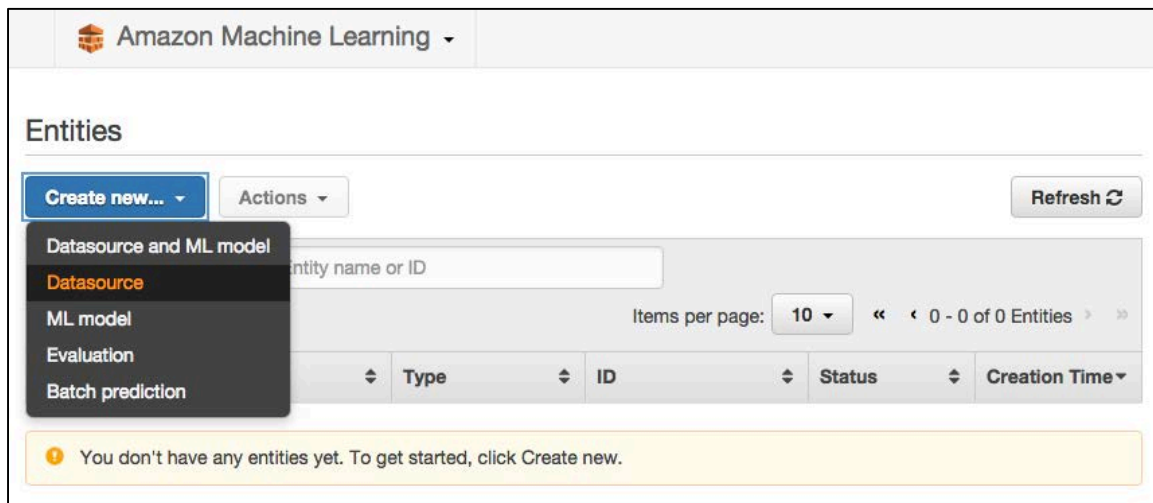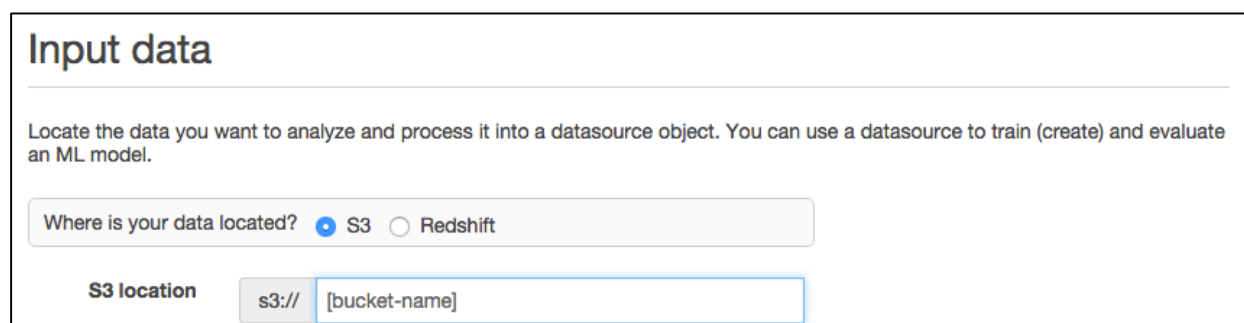13. If not selected by default, select S3 as the location for the data.

14. In the S3 Location text box, cut and paste the S3 bucket name you determined by looking at the CloudFormation stack.
15. After cutting and pasting the bucket name, select the "data" folder when prompted.
16. Select the restaurants.data file when prompted.
17. Click the "Verify" button.

✔ The validation is successful. To go to the next step, click Continue

18. Once the data file has been successfully validated, click the blue "Continue" button.
19. Select "Yes" at the prompt asking if the CSV file contains column names in the first line.
20. The structure of the data file will now be displayed in the table. For each column, you'll see it's name and data type, as well as some sample values. Amazon ML does it's best to guess the data type based on the values found in the file. We don't need to change anything for this lab, but if you did need to alter the data type for a column, you could do so by selecting it from the drop-down menu.
Click the blue "Continue" button.
21. On the next screen, select "Yes" when asked if the datasource will be used to train a model.
22. Now we need to tell Amazon ML which column in the datasource represents the "target variable." Remember, the target variable is the value model will predict. So we want to pick the column that has the known values for our target variable. In this case, it is the "rating" column (the last row). Select the column and click the blue "Continue" button.

23. When prompted for a row identifier, select No and click the blue "Review" button.

24. Click the blue "Finish" button.

Create an ML Model from the Datasource

In this section, we will create a model from the datasource created in the previous step. The datasource contains the data that will be used to train and validate the model created in this step. The datasource also contains metadata, such as the column datatypes and target variable, which will be used by the algorithms that create the model. You can create multiple models from a single datasource, and this is useful when evaluating whether the parameters used to create one model perform better or worse than the parameters used to create a second model. In fact, as we'll see later in this lab, evaluating the performance of a model is a key step in the process, and it's not uncommon to iterate over several models before landing on the one that generates the most accurate predictions.

1. In the AWS console, select the "Machine Learning" service from the list of services.
2. Select the datasource created in the previous step and select "Create ML Model" from the "Actions" menu.
3. Leave the "default" settings and click the blue "Review" button.
4. Click the blue "Finish" button from the next screen, which shows the settings for the ML model.
5. When the model is complete, the state will change from "Pending" to "Complete."
6. Note the Model ID at the top of the screen. You will need this in the next section.

Evaluate an ML Model

Part of the process of creating an ML model in the previous step involves evaluation of the generated model. The Amazon Machine Learning service does this automatically as part of the model creation process. The datasource used to create the model is actually split into two different data sets. The first, which consists of 70% of the data, is used to train the model. The second, which consists of 30% of the data, is used to evaluate the model. This evaluation step is critical, as it shows how well the model performs.

During the evaluation step, each line in the portion of the data reserved for evaluation is run through the generated ML model and a prediction is generated. Since the correct value of the target variable is already known for these items, we can compare the predicted value with the known value. Ideally, we want the generated ML model to correctly predict the known target values a high percentage of the time. But while it might seem counterintuitive, we actually don't want to see the model correctly predict the target values 100% of the time. The reason is that this could be a sign that the model is "over-fitted," which means that instead of generalizing, the model is actually just memorizing the dataset used to train it. This is also why we reserve 30% of the lines in the datasource for evaluation. If we used these to train the model as well as evaluate it, we'd have no way of knowing if the model was making correct predictions because it had memorized the data used during the evaluation, or if it's actually generalizing the data in such a way that it's giving accurate predictions.

The ML model created in this lab exercise is called a "multiclass classification," which means the model is going to predict one value from a set of possible values or classes. To assess this type of model, we want to see how many times the correct class was predicted from the training data set by comparing the predicted value with the actual target value in the training data. To visualize this, we will use a tool called a "confusion matrix," which Amazon ML creates for you as part of the process of creating an ML model.

1. In the AWS console, select the "Machine Learning" service from the list of services.
2. Click on the ML model created in the previous step.
3. From the "ML Model Summary" screen, click the "Evaluation" section from the left-hand menu.
4. Click "Summary" from the first evaluation listed under the "Evaluation" section.
5. From the "Evaluation Summary" screen, click the "Explore Model Performance" button.
6. From the next screen, take some time to explore the model performance by mousing over the individual cells in the confusion matrix. As the mouse hovers over each cell, some statistics will be displayed.

The confusion matrix is structured as follows. The rows represent the true values. The columns represent the predicted values. For each true value in the evaluation dataset, the matrix will show the items were assigned each of the predicted values. The diagonal of the matrix are the cells where the true value is the predicted value. In a well-trained model, these should be colored in dark blue, meaning that the model is correctly predicting the true value most of the time. In fact, some of the cells in the diagonal may actually show that the predicted value matches the true value 100% of the time. A perfect model would show that 100% of the true values matched 100% of the predicted values across all the cells of the diagonal. All the other cells in the matrix would be colored white, meaning there were no incorrect predictions.

However, most models are not going to behave this way and this isn't necessarily a bad thing. A model that correctly predicts the outcome 100% of the time is often likely to be "over-fitted." In most cases, a perfectly good

model may make some incorrect predictions. In the confusion matrix, the incorrect predictions are represented by the cells where the intersection of the row and column doesn't match. Remember—it's okay if the model doesn't perform perfectly. It's common to expect some incorrect predictions. But what you don't want to see in the confusion matrix are dark colored cells where the true value doesn't match the predicted value. This would mean that the model is making incorrect predictions a large number of times. But if you see some light colored cells where the true value doesn't match the predicted value, hover the mouse over the cells to see the percentage of items that fell into this bin. Low numbers aren't a problem here. But higher numbers are. Anything around 50% means the model is only generating predictions that are about as accurate as flipping a coin – in other words, you could flip a coin and decide that "heads" means the true value and "tails" means the predicted value, and your predictions would be as good as the models! Obviously, this isn't desirable. So while 100% accuracy is not a requirement for a well performing model, you do want to minimize the inaccurate predictions.

What do you do in the event that a model isn't performing well? Typically, you will need to iterate. You'll need to go back to the original data set and look to see if it needs further refinement or cleansing. Remember—the quality of your model is only as good as the quality of data you use to train it. Over-fitting is a common problem. Perhaps your model isn't performing well because it will only make a correct prediction for an item if the set of features matches exactly one that it has seen before. In this case, the model has essentially memorized the data. It will correctly predict only those items it's seen before, and will incorrectly (or randomly) predict those items with feature sets it hasn't seen before. In machine learning, it is common to iterate numerous times before arriving at a model that performs well and generalizes the dataset without over-fitting it.

Generating Predictions from an ML Model

There are two ways to generate predictions from an ML model: batch mode, and real-time mode. Batch mode is asynchronous. In this mode, you define a new datasource consisting of the observations for which you want to generate predictions. Like the datasource used to train and evaluate the model, the datasource used for a batch prediction is stored in S3 and consists of a delimited CSV file. In fact, the datasource used for a batch prediction is basically identical to the one use for training and evaluating the ML model with one key difference: the observations in the batch datasource don't contain known values for the target variable.

The second way to generate predictions is in "real-time." This is the approach we will use for this lab, as the predictions we want to generate are based on information being submitted through the web application. To do this, we must enable real-time predictions for our ML model.:

1. In the AWS console, select the "Machine Learning" service from the list of services.
2. Click on the ML model created in the previous step.
3. From the "ML Model Summary" page, click the "Create Real-Time Endpoint" button.
4. Note the URL that appears once real-time predictions have been enabled. You will need this in the next step.

Once real time predictions have been enabled, you can make API calls to generate predictions and receive the results in real time. The code that does this is a Lambda function, and the source code is at *<app_code_path>/src/lambda/predictions/prediction.js.*

Configure the API Gateway API to Use the ML Model for Predictions

In this section, we will configure the API Gateway API we deployed in the previous section to actually make predictions using the ML model we just created. To do this, we will use a DynamoDB table called "config" that will contain the ML model ID and the URL of the real-time prediction endpoint created in the previous step.

1. Open the DynamoDB console.
2. From the list of tables, double-click the row with the name "config." This will open up the table browser for the "config" table.
3. From the "Browse Items" tab, click the "Create Item" button.
4. On the "Put Item" tab, select "Text" from the drop-down menu that shows "Tree."
5. Replace the displayed JSON with this:

```
{
  "setting": "ml",
  "modelId": "[YOUR ML Model ID]",
  "endpoint": "[YOUR PREDICTION ENDPOINT]"
}
```

6. Replace [YOUR ML Model ID] value with the ML model ID you noted in the previous step in which you created the ML model.
7. Replace the [YOUR PREDICTION ENDPOINT] value with the URL you noted in the previous step when you enabled the model for real-time predictions.
8. Click the "Save" button and then click "Ok" to dismiss the confirmation dialog that appears.
9. Click the "Browse Items" tab and then click "Ok" to dismiss the confirmation dialog that appears.
10. Click the "Go" button to rescan the "config" table. You should now see two items in the table, including the item you just created.

Bringing it all together

So far, you have created:

- An Amazon API Gateway
- Amazon Lambda Functions
- A single node Elasticsearch cluster
- A DynamoDB table that uses DynamoDB streams
- A serverless static S3 website
- An Amazon Machine Learning model with real-time predictions.

Now, it is time to see everything in action! The web application you created in **Part 2** will be live and ready to go. Simply type in the S3 URL and you will be able to start searching for restaurants with the application. During this process, you will be returning data from **Elasticsearch on EC2** as well as real-time predictions from **Amazon Machine Learning** through **Amazon API Gateway** and **Amazon Lambda**.
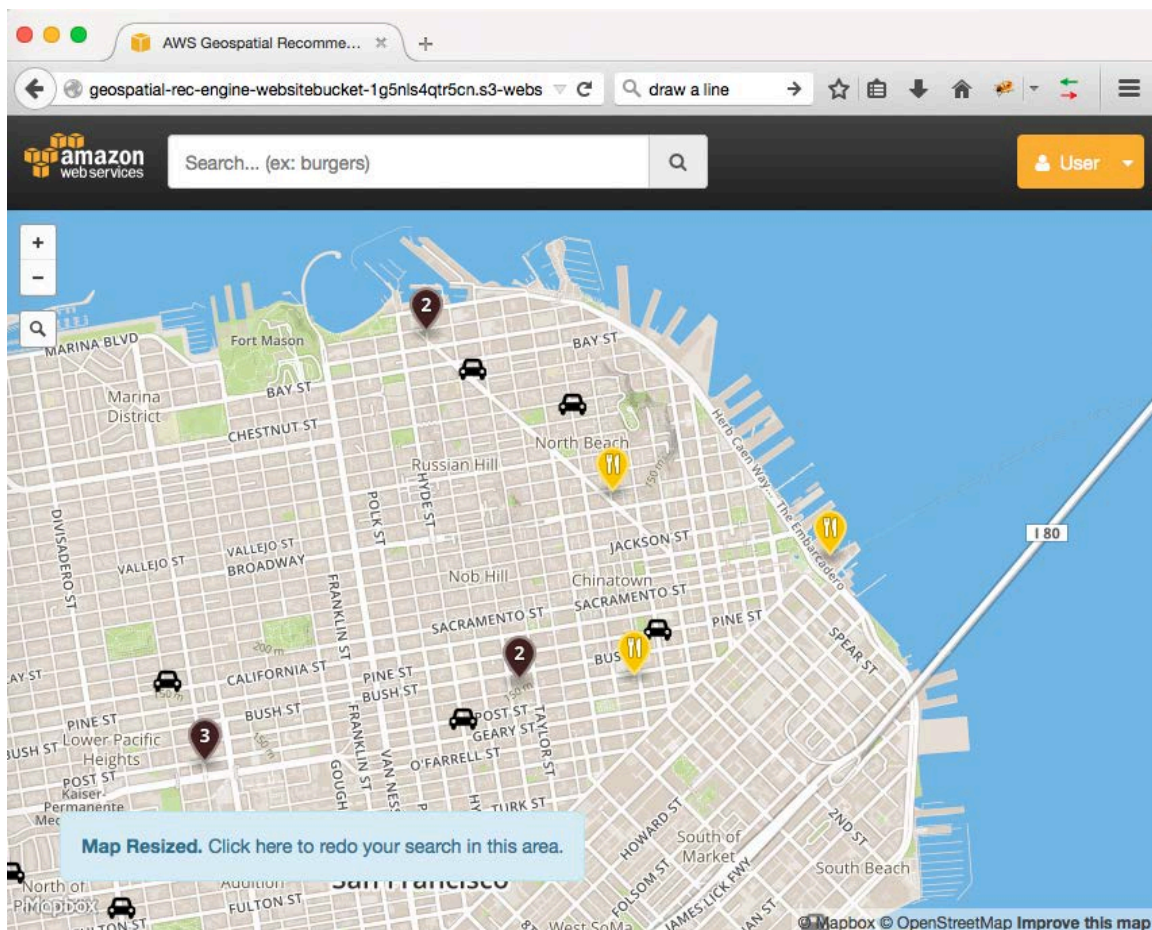
Take a moment to look at the code for the Lambda functions as well as the code for the web application. Feel free to make some modifications!

Simulate real-time driver locations

We will use DynamoDB streams real-time propagation to Elasticsearch to look at driver positions as they are driving in the city. We have written a script which updates the driver location periodically over a given route in the city. The script can be found:

<app_code_path>src/demo/drivers/move_drivers.py

You will need python to run this script. For more information about how to download and run python applications, click here:

https://www.python.org

What's Next

- Write the code to complete all of the methods in the API Gateway.
- Improve the search capability.
- Implement typeahead for search results in the query text box.
- Add functionality to drivers.
- Implement better mapping of error handling from Lambda errors to API Gateway HTTP status codes.
- Add credentials to the endpoints.
- Implement web identities with Cognito and IAM roles.
- Generate API Gateway SDK. Use that instead of basic API Gateway endpoints.
- Configure and use your own domain name.
- Add CloudFront CDN in front of your S3 resources.
- Add functionality to create and manage orders.