Serverless Immersion Day

Getting Started with Amazon API Gateway

September, 2017

Table of Contents

Overview

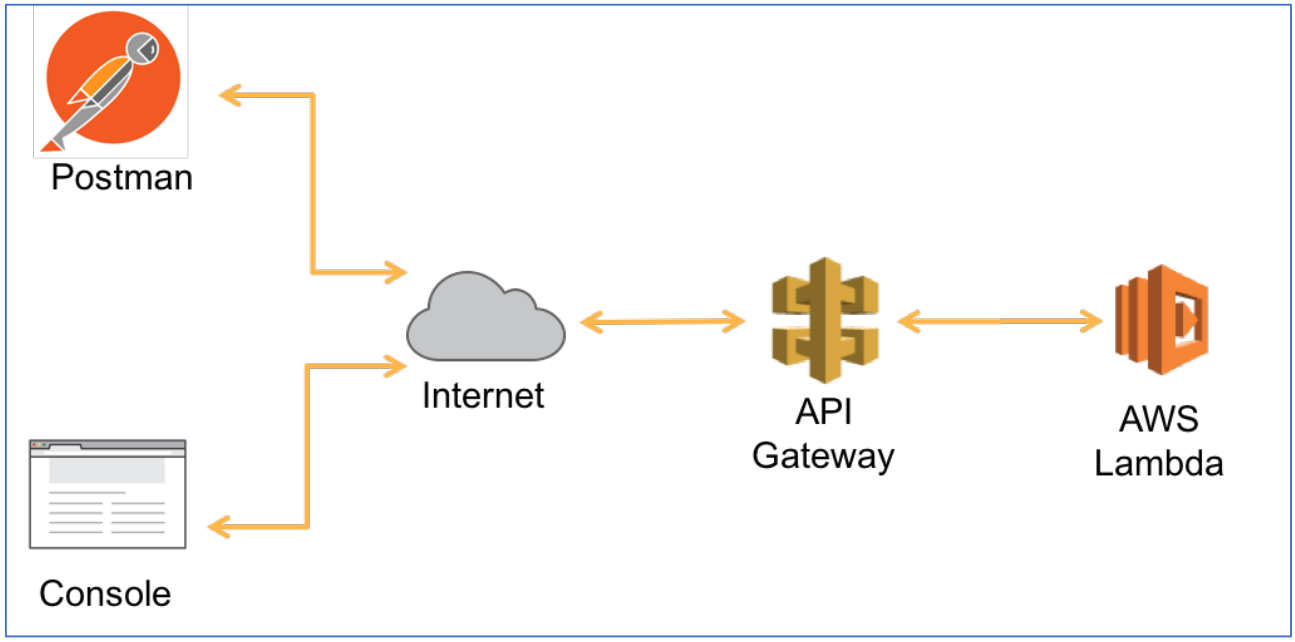


Figure 1: Lab Architecture

Amazon API Gateway is a fully managed service that makes it easy for developers to create, publish, maintain, monitor, and secure APIs at any scale. API Gateway acts as a "front door" for applications to access data, business logic, or functionality from your back-end services, such as workloads running on Amazon Elastic Compute Cloud (Amazon EC2), code running on AWS Lambda, or any Web application.

In lab 1, you will create an API that invokes a lambda function to calculate the price of a house (see appendix – lambda function overview). In the first part of the lab, you will deploy a pre-defined sample lambda function and create the back-end API. You will test your APIs using either the API Gateway console or Postman. In lab 2, you will learn how to cache service response, throttle requests, and create various usage plans. The overall architecture for the lab is depicted in figure 1 above.

Lab Pre-Requisites

In order to complete this immersion day lab, you'll need an AWS Account with access to create AWS IAM, S3, Cognito, Lambda, and API Gateway. The code and instructions in this workshop assume only one student is using a given AWS account at a time.

- Install AWS CLI - Follow the AWS CLI Getting Started guide to install and configure the CLI on your machine.
- Deploy CostCaluculator and MedianPriceCalculator Lambda Functions – See Appendix for deployment instructions
- Install Postman - https://www.getpostman.com/apps

Lab 1: Creating Your First API

1. Sign into the AWS Management Console and open API Gateway console at https://console.aws.amazon.com/apigateway
2. If this is your first API, you will see the Amazon API Gateway welcome page. Click on '**Get Started**'. Otherwise, select '**Create API**'.

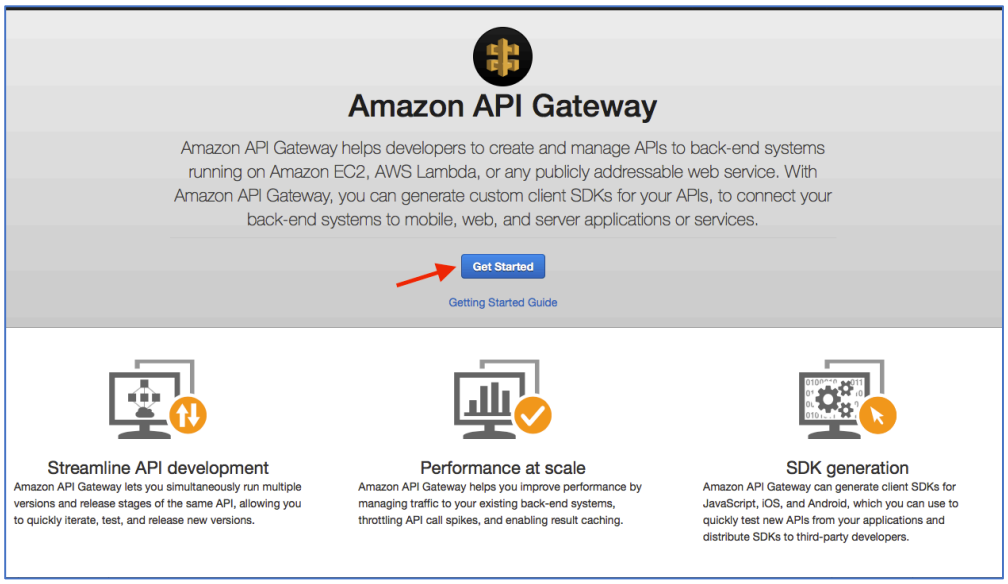


Figure 2: API Gateway Main Console

3. Select "**New API**"
4. Under Settings, Enter the following values:
   API Name: **calculatePrice**
   Description: **Calculates the price of a house per square meters**

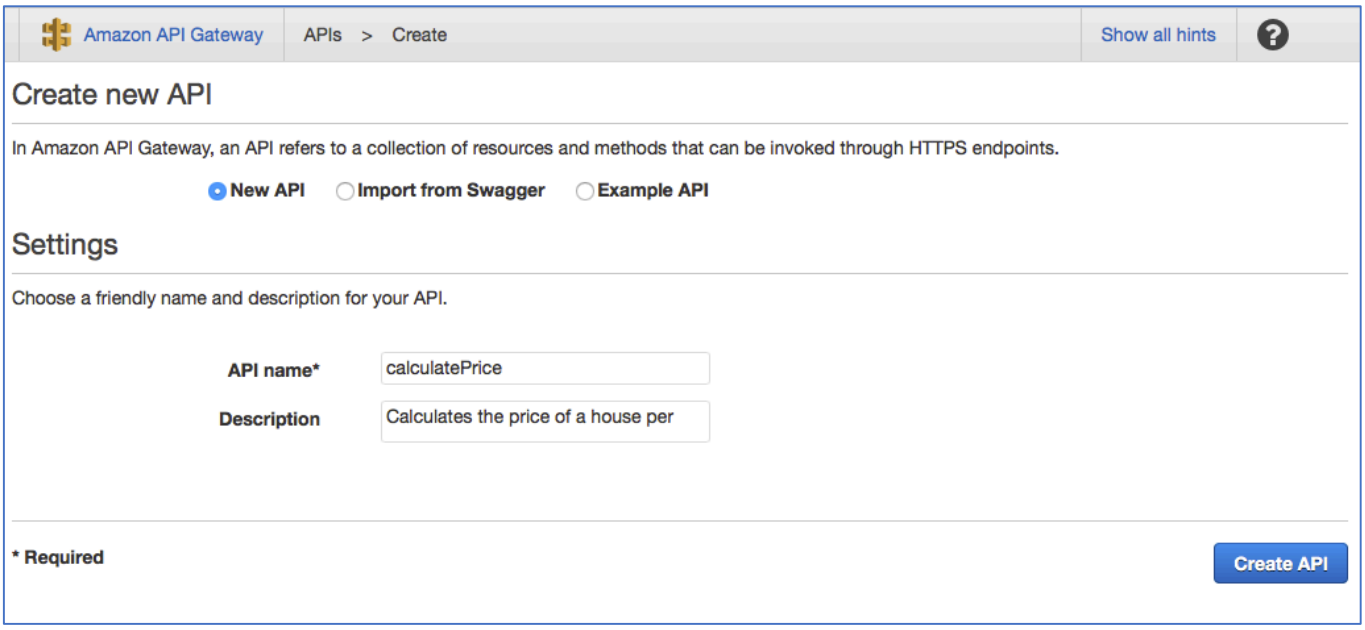


Figure 3: Creating a new API

5. Click '**Create API**'
6. Leave 'Edge Optimized' default
7. Under *Actions*, choose **'Create Resource'**

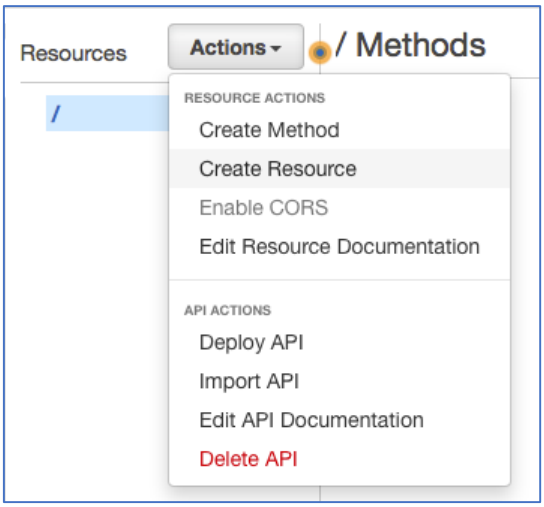


Figure 4: Creating a new resource

8. Enter the following values:
   Resource Name: **pricePerMeter**
   Resource Path: **pricePerMeter**
9. Click ' **Create Resource**'

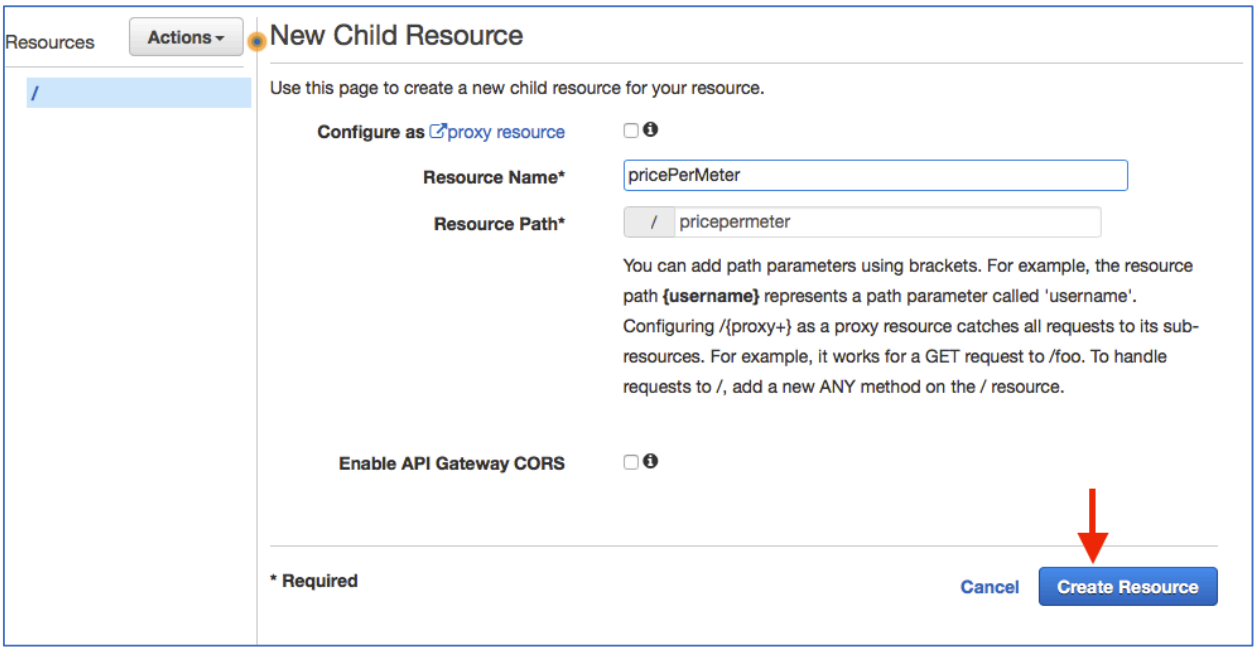


Figure 5: Resource Name and path

10. Select *Actions,* choose **'Create Method'**

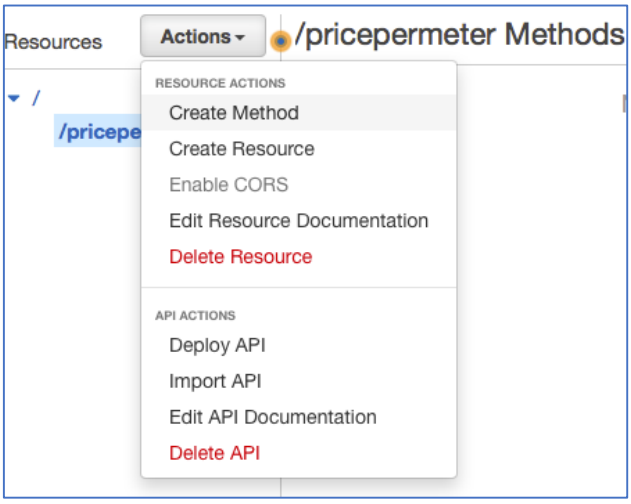


Figure 6: Creating a method

11. Choose '**Post'** and Click on ***check*** *icon* to confirm.
12. Under the *Post* method, Choose the following values:
    Integration Type: **Lambda Integration**
    Lambda Region: **us-east-1**
    Lambda Function: **serverless-immersion-day-stac-CalculateCostPerUnit-XX**
13. Click **Save**.

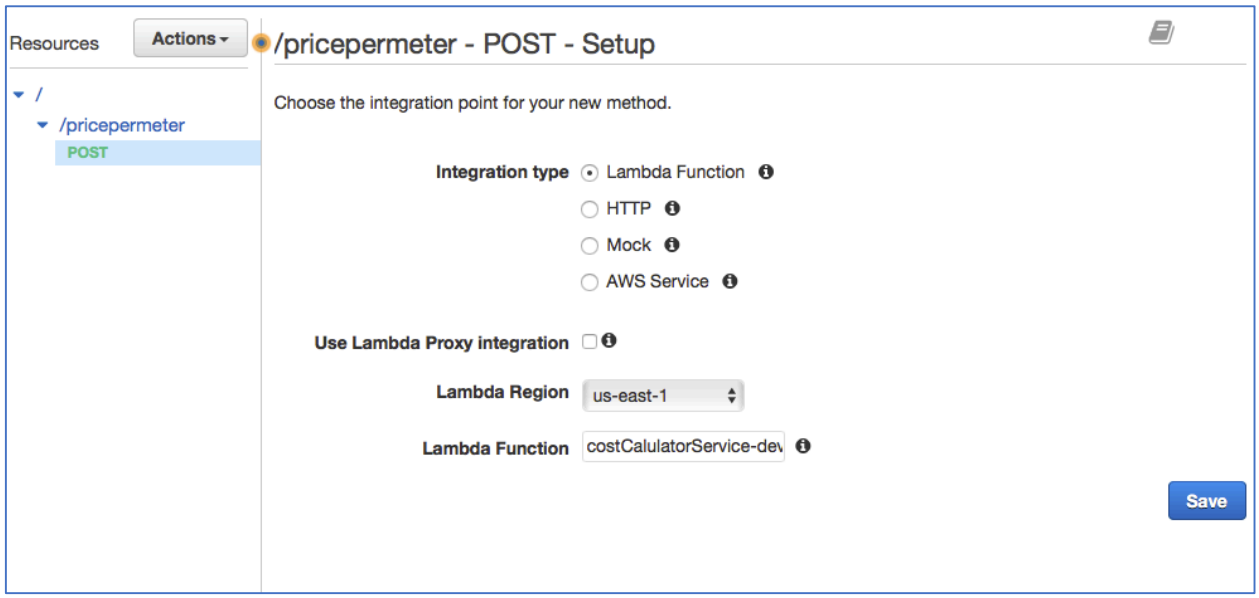


Figure 7: Configuring POST action

14. In the dialogue box, "Add Permission to Lambda Function", Click '**OK**'.

You have now configured your API to call the 'costCalculator' lambda function. The API Gateway pane shows the request and response flow of data.

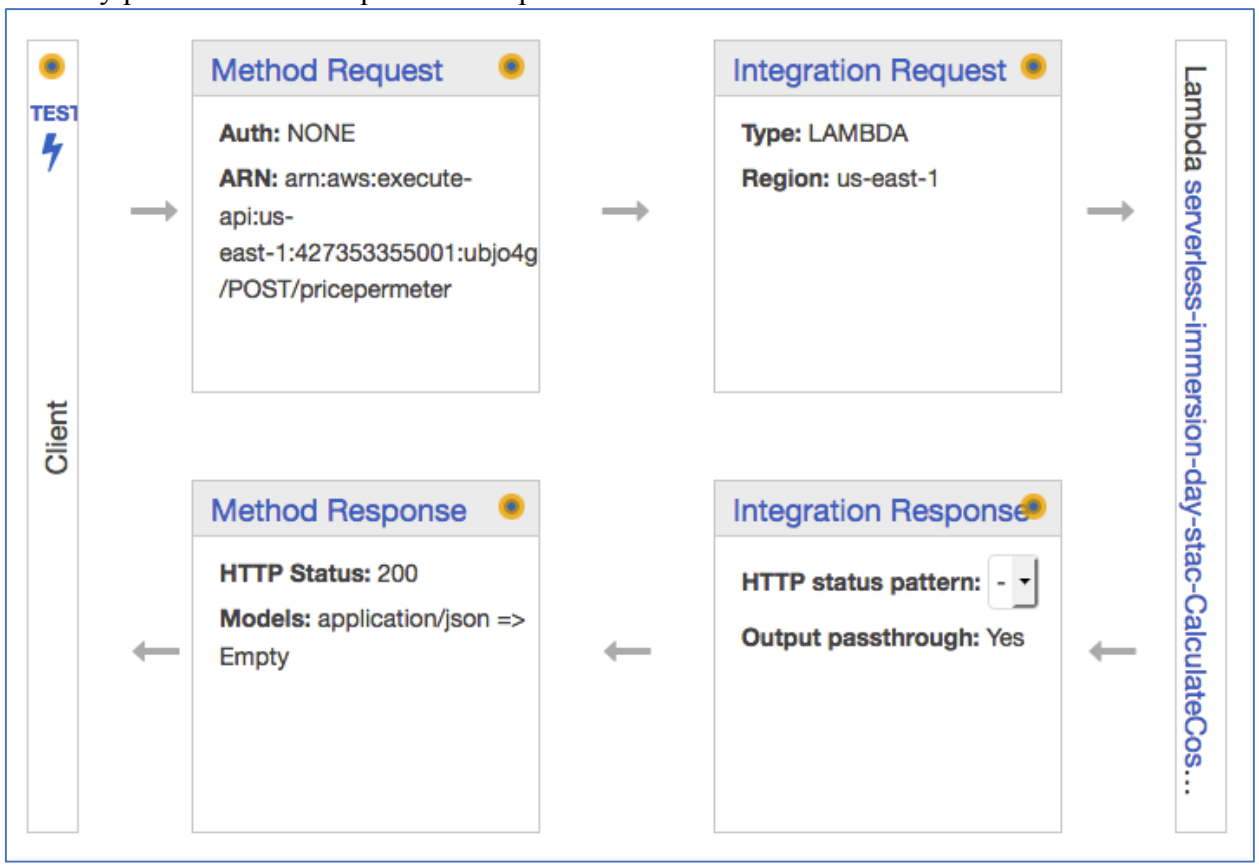


Figure 8: API Gateway console pane

15. Click on the '**Test**' icon to provide a sample request message.
16. Copy and Past the following JSON Sample in the '**Request Body**' section

```
{
 "price": "400000",
 "size": "1600",
 "unit": "sqFt",
 "downPayment" : "20"
}
```

17. } Click '**Test**'

Please note the Response Body with a "status code" of 200. The body of the response contains the price per unit and the total cost. The header of the response along with the message body is composed by the lambda function. The testing page also shows a complete request/response log.

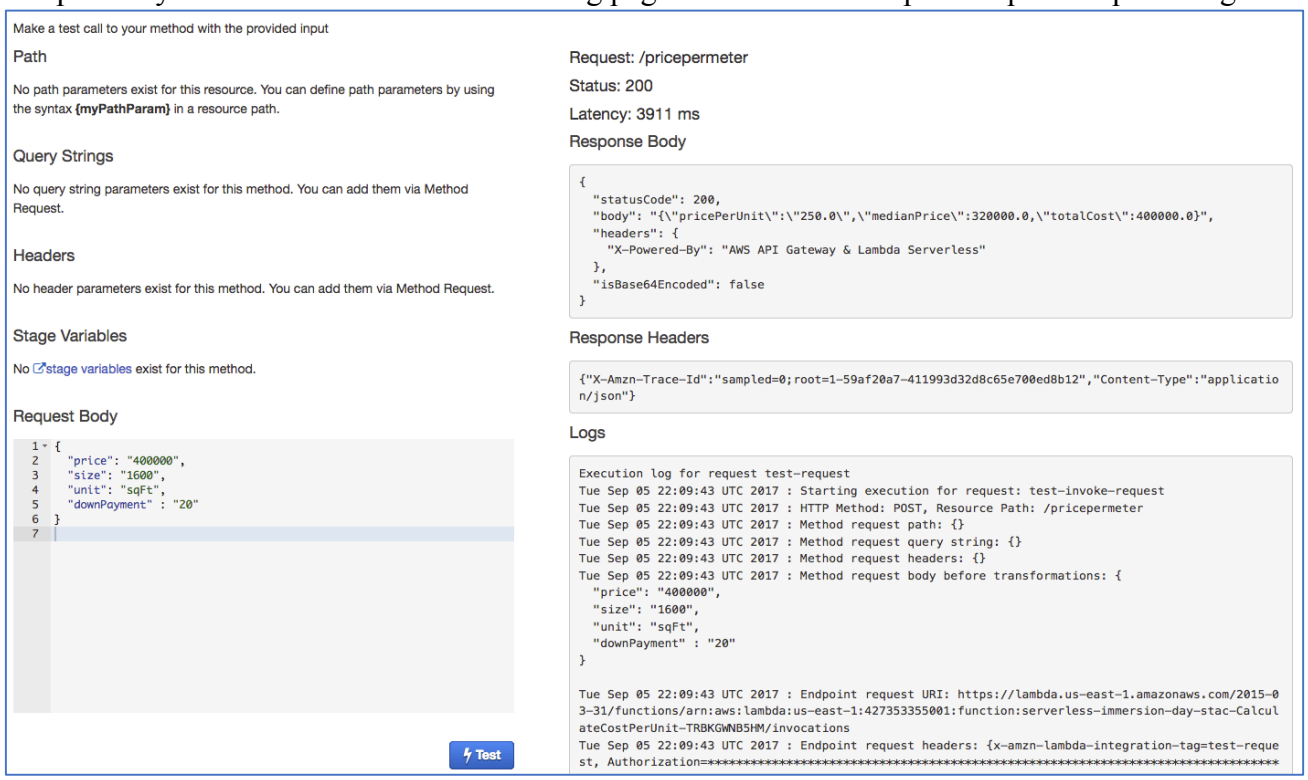


Figure 9: Testing API

Notice that the lambda service returned the total cost, but did not account for the down payment. In the next section, we will transform the message to pass the down payment amount to the lambda function.

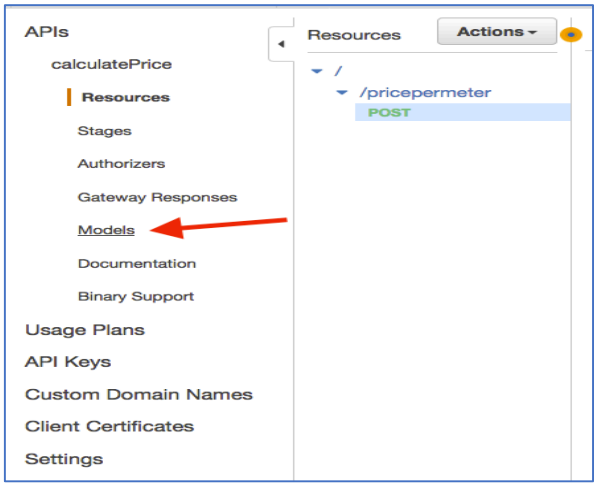
Message Transformation

In API Gateway, a mapping template is used to transform data from one format to another. A JSON path expressions can be used to map and transform the integration payload to any desired format. In addition, a model (schema) can be created to define the structure of a message payload. Having a model also enables you to generate an SDK that can be used by the client application to send properly formatted messages.

In this section, you will first create a model to represent the schema of the incoming request and response messages. Then you will validate and transform the incoming request message to match the downstream service (lambda) specification. The returned response message will also be formatted to properly capture the unit metrics.

Building a Model

1. In the box that contains the name of the API (calculatePrice), select **Models**.



2. Click '**Create**'.
3. Enter the following information:

Model name: **costCalculatorRequest**
ContentType: **application/json**
Model Description: **CostCalculator incoming request schema**
Model Schema: Copy and Paste the following:

```
{
"$schema": "http://json-schema.org/draft-04/schema#",
"title": "costCalculatorRequestModel",
"type": "object",
"properties": {
        "price": { "type": "number" },
        "size": { "type": "number" },
        "unit": { "type": "string"},
```



```
                "downPayment": { "type": "number"}
        }
    }
```

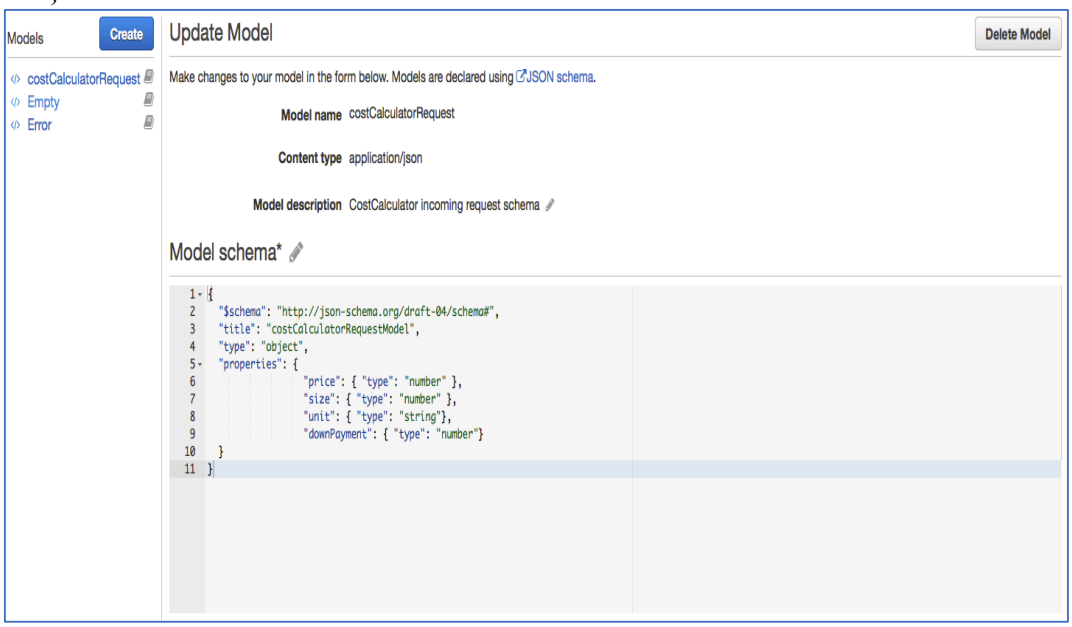


Figure 10: Updating Model

4. Click **Create model.**

Now that the models are created, the next step is to transform the incoming request and response messages.

Transform Request Payload

1. Go back to the '**CalculatePrice'** API and click on the '**POST'** action.
2. From the Gateway pane, click on '**Integration Request'**

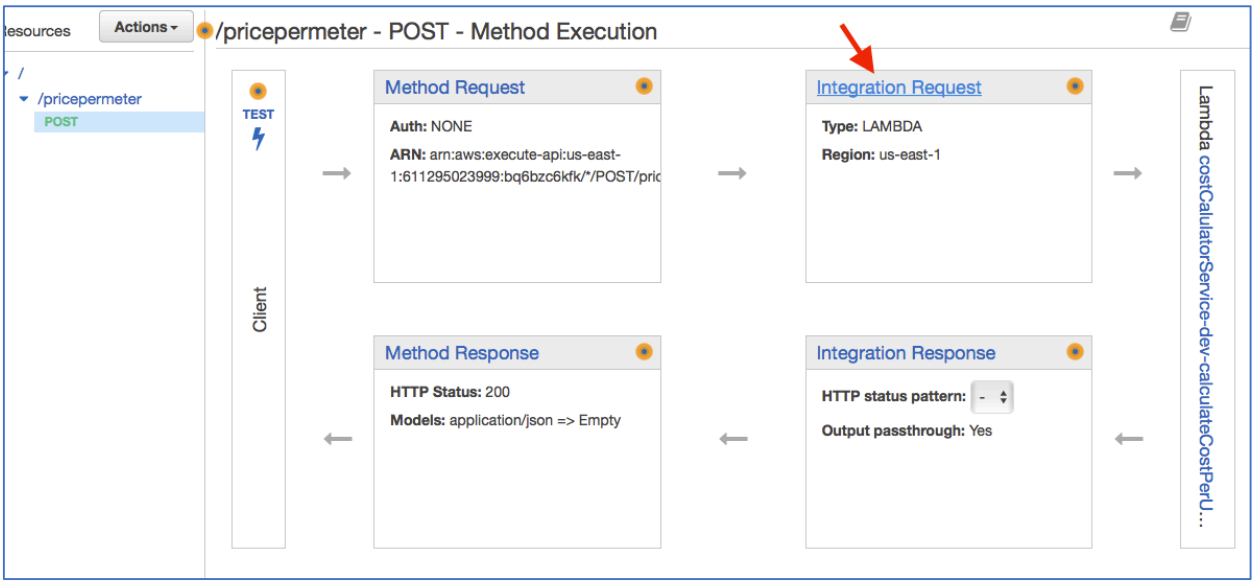


Figure 11: Integration Request

3. From the Integration Request pane, expand the '**Body Mapping Templates**'
4. Select '**When there are no templates defined (recommended)**'
5. Click on '**Add mapping template**'
6. Type **'Application/json'** for the Content-Type. Select '**check**' mark.
7. Under the '***Generate Template***', select '**CostCalculatorRequest**' model
8. Update the template as follows:

```
#set($inputRoot = $input.path('$'))
{
  "price" : "$inputRoot.price",
  "size" : "$inputRoot.size",
  "unit" : "$inputRoot.unit",
  "downPaymentAmount" : $inputRoot.downPayment
}
```

9. Click **Save**

The first statement #set($inputRoot = $input.path('$')) uses a JSONPath expression and returns an object representation of the result. This allows you to access and manipulate elements of the payload natively in Apache Velocity Template Language (VTL).

After the 'inputRoot' variable is assigned to the root of the request, we can map the value of price, size and unit into the appropriate fields. **Notice that the last statement is mapping the value of 'downPayment to a new attribute called 'downPaymentAmount'**. The downstream lambda function uses the downPaymentAmount to calculate the total price. (i.e. totalPrice = price + downPaymentAmount).

The next step is to map the integration response. An API Gateway response is identified by a response type defined by API Gateway. The response consists of an HTTP status code, a set of

additional headers that are specified by parameter mappings, and a payload that is generated by a non-VTL mapping template

Similar to the request mapping, create the mapping template for the response payload. Perform the following:

1. Go back to the '**CalculatePrice'** API and click on the '**POST'** action.
2. From the API Gateway pane, select '**Integration Response'.**
3. Click on '**Integration Response'**

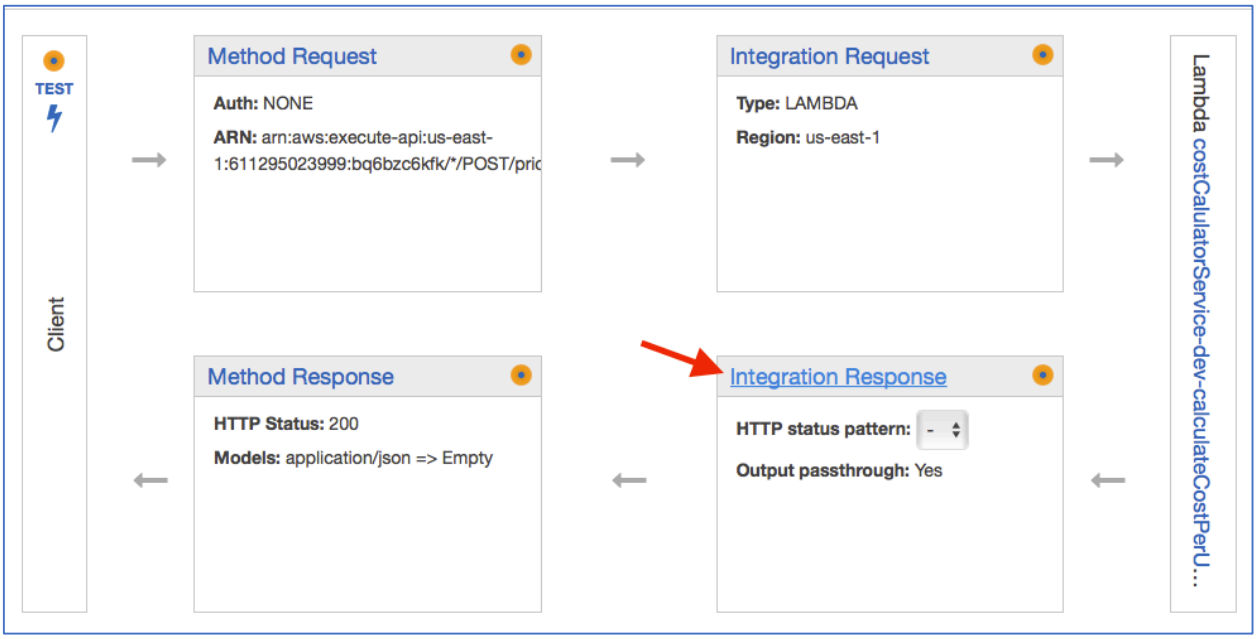


Figure 12: Integration Response

4. **Expand** the 200 Method response row
5. Expand the '**Body Mapping Templates'**
6. Click on '**application/json**' Content-Type
7. Choose ''**Method Request Passthrough**"
8. Click **Save**

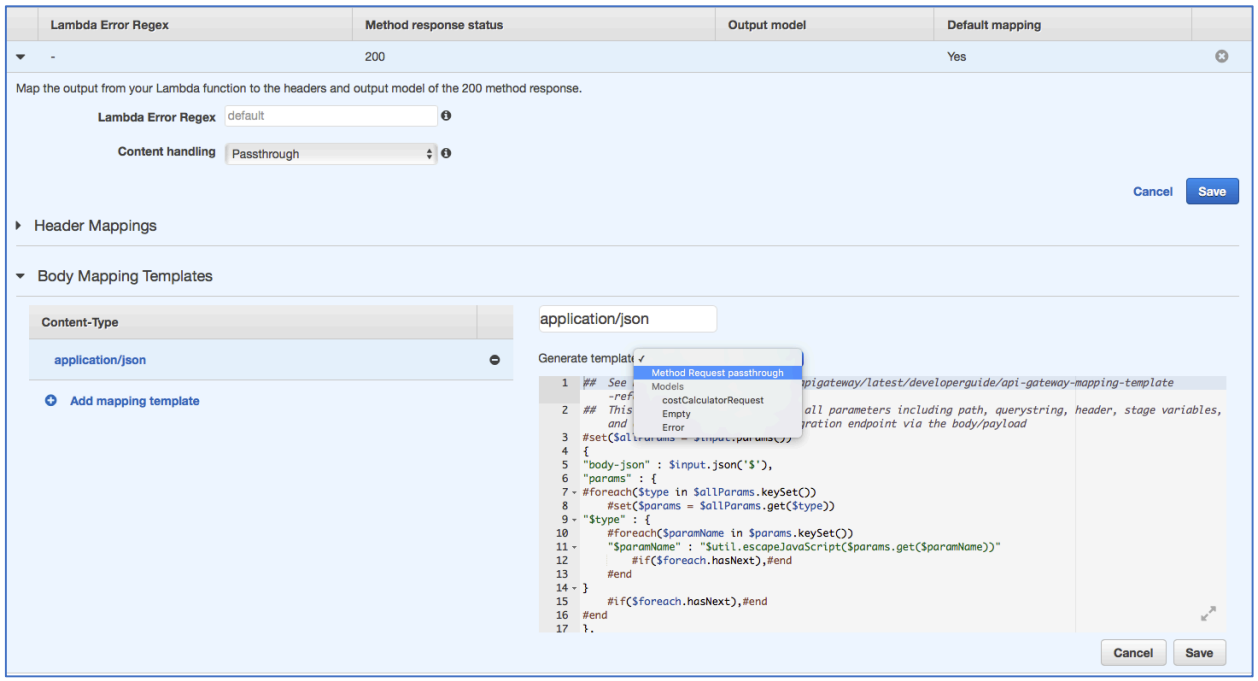


Figure 13: Method Request - Passthrough

In the response mapping, we have chosen to pass through the response content from lambda without making any further modifications. The 'method request passthrough' is a pre-defined template that maps the request header, body, parameters and context. The $context variable holds all the contextual information of your API call.

1. Go back to your 'calculatePrice' API and click on the 'POST' method to test your API.
2. Click on the '**Test**' icon to provide a sample request message.
3. Copy and Past the following JSON Sample in the '**Request Body**' section

```
{
 "price": "400000",
 "size": "1600",
 "unit": "sqFt",
 "downPayment" : "20"
}
```

4. } Click on '**Test**'

Path

No path parameters exist for this resource. You can define path parameters by using the syntax **{myPathParam}** in a resource path.

Query Strings

No query string parameters exist for this method. You can add them via Method Request.

Headers

No header parameters exist for this method. You can add them via Method Request.

Stage Variables

No stage variables exist for this method.

Request Body

```
1 {
2   "price": "400000",
3   "size": "1600",
4   "unit": "sqFt",
5   "downPayment" : "20"
6 }
7
```

Test

Request: /pricepermeter

Status: 200

Latency: 94 ms

Response Body


```
{
  "body-json": {
    "statusCode": 200,
    "body": "{\"pricePerUnit\":\"250.0\",\"totalCost\":400020.0}",
    "headers": {
      "X-Powered-By": "AWS API Gateway & Lambda"
    },
    "isBase64Encoded": false
  },
  "params": {
    "path": {},
    "querystring": {},
    "header": {}
  },
  "stage-variables": {},
  "context": {
    "account-id": "611295023999",
    "api-id": "bq6bzc6kfk",
    "api-key": "test-invoke-api-key",
    "authorizer-principal-id": "",
    "caller": "AIDAIO4AZ3P5D75JJS57K",
    "cognito-authentication-provider": "",
    "cognito-authentication-type": "",
    "cognito-identity-id": "",
    "cognito-identity-pool-id": "",
    "http-method": "POST",
    "stage": "test-invoke-stage",
    "source-ip": "test-invoke-source-ip",
    "user": "AIDAIO4AZ3P5D75JJS57K",
    "user-agent": "Apache-HttpClient/4.5.x (Java/1.8.0_131)",
    "user-arn": "arn:aws:iam::611295023999:user/mustafa",
    "request-id": "test-invoke-request",
    "resource-id": "b0m83w",
    "resource-path": "/pricepermeter"
  }
}
```


Response Headers


```
{"X-Amzn-Trace-Id":"sampled=0;root=1-5991ec49-d6c37a63916e364236c25f9a","Content-Type":"applicatio
n/json"}
```


Figure 14: Message Response

Notice that the 'downPayment' amount was added to the totalCost. The response also contains the header and context attributes.

Request Validation

Request validation is used to ensure that the incoming request message is properly formatted and contains the proper attributes. You can set up request validators in an API's Swagger definition file and then import the Swagger definitions into API Gateway. You can also set them up in the API Gateway console or by calling the API Gateway REST API, AWS CLI, or one of the AWS SDKs.

The API Gateway console lets you set up the basic request validation on a method using one of the three validators:

- **Validate body**: This is the body-only validator.
- **Validate query string parameters and headers**: This is the parameters-only validator.
- **Validate body, query string parameters, and headers**: This validator is for both body and parameters validation.

In this section, we will use the console to setup request validation and validate only the body.

1. Select '**CalculatePrice'** API and choose the 'POST' method.
2. Choose **Method Request**.

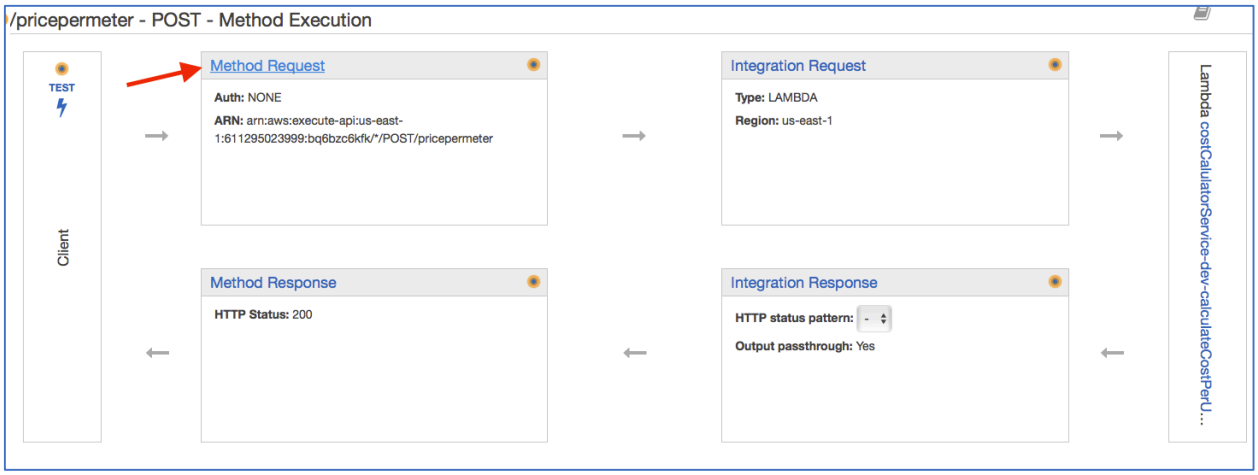


Figure 15: Method Request

3. Choose the pencil icon of **Request Validator** under **Settings**.
4. Choose `Validate body` from the **Request Validator** drop-down list and then click the check mark icon to save your choice.

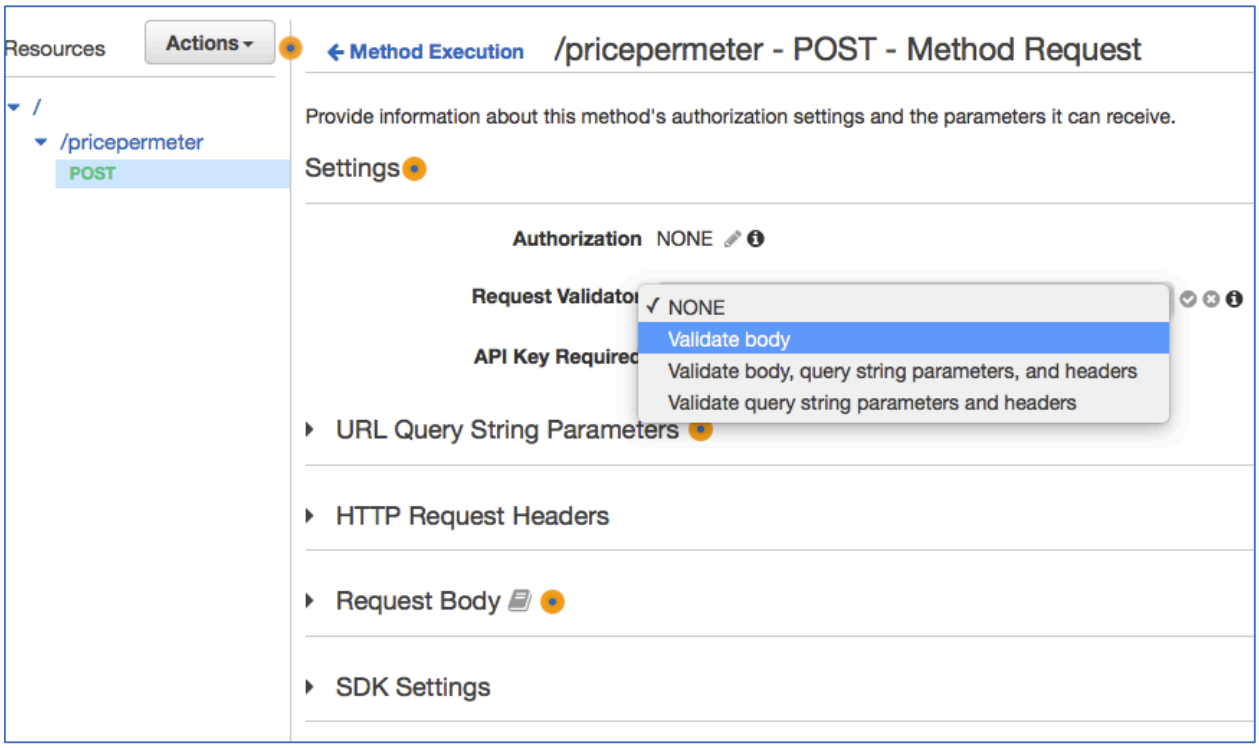
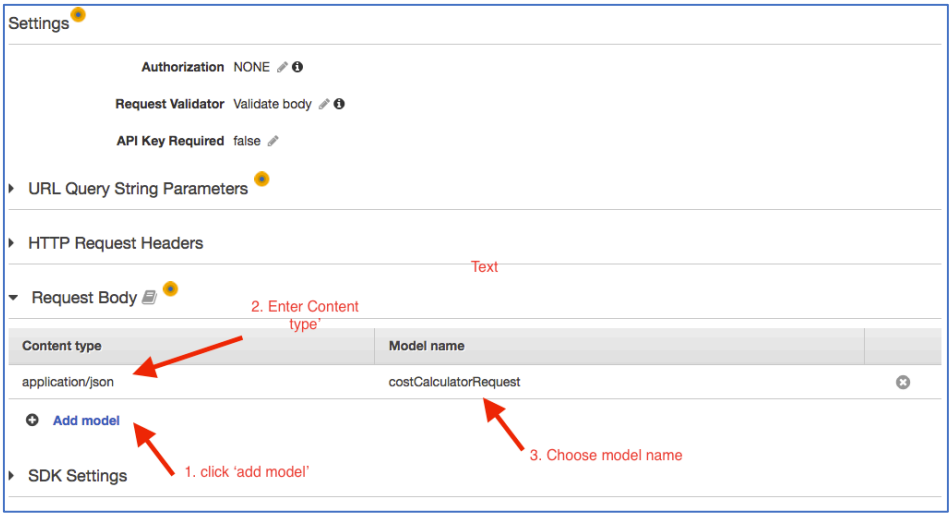


Figure 16: Message Validation

5. Expand **'Request Body'**
6. Click **'Add model'**
7. Enter '**application/json**' for Content Type
8. Choose '**costCalculatorRequest**' for the model name
9. Click the '**check**' icon to confirm.



Go back to the API Gateway console and select the 'POST' integration method to test out the API.

1. Click on the '**Test**' icon to provide a sample request message.
2. Copy and Past the following JSON Sample in the '**Request Body**' section

```
{
  "price": "400000",
  "size": "1600",
  "unit": "sqFt",
  "downPayment" : "20"
}
```

3. } Click on '**Test**'

Notice that this time, the message failed with a message of "**Invalid request body**".

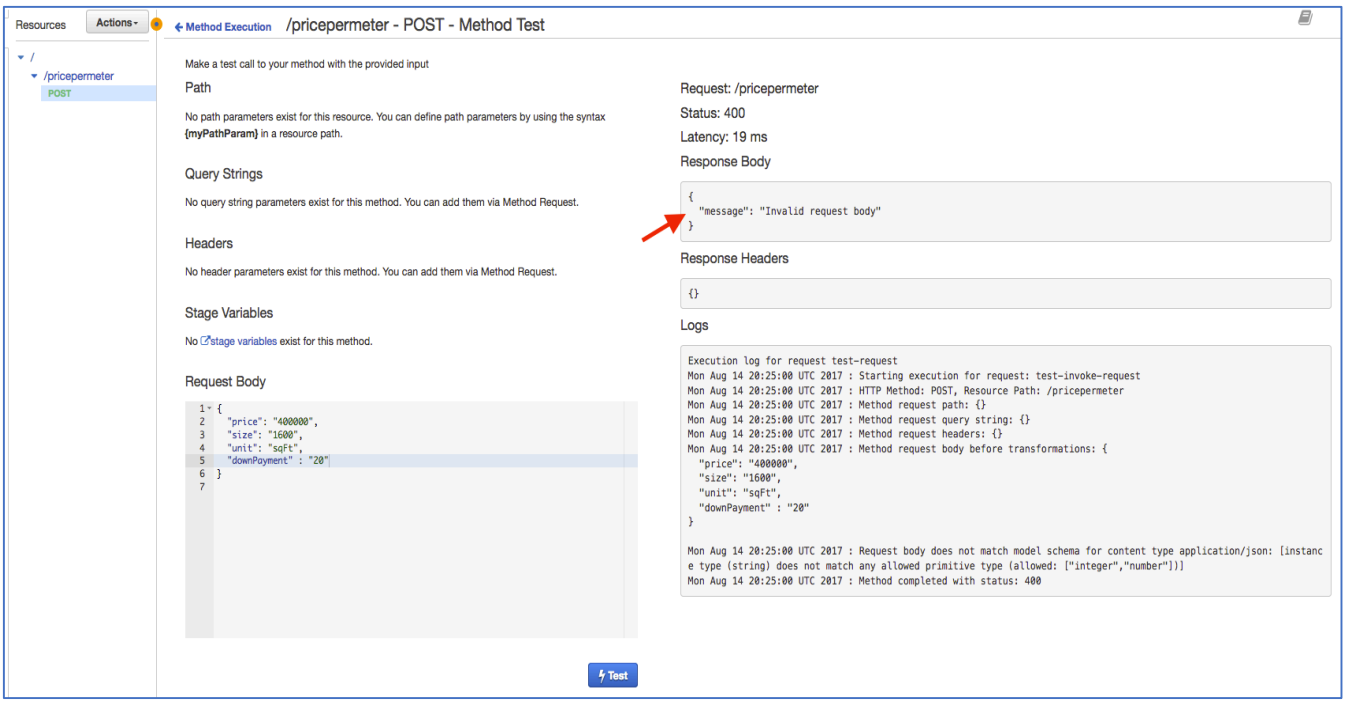


Figure 17: Failed Validation Response

The request failed validation because price, size and downPayment are defined as 'number' in our model/schema. Hence, remove the string quotes (" ") from price, size, and downPayment and try your request again.

4. Click on the '**Test**' icon to provide a sample request message.
5. Copy and Past the following JSON Sample in the '**Request Body**' section

```
{
  "price": 400000,
  "size": 1600,
  "unit": "sqFt",
  "downPayment" : 20
}
```


6. Click on '**Test**'

You should see that the normal response is returned by the service.

Authentication and Authorization

At this point, our API is wide open and we need to protect it by implementing an authentication mechanism. API Gateway supports IAM auth, Cognito User Pools, and custom authorizers written as Lambda function. In this lab, we will leverage Amazon Cognito to have only registered users use our 'costCalculator' API.

The first step is to create a Cognito User Pool. A User Pool is our user directory. We can register users in the pool and users can authenticate with their credentials. The outcome of a successful authentication against User Pools is an Open ID Connect-compatible (OIDC) identity token and a JWT access token

1. From the services menu, choose 'Cognito' or go to https://console.aws.amazon.com/cognito
2. Click on '**Manage your User Pools**'
3. Click on '**Create a User Pool**'
4. Enter '**CostCalculatorUserPool**'

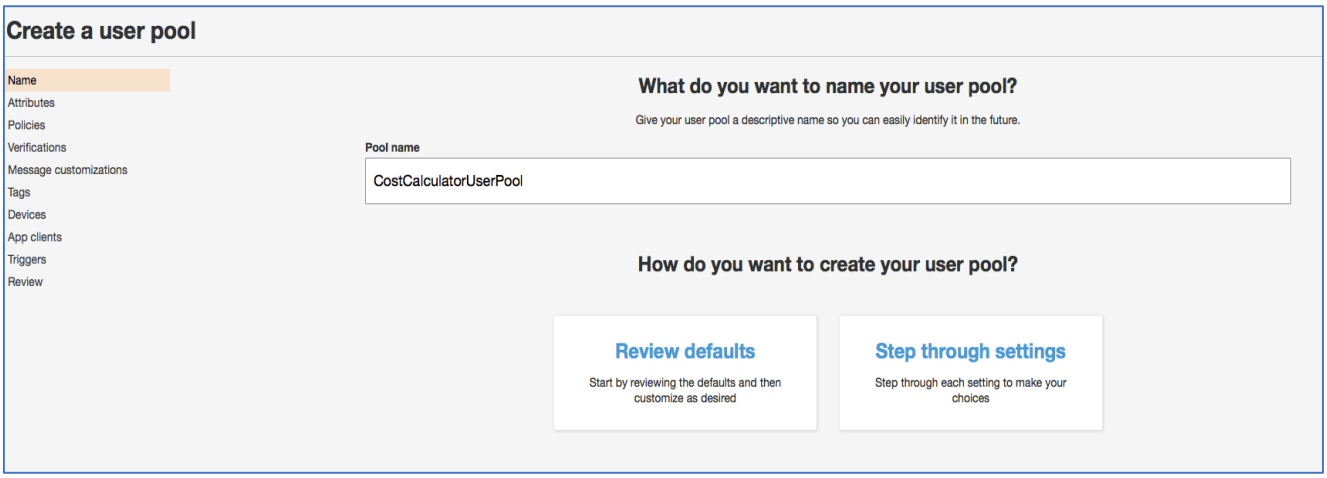


Figure 18 : Creating User Pool

5. Click '**Step through Settings**'
6. Click on the **App Clients** menu from the left navigation pane
7. Click '**Add an app Client**'
8. Enter '**Cost-Calculator-App-Client**' for the name
9. **Uncheck** Generate client Secret
10. Check '**Enable sign-in API for server-based authentication (ADMIN_NO_SRP_AUTH)**'
11. Leave all other default fields and click '**Create app client**'.

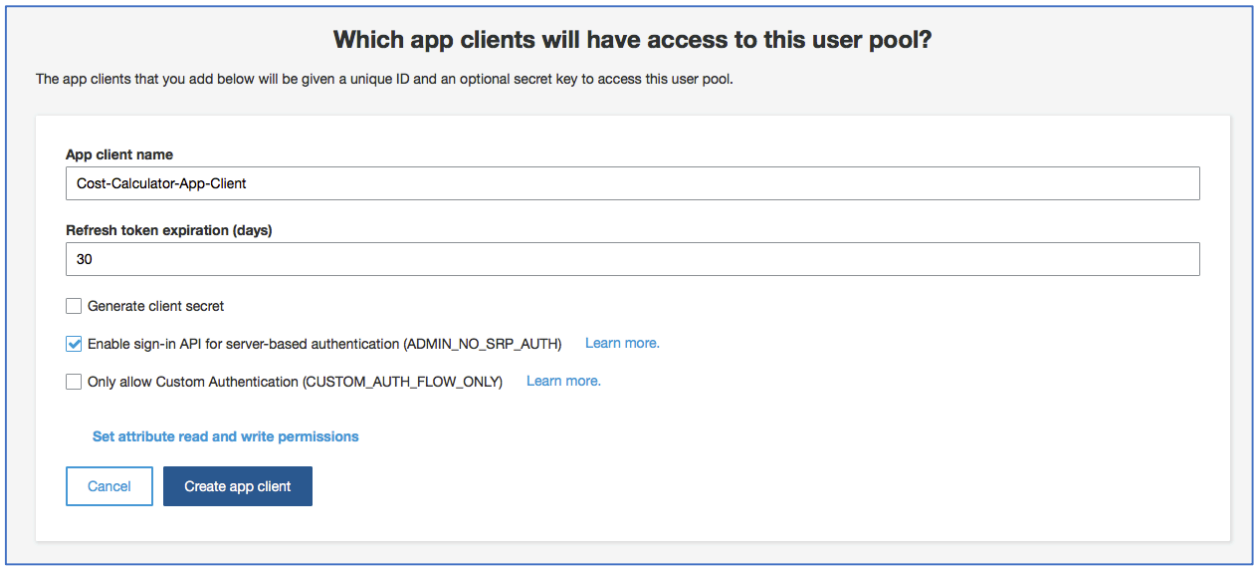


Figure 19: Client App Settings

12. Click on the '**Review**' item from the navigation pane
13. Leave all the default fields and click '**Create Pool**'

Note1: In the user pool screen, make a note of your **Pool ID**

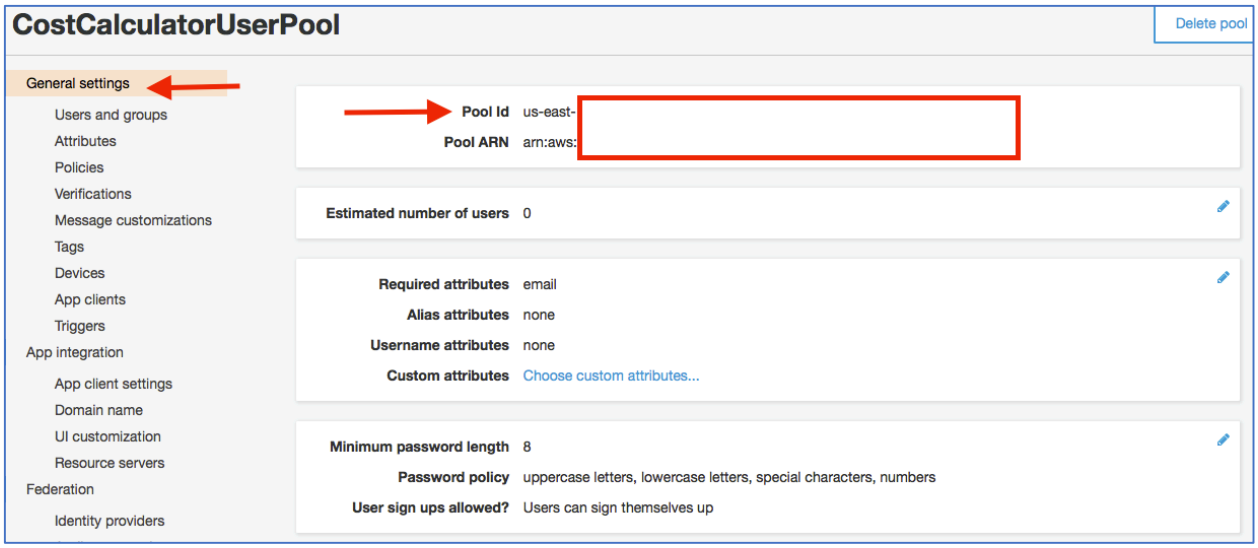
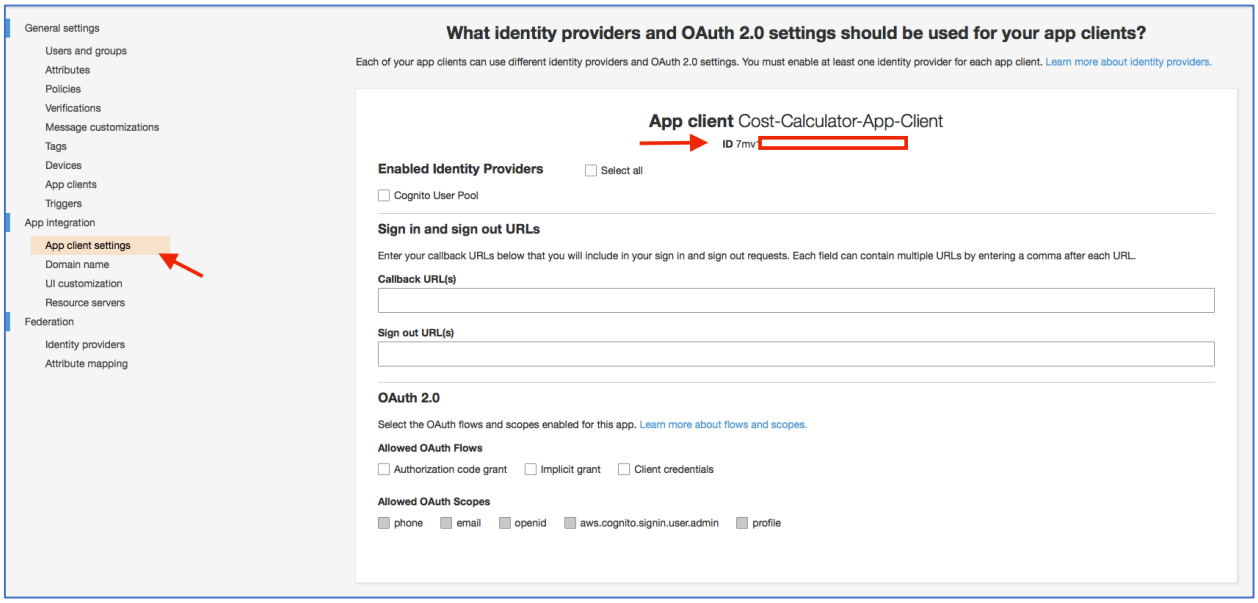


Figure 20 : General Settings Pool ID

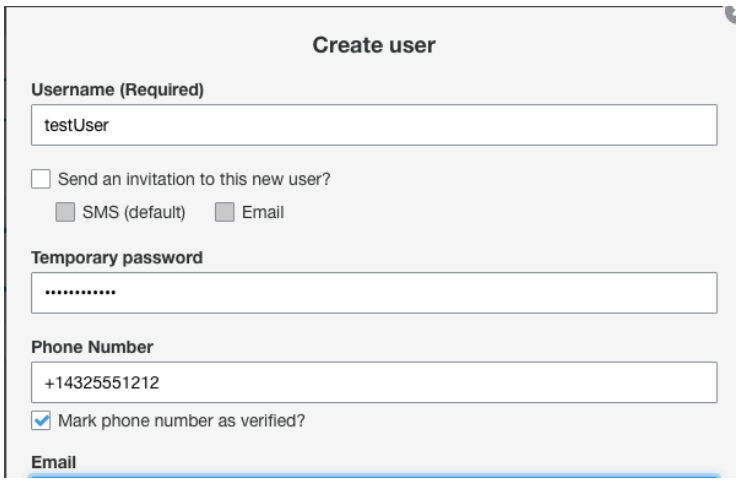
14. Click on 'App client settings'

Note 2: Make a note of your **App Client ID**



The next step is to create users that will be allowed to call our CostCalculatorAPI. From the CostCalculatorUserPool, perform the following:

1. Click on '**Users and group**' from the left navigation pane of CostCalulatorUserPool
2. Click '**Create User**'
3. Enter the following values:
   UserName: **testUser**
   Send an invitation to new user: **Uncheck**
   Temporary password: **testUser123!**
   Phone Number: enter `+14325551212`
   Email: *enter your e-mail address*
   Mark email as verified? **Check**



4. Click '**Create User**'

Now that the user pool is configured, open up a terminal and use AWS CLI to simulate a login for the user.

1. Enter the following command. Replace the pool id, client id, and username/password attributes:

```
 aws cognito-idp admin-initiate-auth --user-pool-id <YOUR POOL ID> --client-id
<YOUR CLIENT ID> --auth-flow ADMIN_NO_SRP_AUTH --auth-parameters
'USERNAME=testUser,PASSWORD="testUser123!"'
```

2. The first response to this request should be an authentication challenge, the user pool reminds you that you need to change the password for the user. If you recall, when we created the user it asked us for a temporary password, not the final one

```
  "ChallengeName": "NEW_PASSWORD_REQUIRED",
  "ChallengeParameters": {
    "USER_ID_FOR_SRP": "testUser",
  "requiredAttributes": "[]",
     "userAttributes": "{\"email_verified\":\"true\",\"email\":\"usereMail\"}"
},
"Session": "..."
}
```

3. Use the CLI to set the final password for the user. Replace the pool ID, client ID, username/pwd and the session attributes. The session value can be retrieved from the response to step 2. Ensure your final passwords contains lower case, upper case, number and character symbols.

```
aws cognito-idp admin-respond-to-auth-challenge --user-pool-id <YOUR POOL ID>
--client-id <YOUR CLIENT ID> --challenge-name NEW_PASSWORD_REQUIRED --
challenge-response 'USERNAME=testUser,NEW_PASSWORD=<FINAL PASSWORD>' --session
'<SESSION VALUE FROM PREVIOUS RESPONSE>'
```

4. The output from this call should be a valid session

```
{
    "AuthenticationResult": {
        "ExpiresIn": 3600,
        "IdToken": "...",
        "RefreshToken": "...",
        "TokenType": "Bearer",
        "AccessToken": "..."
    },
    "ChallengeParameters": {}
```



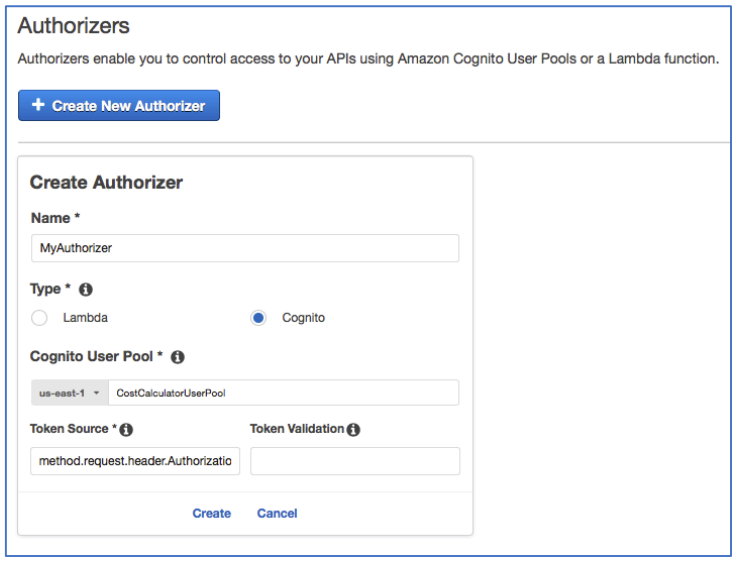
```
}
```

The response contains three important parameters. The IdToken is the Open ID Connect-compatible identity token that API Gateway uses to authenticate calls - this is the token we will pass to the APIs in the Authorization header. The AccessToken is a JWT token that contains the user scopes and identity pool information. The last property is a refresh token, you can use this token with the User Pool APIs to fetch a new identity and access token.

Note 3: If needed, you get a new token for a user by using the following AWS CLI command:

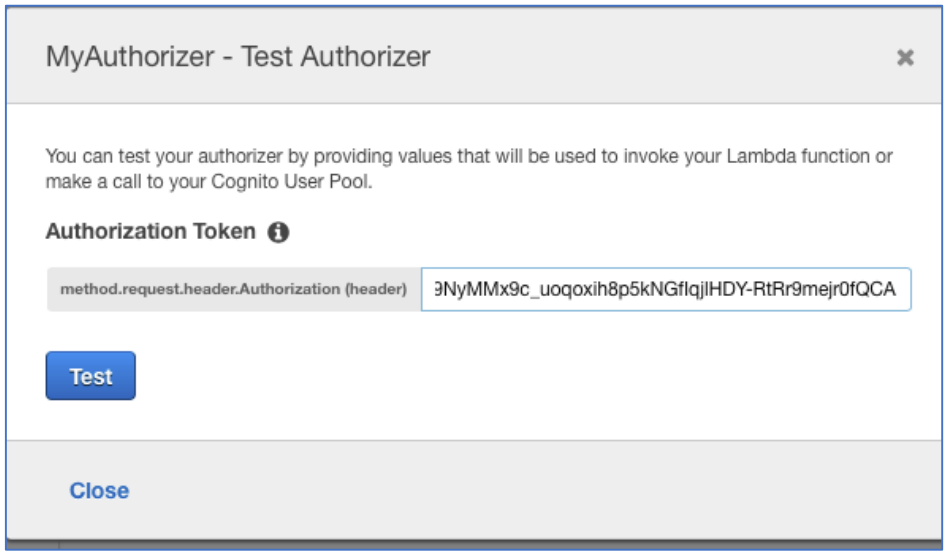
```
aws cognito-idp admin-initiate-auth --user-pool-id <YOUR POOL ID> --client-id <YOUR CLIENT ID>
--auth-flow ADMIN_NO_SRP_AUTH --auth-parameters 'USERNAME=testUser,PASSWORD="<YOUR NEW
PASSWORD>"'
```

1. Go back to API Gateway Service or go to https://console.aws.amazon.com/apigateway
2. Click on '**calculatePrice**' API
3. Click '**Create New Authorizers**'
4. Name: "**MyAuthorizer**"



5. Enter the following values:
   Cognito Region: *choose your region (us-east-1)*
   Cognito user Pool: '**CostCalculatorUserPool**'
   Token Source: '**method.request.header.Authorization**'

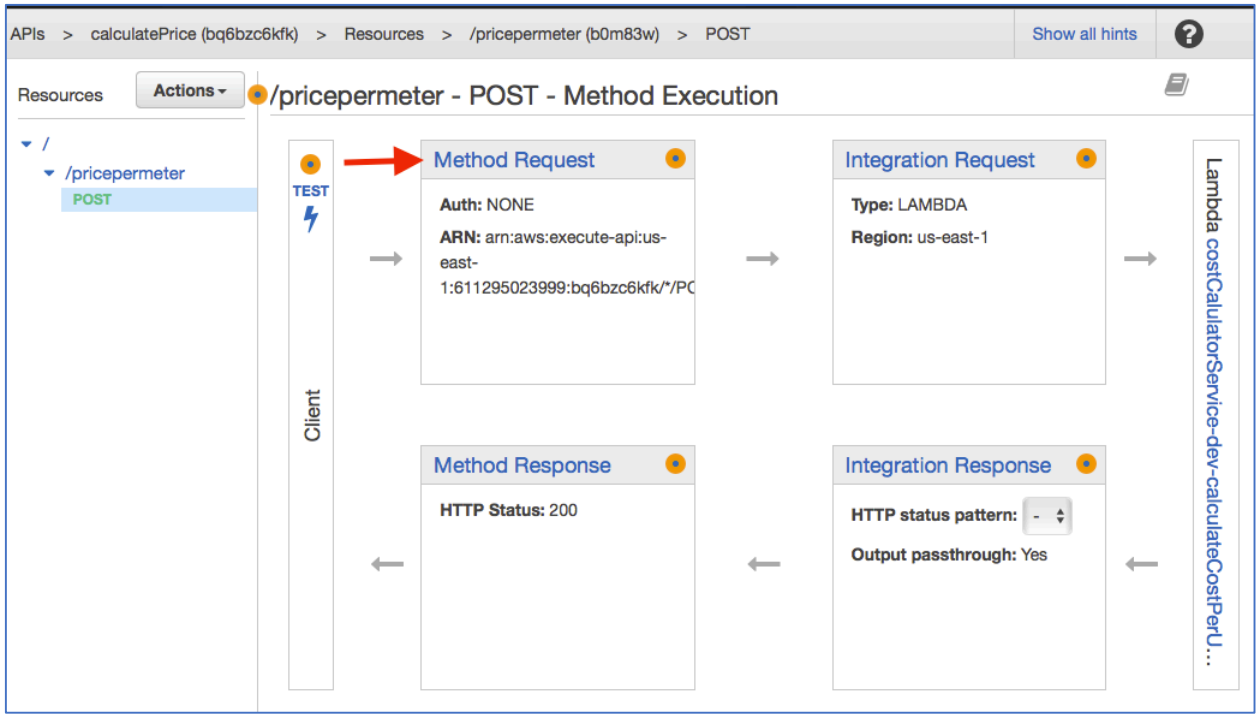
6. Click '**Create**'
7. Test your Authorizer by clicking **Test**
8. Enter the returned **Identity token** for authorization value



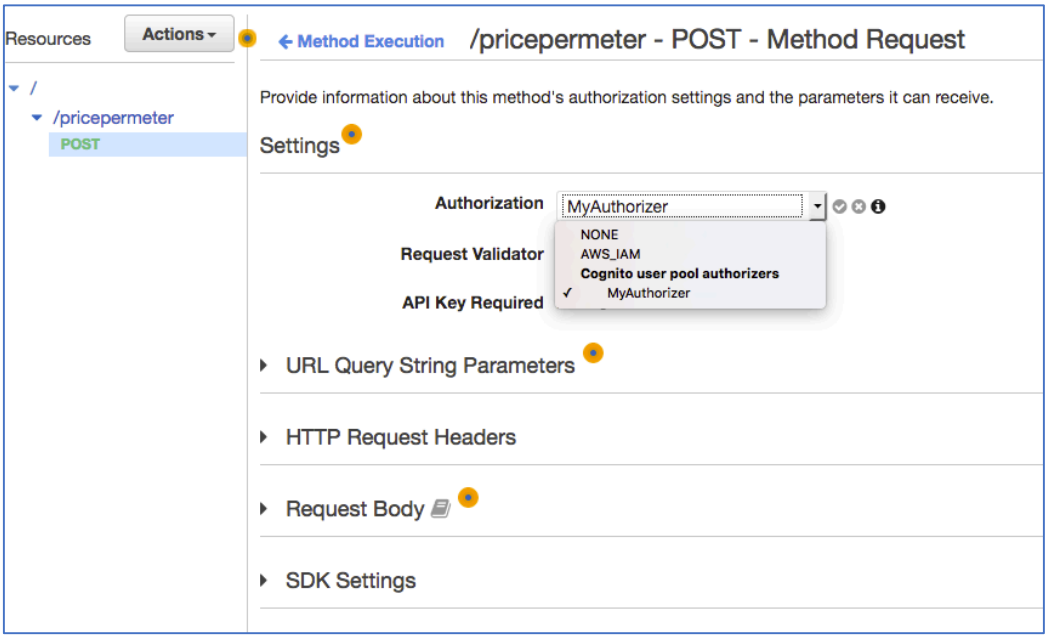
You should see response code 200 with appropriate claims.

Go back to the 'CalculatePrice' API and click on the POST action. Set the Method Request Authorization for the API.

1. From the API Gateway Console, click on '**Method Request**'



2. Set the Authorization to your Cognito Authorizer (i.e. MyAuthorizer).

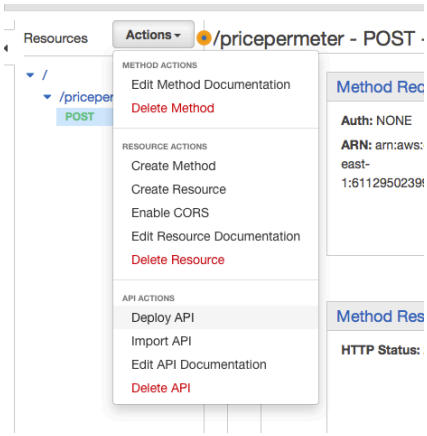


3. Click the '**check**' mark icon to confirm the Authorization.

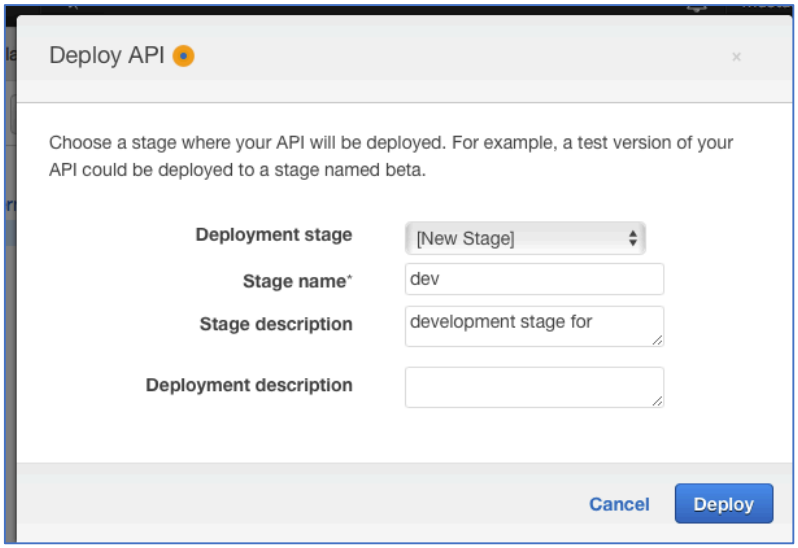
API Deployment

At this point, we have been testing our API using the API Gateway console. Now that we have made the necessary configuration for authorization, we are ready to deploy our API .

1. Select **POST method** from the API Gateway console for the 'calculatePrice' API
2. Select "**Action**" then choose '**Deploy API**'



3. Create a new Stage. Enter the following values:
   Stage Name: dev
   Stage description: development stage for calculate price API
   Deployment

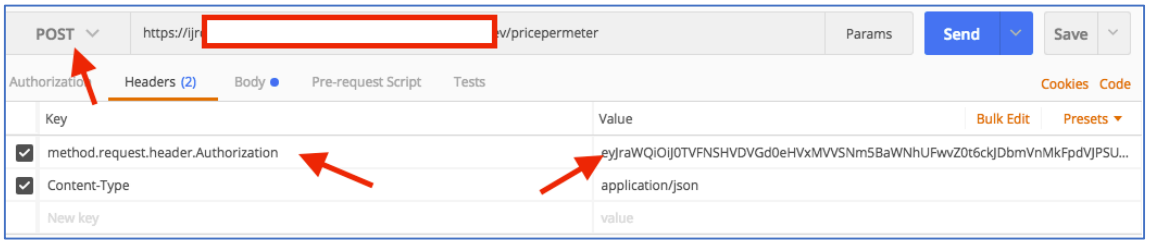


4. Click **'Deploy'**
5. Expand the '**dev**' stage, and click on the 'POST' method of the pricepermeter.
   Note 4 the API/Invoke URL. Copy this value as it will be used in the subsequent steps.

Up to this point, we have used the local API Gateway console to test our API. Now that we have deployed the API, we can test it using external tools. For this lab, we will leverage PostMan to test our API.

1. Insert the service URL into postman
2. Change the method type to POST

3. Click **Headers**. Insert the following attributes
   Key: **method.request.header.Authorization**
   Value: <Your IdToken>

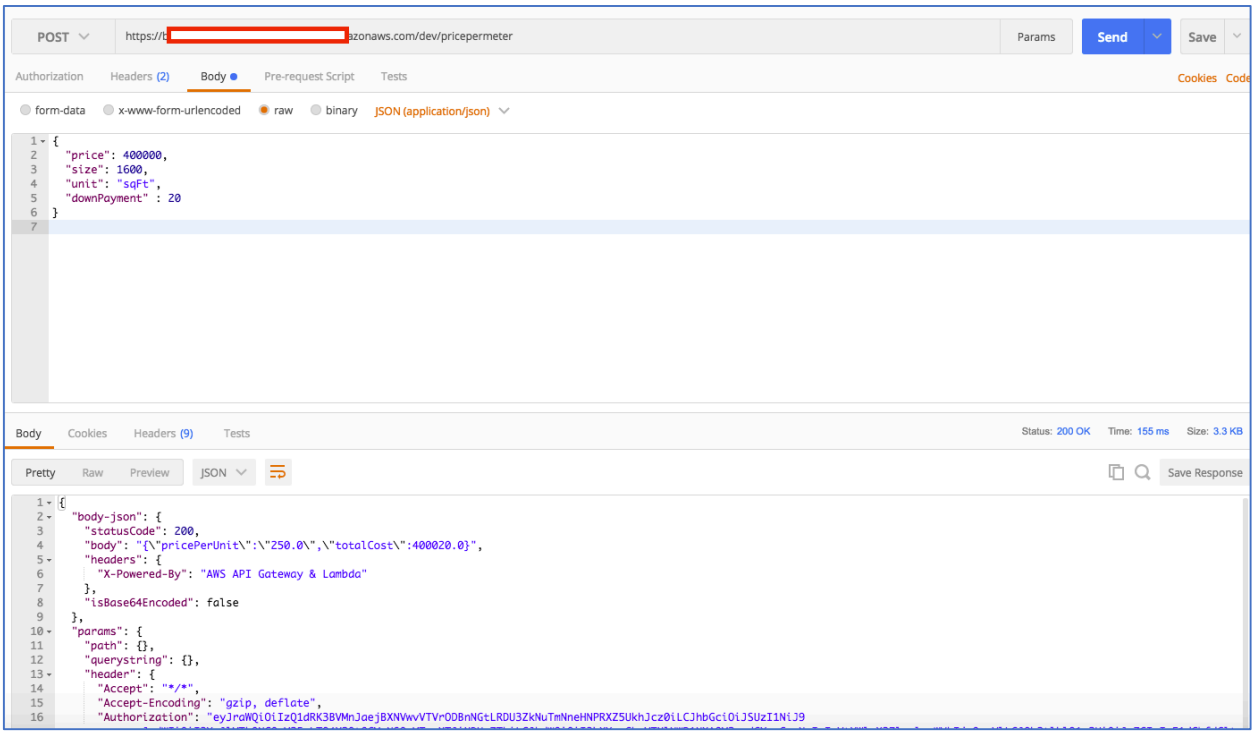


4. Click **Body**
5. Choose 'raw' type
6. Change Body type to '**JSON (application/json)**'
7. Insert the following request

```
{
 "price": 400000,
 "size": 1600,
 "unit": "sqFt",
 "downPayment" : 20
}
```

8. Click '**Send**'

Note 5: If you get "message": "Unauthorized", it maybe that your ID token has expired. See note 3 above to refresh your token.



Congratulations, you have just deployed your first API !

Lab 1 Summary

In lab 1, you created your first API using AWS API Gateway. In this lab, you configured API Gateway to call a lambda service, transformed and validated the request message, and secured the API. In the next lab, you will learn additional features of API Gateway.

Lab 2: Additional API Gateway Features

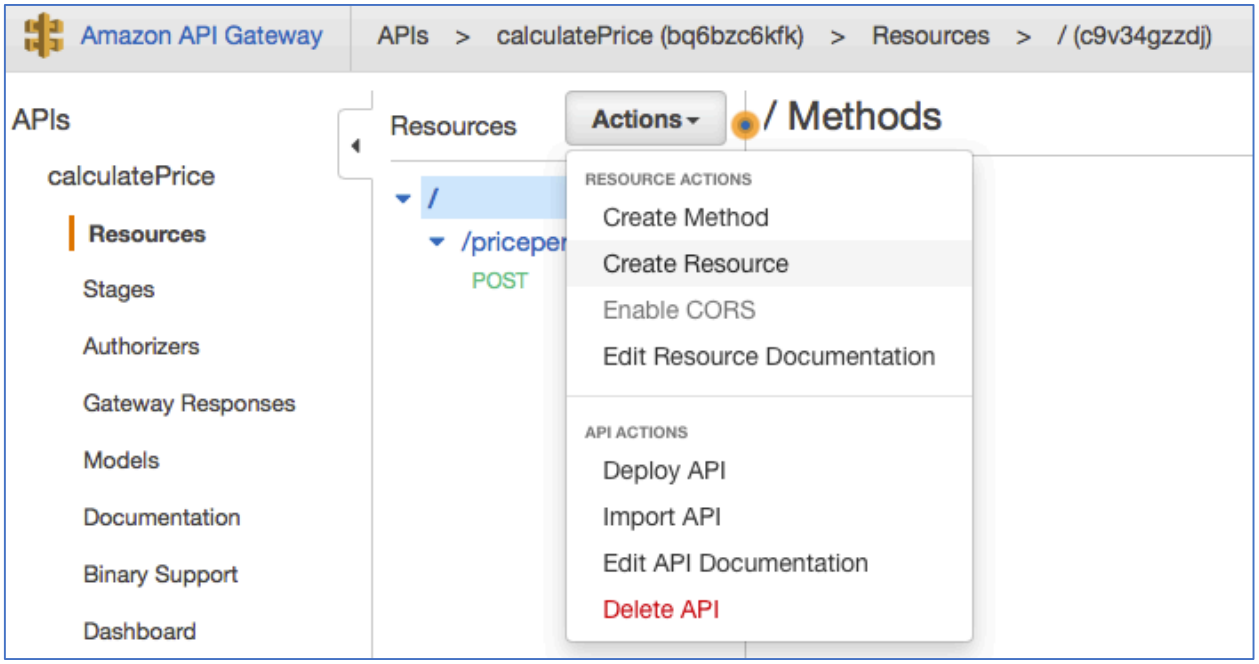
In the second part of the lab, we will create a new API and configure the API to perform message caching and throttling. In addition, we will configure usage plans for our platinum, silver and gold customers.

Message Caching

You can enable API caching in Amazon API Gateway to cache your endpoint's response. With caching, you can reduce the number of calls made to your endpoint and also improve the latency of the requests to your API. When you enable caching for a stage, API Gateway caches responses from your endpoint for a specified time-to-live (TTL) period, in seconds. API Gateway then responds to the request by looking up the endpoint response from the cache instead of making a request to your endpoint.

API Gateway enables caching at the stage or method level. In this lab, we will create a new resource called 'medianPriceCalculator'. This service returns the median price of houses in US or Canada. Since these regional prices do not change often, we can cache the results.

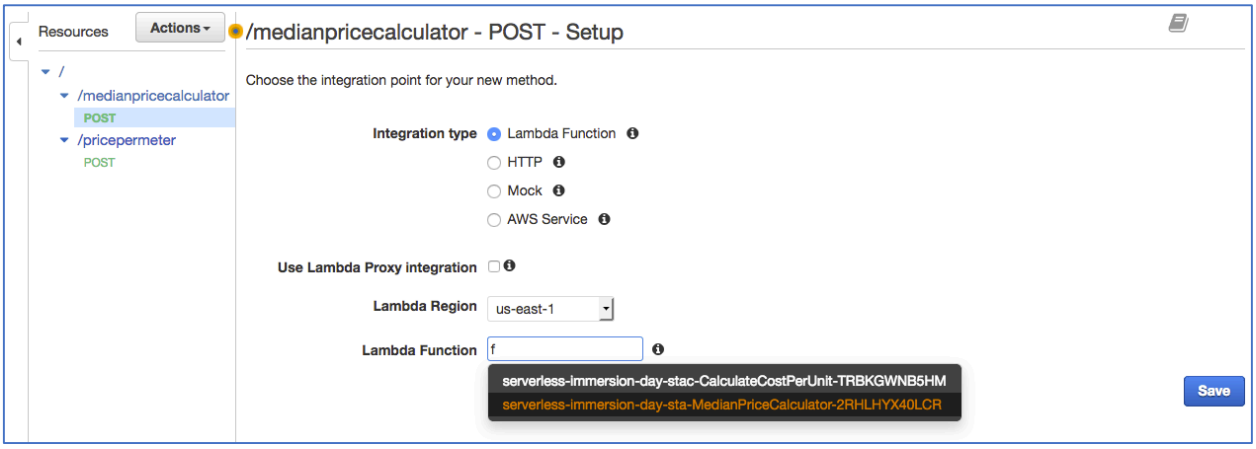
1. From API Gateway Console, select the root element of **CalculatePrice** API
2. Click **Actions**, then choose '**Create Resource**'



3. Enter the following values
   Resource Name: '**medianPriceCalculator**'
   Resource Path: '**medianPriceCalculator**'

4. Click '**Create Resource**'

5. Click on **Actions**, then Choose '**Create Method**'.
6. Choose '**POST**' and select **check mark.**
7. For the Post Setup, choose the following values:
   Integration Type**: Lambda Function**
   Lambda Region: **us-east-1 (or your chosen region)**
   Lambda Function: **serverless-immersion-day-sta-MedianPriceCalculator-XX**

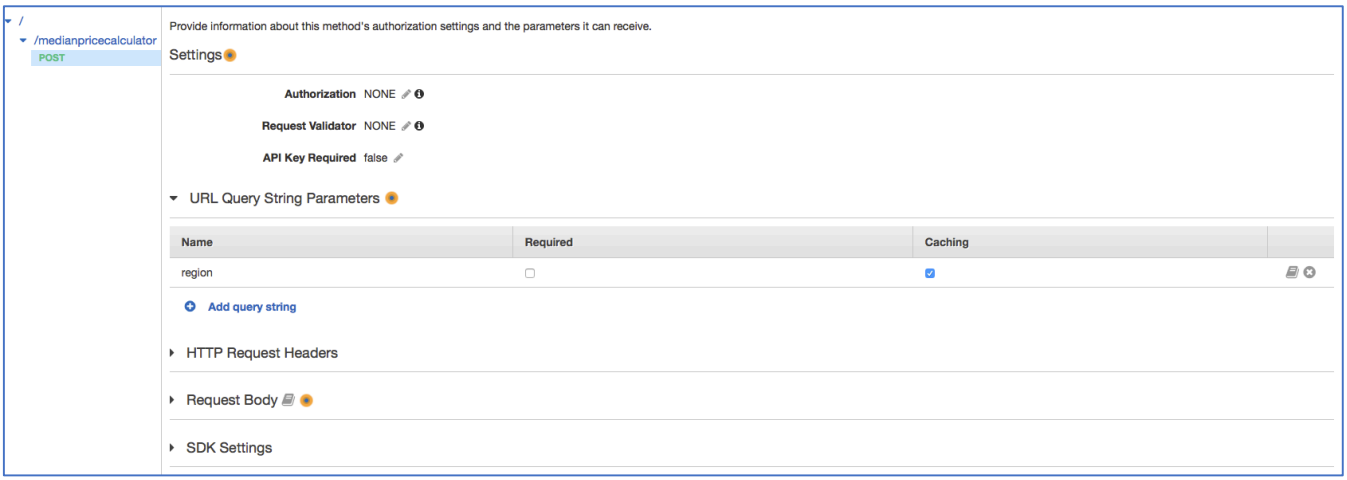


8. Click '**Save**', then '**Ok**'.

Now that the API is created, we can enable caching at the method level. Since the median price for Canada and US are different, it would make sense to cache based on the chosen region. When a cached method or integration has parameters, which can take the form of custom headers, URL paths, or query strings, you can use some or all of the parameters to form cache keys. API Gateway can cache the method's responses, depending on the parameter values used.

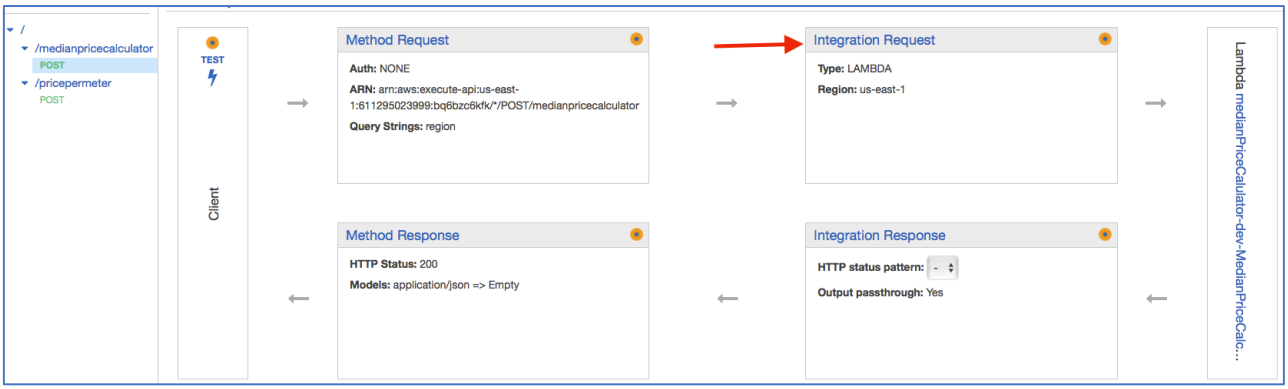
In this example, we will create a query parameter to cache the results based on the region attribute. The 'region' can take a value of 'US' or 'CA', United States or Canada respectively.

1. Click the '**POST**' method of the medianpriceCalculator API.
2. From the API Console, click on '**Method Request**'
3. Expand the '**URL Query String Parameters**'.
4. Click '**Add query String**'
5. Enter '**region**' for the name
6. Click on '**Check**' icon
7. Check '**Caching**'

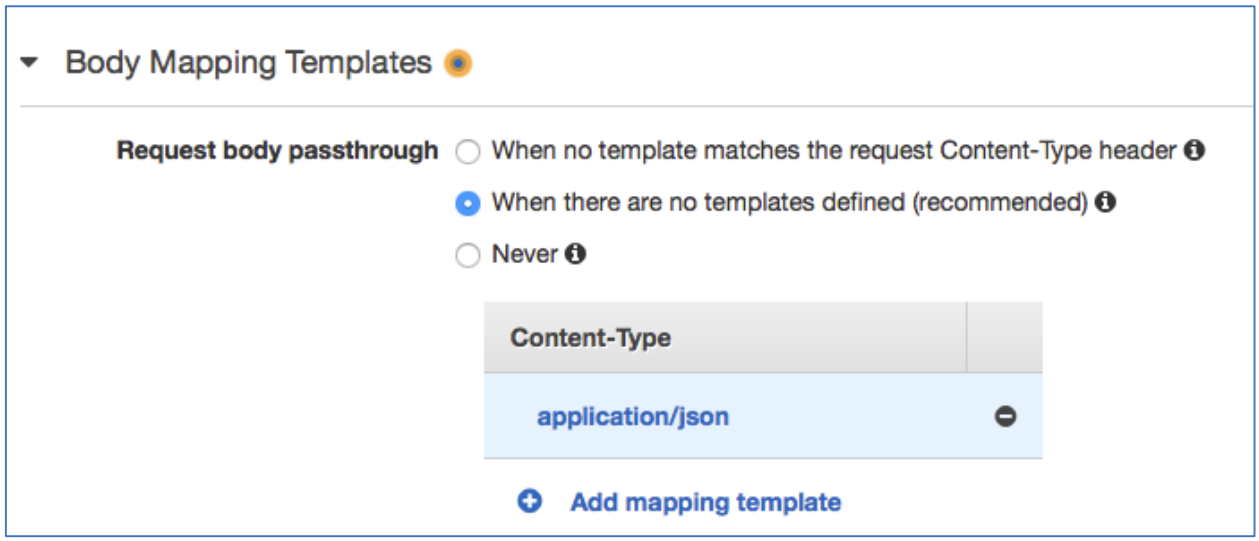


Now that we are passing the 'region' as a query parameter, we need to map it to the integration payload.

1. Click on '**medianpricecalculator'** and then click on '**POST'** action.
2. Click on '**Integration Request'** from the API Console

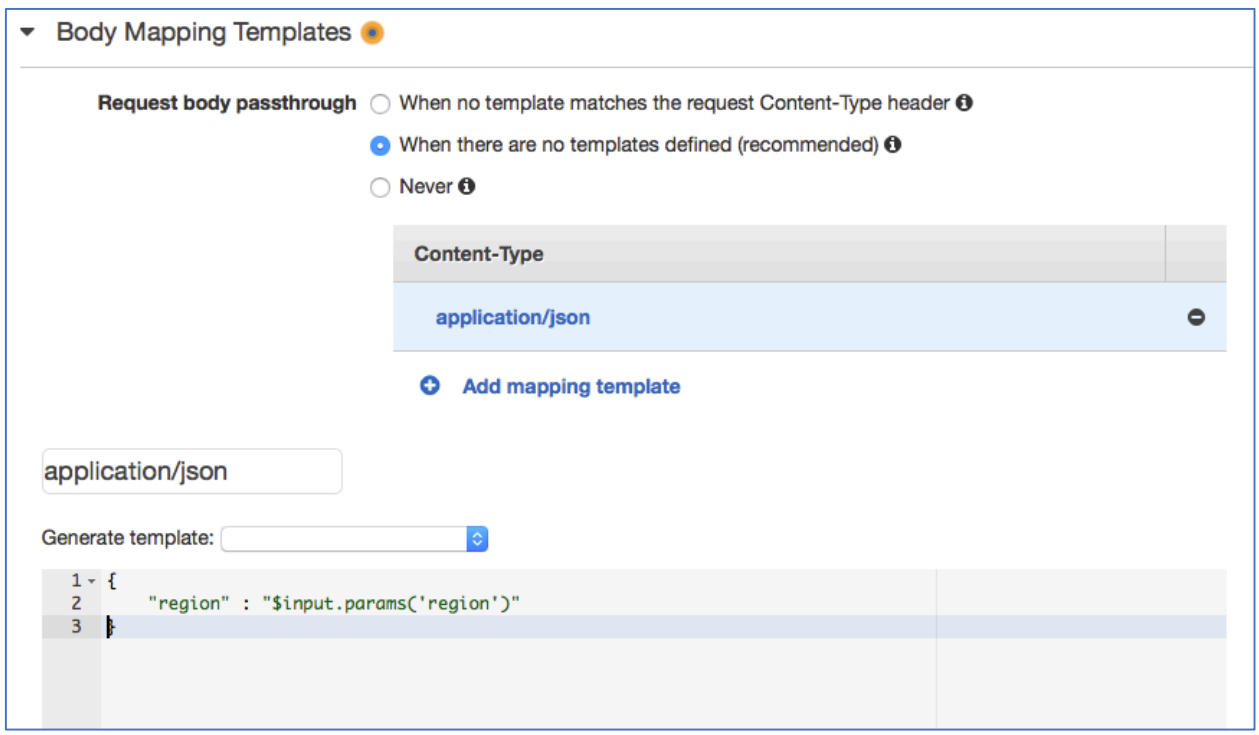


3. Expand '**Body Mapping Templates'**
4. Click on '**When there are no templates defined (recommended)'**
5. Click '**Add mapping template'**
6. Enter '**application/json'** for content-Type
7. Click on **Check mark**



8. Click on '**application/json**' content type.
9. Enter the following mapping for the 'application/json' template

```
{
   "region" : "$input.params('region')"
}
```



10. Click '**Save**'

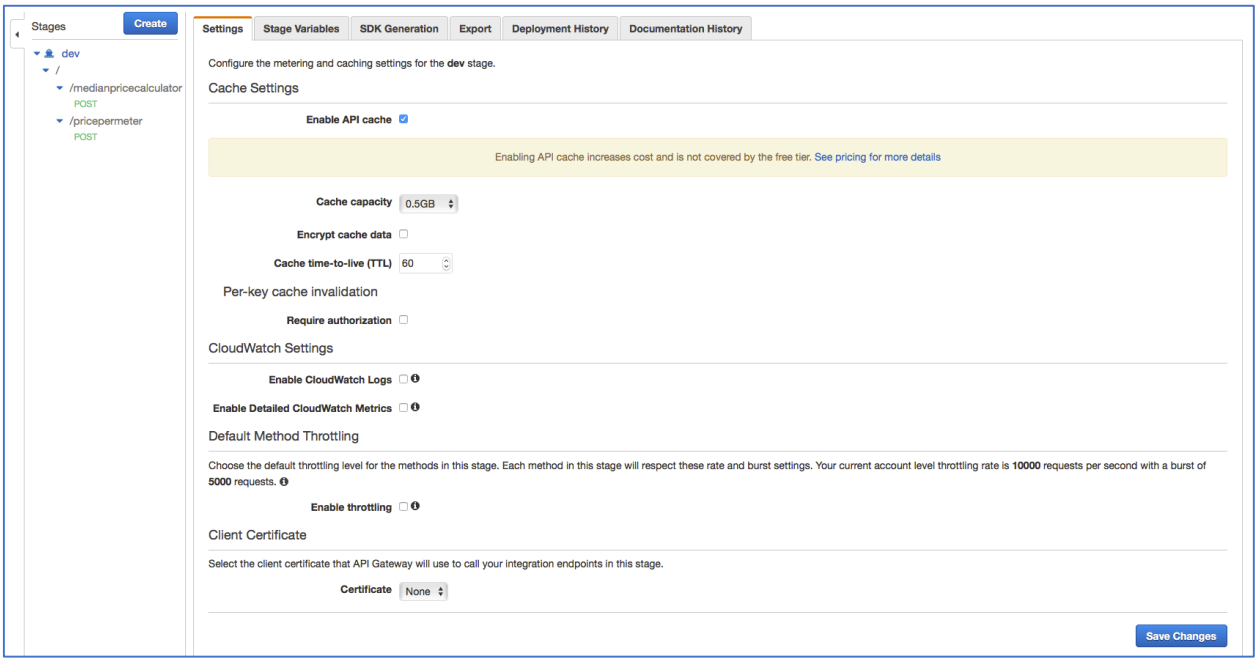
At this point, we are essentially taking the 'region' value from the query parameter and mapping it into the request.

Now, its time to deploy our new API.

1. Click on '**POST**' method of '**medianpricecalculator**'
2. Click on **Actions**
3. Click on '**Deploy API**'
4. Choose '**dev**' stage
5. Enter optional description
6. Click '**Deploy**'

Now that the method is deployed, we can go ahead and enable caching.

1. Click '**Stages**' under the 'CalcualtePrice' API
2. Click on the '**dev**' stage
3. Check '**Enable API Cache**'
4. Choose Cache capacity: **0.5** GB
5. Enter Cache time-to-live (TTL): **60**
6. Leave all other fields as default



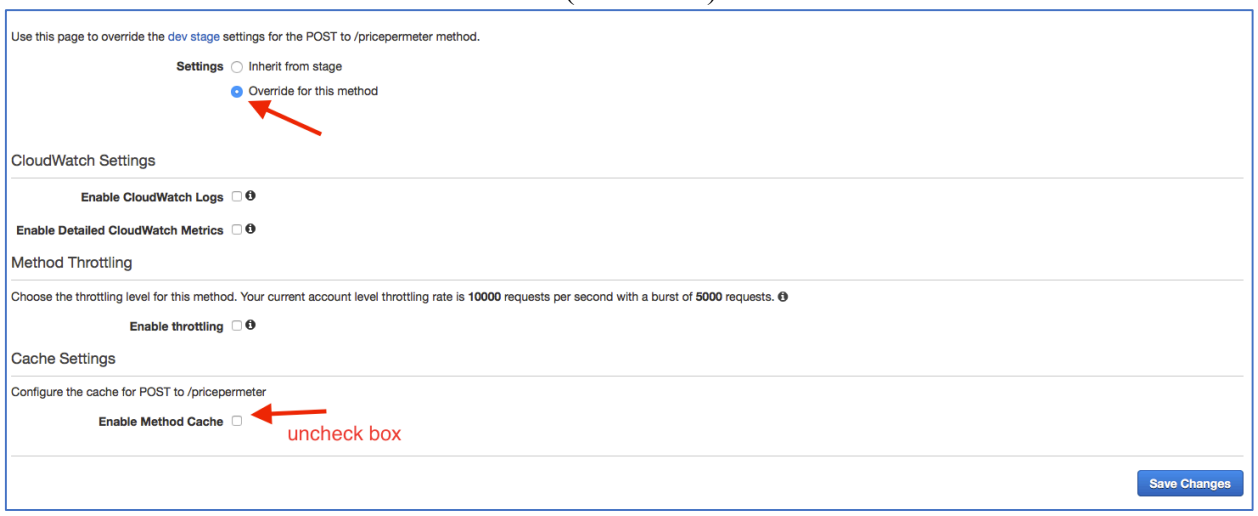
7. Click Save Changes

Notice that the Cache Status is 'CREATE_IN_PROGRESS'. Once the cache state changes to '**Available**', caching will be enabled.

Note 6: You may have to wait until cache is AVAILABLE prior to proceeding.

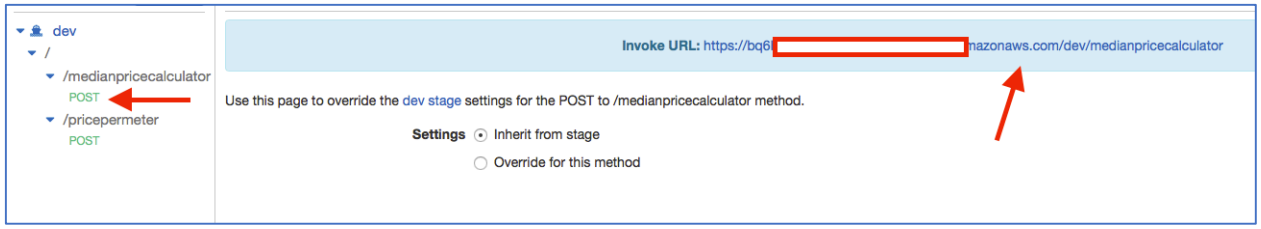
Since we enabled caching at the stage, it will automatically apply to all the resources deployed under the stage. However, we only want to cache the newly created ***medianpricecalculator*** resource. It would not make sense to cache the pricepermeter service. Hence, we have to override the *dev* stage settings in pricepermeter.

1. From the '**dev**' stage, click on '**pricepermeter**'
2. Click on the '**POST**' method
3. Select '**Override for the method**'
4. **Uncheck** the Enable Method Cache (if checked)



5. Click '**Save Changes**'

Go back to the **medianpricecalculator** POST method from the 'dev' stage and Copy the Invoke URL.



Testing Your Cached Resource

Now that we have created our second API, it is time to test it. Notice that the medianPriceCalculator service gives the median house price for two US regions (US and Canada). Since we are caching based on the 'region' parameter, the responses from '?region = 'US' will be cached separately from the response from ?region='CA'.

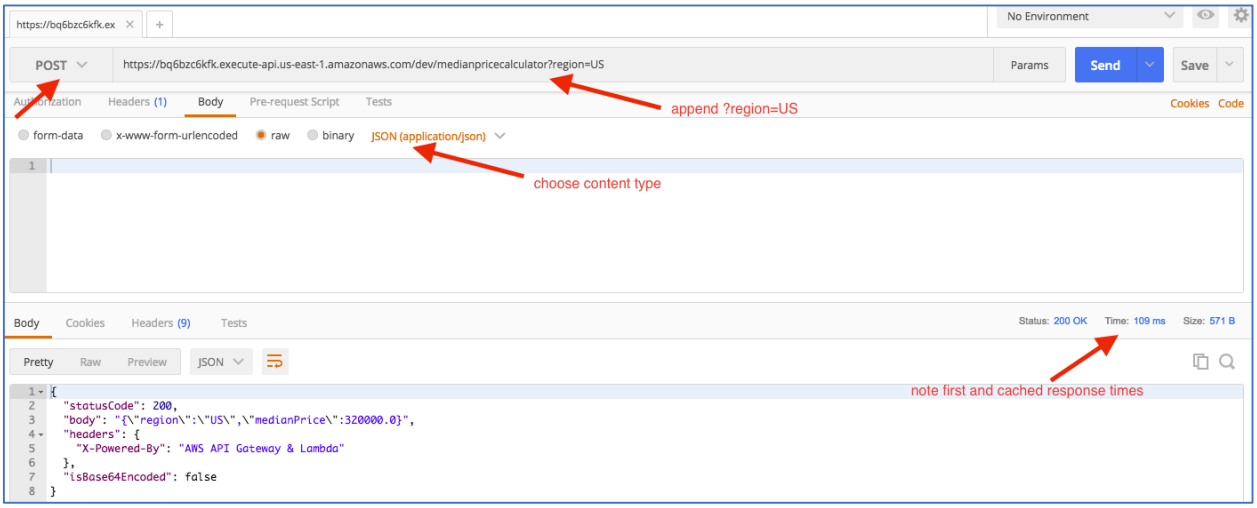
Go ahead and create a new API using POST Man.

1. Enter the copied URL and append **?region=US** at the end of the URL

Example:

http://….i.us-east-1.amazonaws.com/dev/medianpricecalculator?region=US

2. Change Method type to '**POST**'
3. Click on '**Body**' and choose '**Raw**'
4. Set Content-Type to '**JSON (application/xml)**



5. Click **'Send'**

Notice the Response time for the first invocation. Go ahead and invoke the method several more times. You will notice that the subsequent invocation response time is much faster.

You can change the region to 'CA' to get the response for Canada region.

(OPTIONAL)We have set the cache to expire after 60 seconds. You can also manually flush the cache from the API Gateway console and observer the cached response times after the flush.

Usage Plans and Message Throttling

To prevent your API from being overwhelmed by too many requests, Amazon API Gateway throttles requests to your API. There are pre-defined steady-state and burst throttling limits set at the account level. As an API owner, you can set the default method throttling to override the account-level request throttling limits for a specific stage or for individual methods in an API. In addition, you can setup usage plans to restrict client request submissions to within specified request rates and quotes.

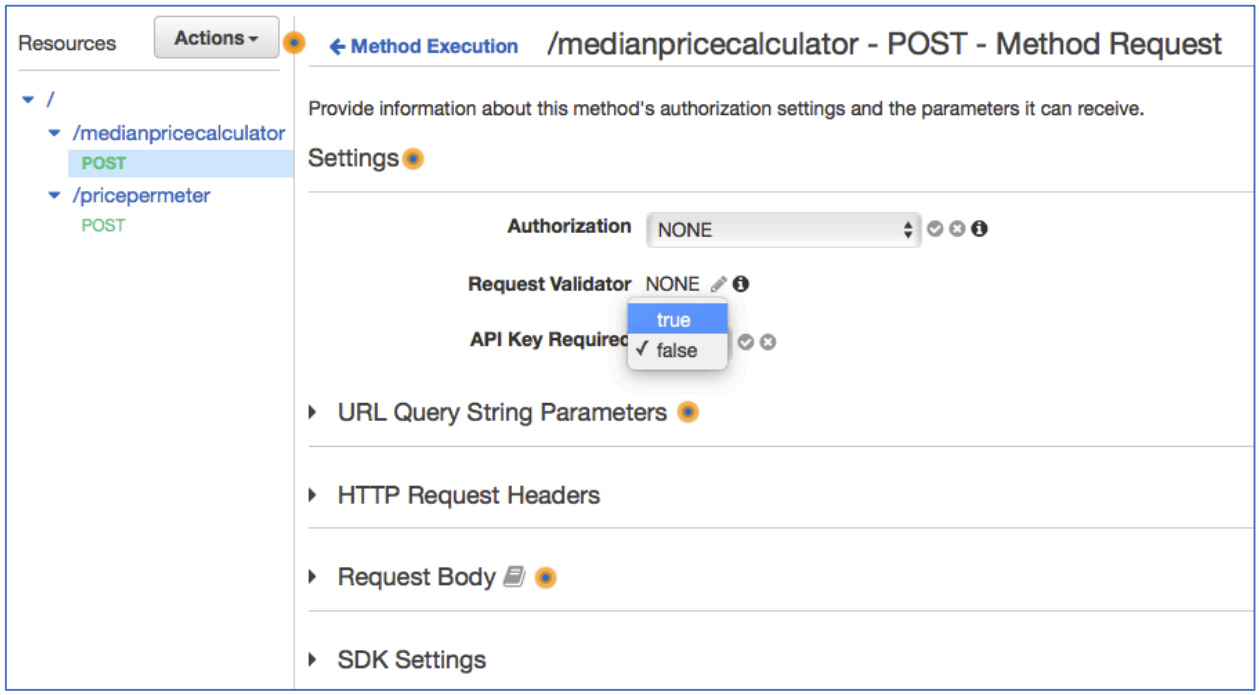
A usage plan provides access to one or more deployed API stages with configurable throttling and quota limits enforced on individual client API keys. API callers are identified by API keys

that can be generated by API Gateway. The throttling prescribes the request rate limits applied to each API key.

In the subsequent sections of the lab, we are going to setup API Keys to track our API Callers. The API keys will then be used to setup usage plans and restrict each client based on their tier level (platinum, gold, silver, etc).

Setting up API Keys

1. If not already done, Sign in to the AWS Management Console and open the API Gateway console at https://console.aws.amazon.com/apigateway/.
2. In the API Gateway main navigation pane, choose **POST** action of medianpricecalculator.
3. Select **Method Request**
4. Under the **Settings** section, choose `true` for **API Key Required**.
5. Select the **check-mark** icon to save the settings.

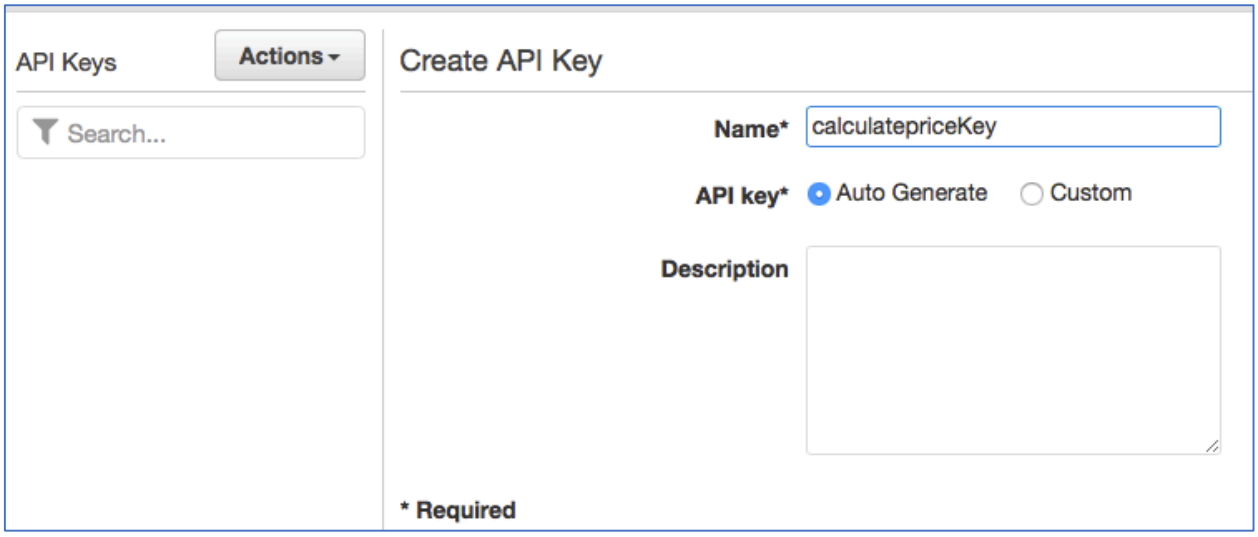


6. Deploy or redeploy the API for the requirement to take effect. (Choose Actions → Deploy API). Deploy to the **dev** stage.

Now that our API requires an API key, we have to create one.

1. In the API Gateway main navigation pane, choose **API Keys**.
2. From the **Actions** drop-down menu, choose **Create API key**.
3. Enter fowllowing values:

   Name: ‘calculatepriceKey’
   API Key: Autogenerate



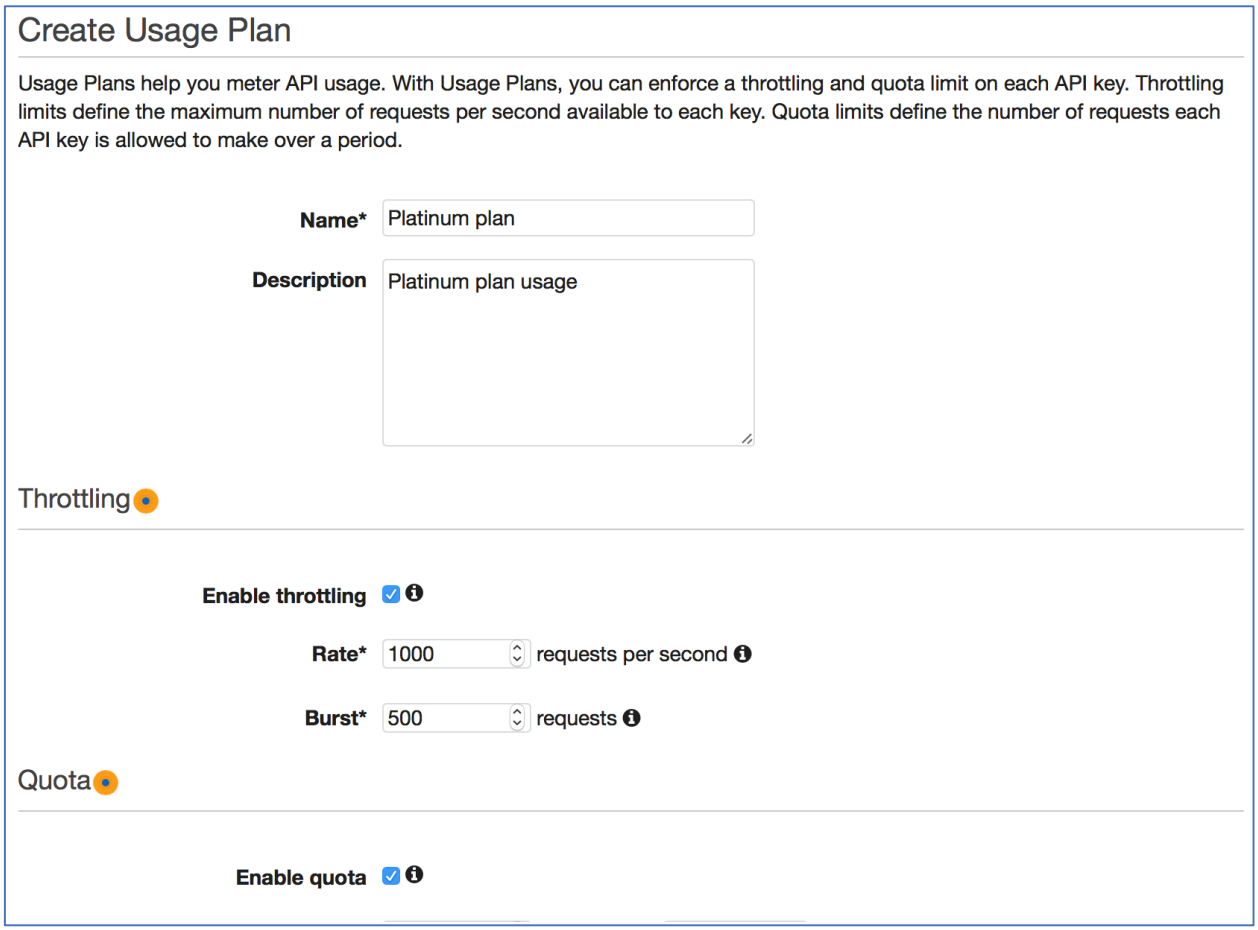
4. Click **Save**

Setting Up Usage Plans

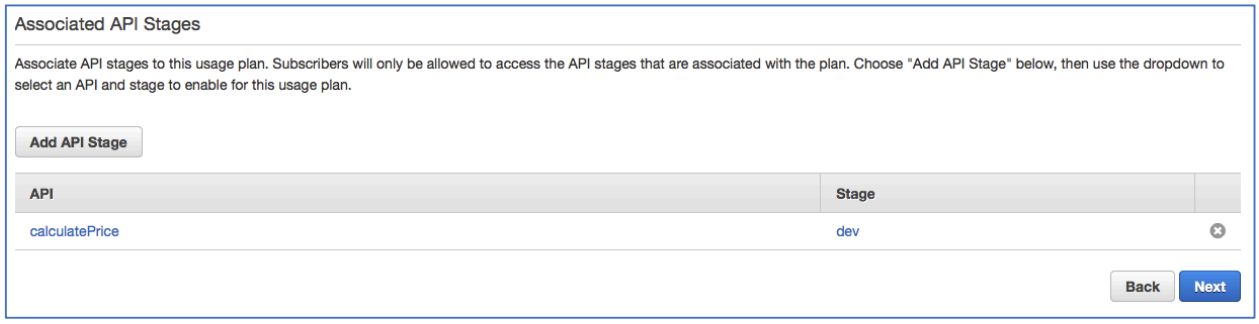
Once the API keys are created, the next step is to create a Usage plan.

1. In the Amazon API Gateway main navigation pane, choose **Usage Plans**,
2. Click **Create** to create a new usage plan.
3. Under **Create Usage Plan**, enter the following:

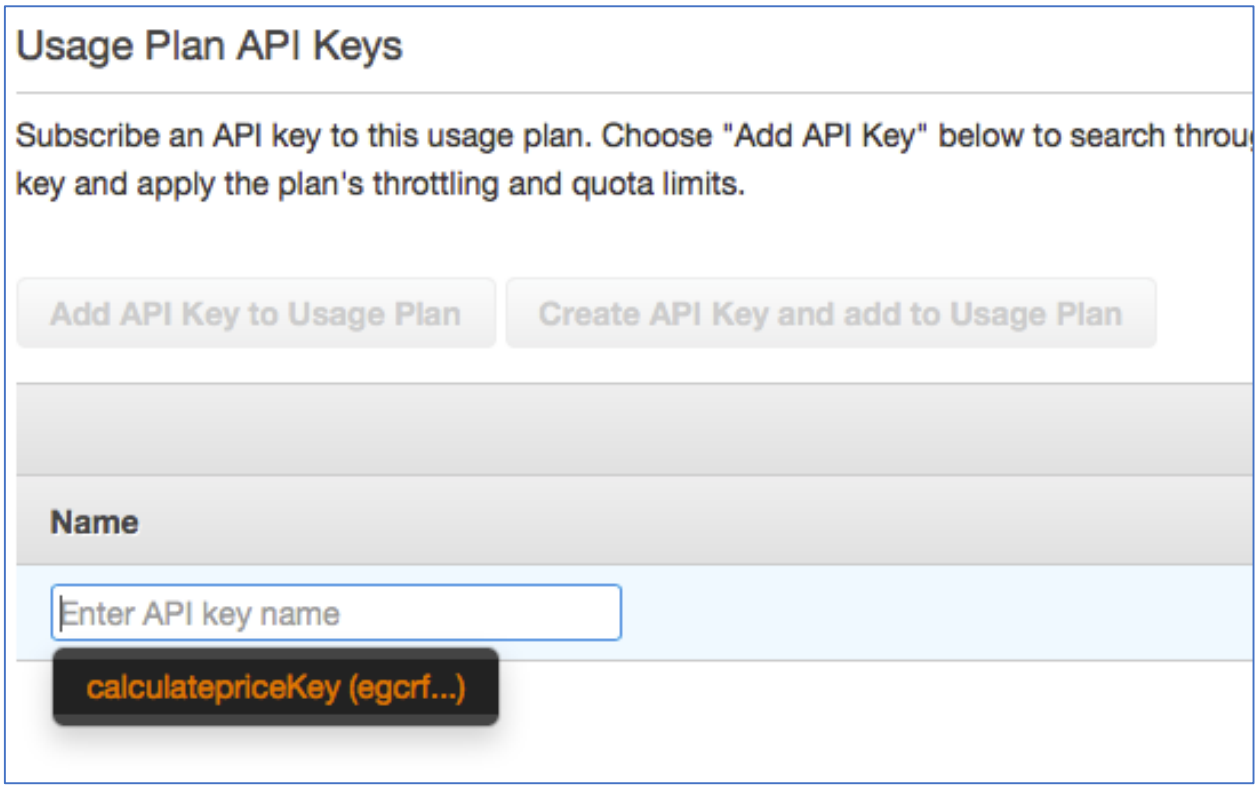
   For Name, type ‘**Platinum**’
   For **Description**, type a description for your plan.
   Select **Enable throttling** and set **Rate** to **1000** and **Burst** to **500**
   Choose **Enable quota** and set its **limit** to **5000** for a selected time interval **Month**.



4. Choose **Next**.
5. Click '**Add API Stage**'
6. Choose '**CalculatePrice**' API
7. Choose '**dev**' stage
8. Select the **check box** to confirm



9. Click '**Next**'
10. Click '**Add API Key to Usage Plan**'
11. Type '**calculatepriceKey**'



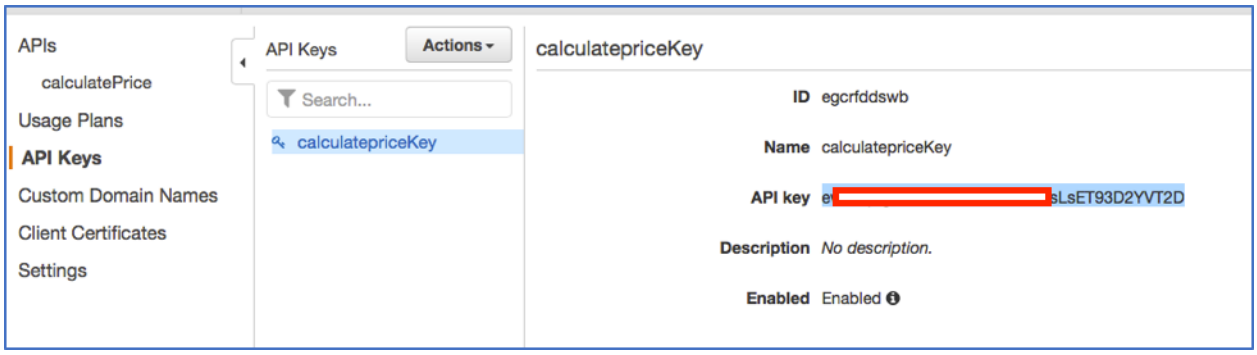
12. Click '**check mark**' to confirm
13. Click **Done**

At this point, we have created a 'platinum' usage plan that enables the calculate price API to accept 100 requests and burst up to 200 requests. If desired, we can also create "silver" and "bronze" plans that enable fewer requests.

Testing API with Usage Plan

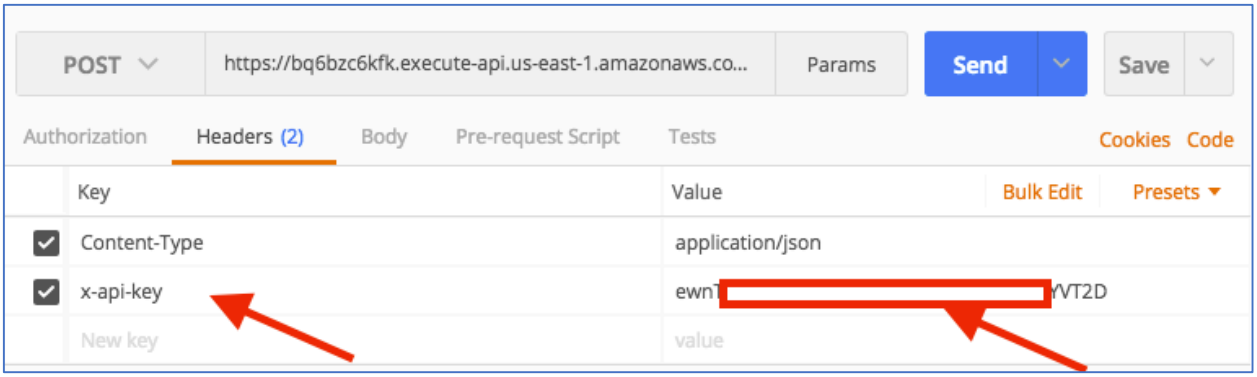
Once the Keys are created and associated with an API, they are typically distributed to developers or customers. In this lab, we will use POSTMAN to test our deployed service with the API Keys.

1. In the Amazon API Gateway main navigation pane, choose **API Keys**
2. Select '**calculatepriceKey**'
3. Click '**Show**' to reveal the **API Key**
4. Copy the API Key in a notepad



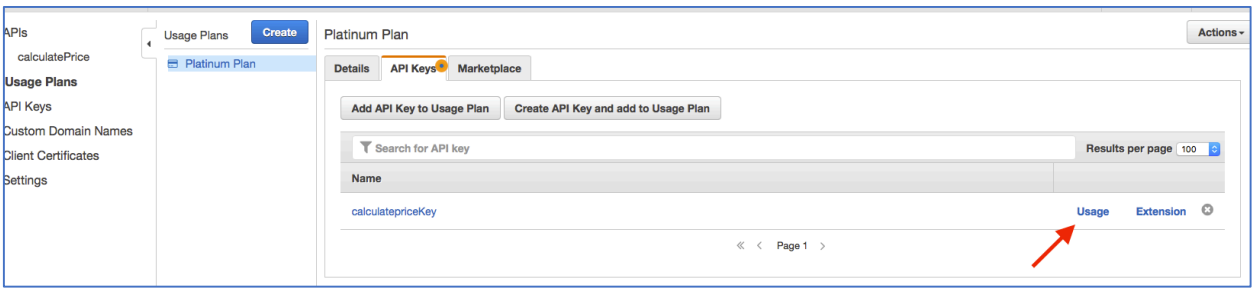
Using Postman, configure the following

1. Select the 'medianpricecalculator' in POSTMAN
2. Set the Header attribute 'x-api-key'
3. Set the value with your API KEY
4. Click **Send**



Invoke the method several times and then go to usage plan to see the number of invocations remaining.

1. In the Amazon API Gateway main navigation pane, choose **Usage Plans**
2. Click on **'Platinum Plan'**
3. Click on '**API Keys'**
4. Click '**Usage'**



5. Notice the usage **statistics**. **Please note that usage data can be delayed**.
6. Go back to the Usage Plan and Click on **Extension**
7. You can observer the number of requests that are still available (**Please note that usage data can be delayed**).

Lab 2 Summary

In the second part of this lab, you were able to successfully enable caching for your API. You learned how to cache data at the stage level, and method level. In particular, you were able to create an API and cache the data based on the parameter query. In addition to caching, you were also able to setup API Keys and associate those keys with a usage plan. The usage plan restricted the throttling limit of your API.

Appendix

Lambda Function Overview

This lab deploys two lambda functions. Below is a description of each function and the applicable lab sections.

Function Name	Description	Lab Usage
CostCalculator	Lambda function that calculates the price per unit and the total cost of a house. Price per unit = price/size Total cost = downpaymentAmount + price	Lab 1
MedianPriceCalculator	Lambda function that returns the median price of a house in US ($320,000) and Canada (240,000).	Lab 2

Deploying Lambda Functions

1. Download the Serverless Application Model (SAM) template (package-template.yaml). https://s3.amazonaws.com/serversless-immersion-api-gateway/packaged-template.yaml packaged-template.yaml

2. If not already done, install AWS CLI - Follow the AWS CLI Getting Started guide to install and configure the CLI on your machine

3. If not already done, run the following command to configure your AWS CLI

```
$ aws configure
AWS Access Key ID [None]: <Enter Access Key ID>
AWS Secret Access Key [None]: <Enter Secret key>
Default region name [None]: us-east-1
Default output format [None]: json
```

4. Run the following command to deploy the lambda functions

```
aws cloudformation deploy --template-file path-to-template-file/packaged-template.yaml --stack-name serverless-immersion-day-stack --capabilities CAPABILITY_IAM
```

The above command deploys two lambda functions.

Once the above command is executed, you can verify the deployment of these functions by going to the lambda console.

5. Go to the Lambda console at https://console.aws.amazon.com/lambda
6. Verify that **serverless-immersion-day-api-CalculateCostPerUnit**-XX and **serverless-immersion-day-api-MedianPriceCalculator-**XX are listed.

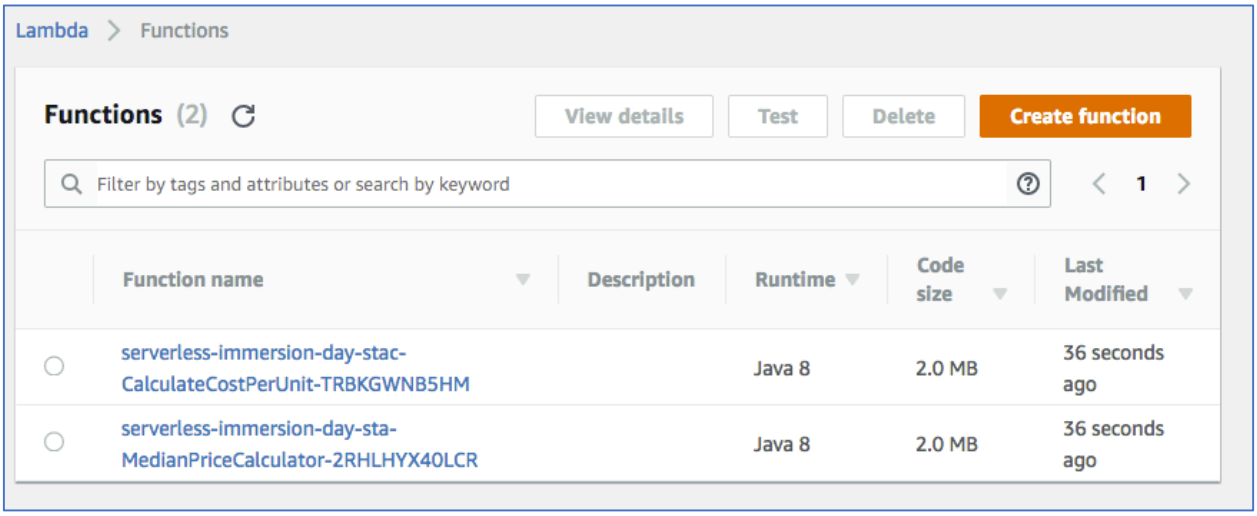


Figure 21: Lambda functions

You are now ready to start the lab !!