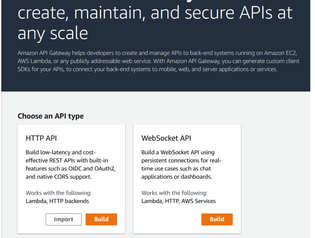
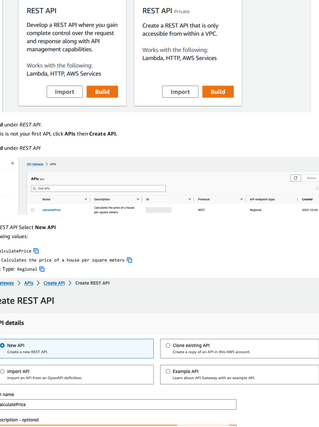
Create Your First API

1. Sign in to the AWS Management Console and open **Amazon 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.

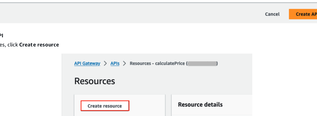




- Click on **Build** under *REST API*.

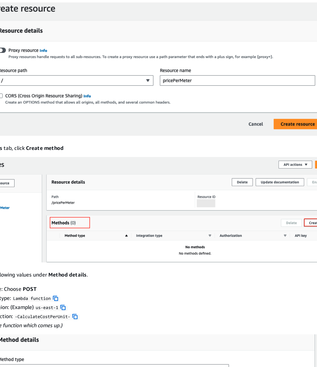
3. (Optional) If this is not your first API, click **APIs** then **Create API**.

- Click on **Build** under *REST API*

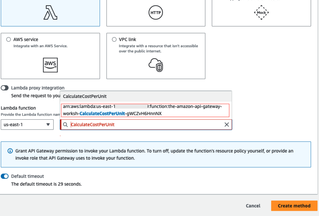


4. Under *Create REST API* Select **New API**
5. Enter the Following values:

- API name: `calculatePrice`
- Description `Calculates the price of a house per square meters`
- API Endpoint Type: `Regional`



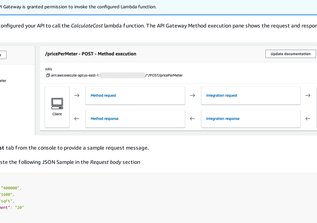
6. Click **Create API**
7. Under Resources, click **Create resource**



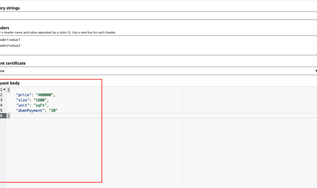
8. Enter the Following values:

- Resource path: Choose def ault (/)
- Resource name: `pricePerMeter`

9. Click **Create resource**



10. Under **Methods** tab, click **Create method**



11. Provide the following values under **Method details**.

- Method type: Choose **POST**
- Integration type: `Lambda function`
- Lambda Region: (Example) `us-east-1`
- Lambda Function: `CalculateCostPerUnit`
  - *(Select the function which comes up.)*

12. Click **Create method**

> **Notice**
> By default, API Gateway is granted permission to invoke the configured Lambda function.

13. You have now configured your API to call the *CalculateCost* lambda function. The API Gateway Method execution pane shows the request and response flow of data.

14. Click on the **Test** tab from the console to provide a sample request message.

- Copy and Paste the following JSON Sample in the *Request body* section

```
{
  "price" : "400000",
  "size" : "1500",
  "unit": "sqft",
  "downPayment" : "20"
}
```


15. 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.

- 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.

The Amazon API Gateway Workshop

- Introduction
- Getting Started
- Module 1 - Introduction to Amazon API Gateway
  - Module Goals
  - Create Your First API
  - Message Transformation
  - Request Validation
  - Authentication and Authorization with Cognito
  - API Deployment
  - Message Caching
  - Usage Plans and Message Throttling
  - Authentication and Authorization with IAM
  - Clean Up
- Module 2 - Deploy your first API with IaC
- Module 3 - API Gateway REST Integrations
- Module 4 - Observability in API Gateway
- Module 5 - WebSocket APIs
- Module 6 - Enable fine-grained access control for your APIs
- Clean up
- Resources

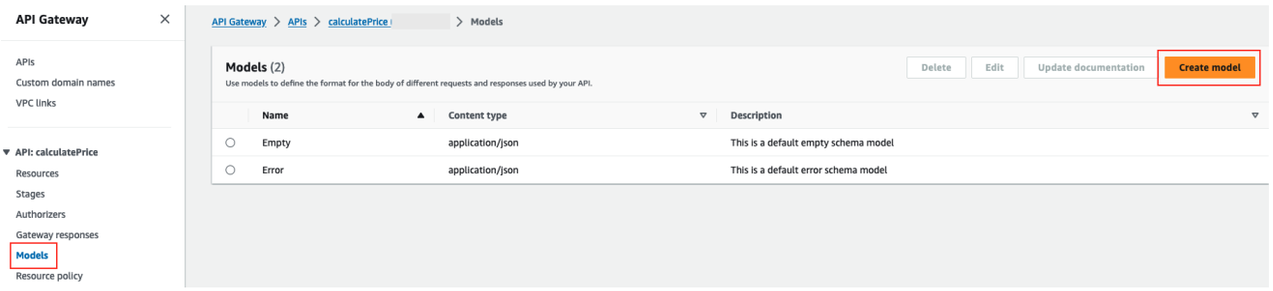
Message Transformation

In API Gateway, a mapping template is used to transform data from one format to another. 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.

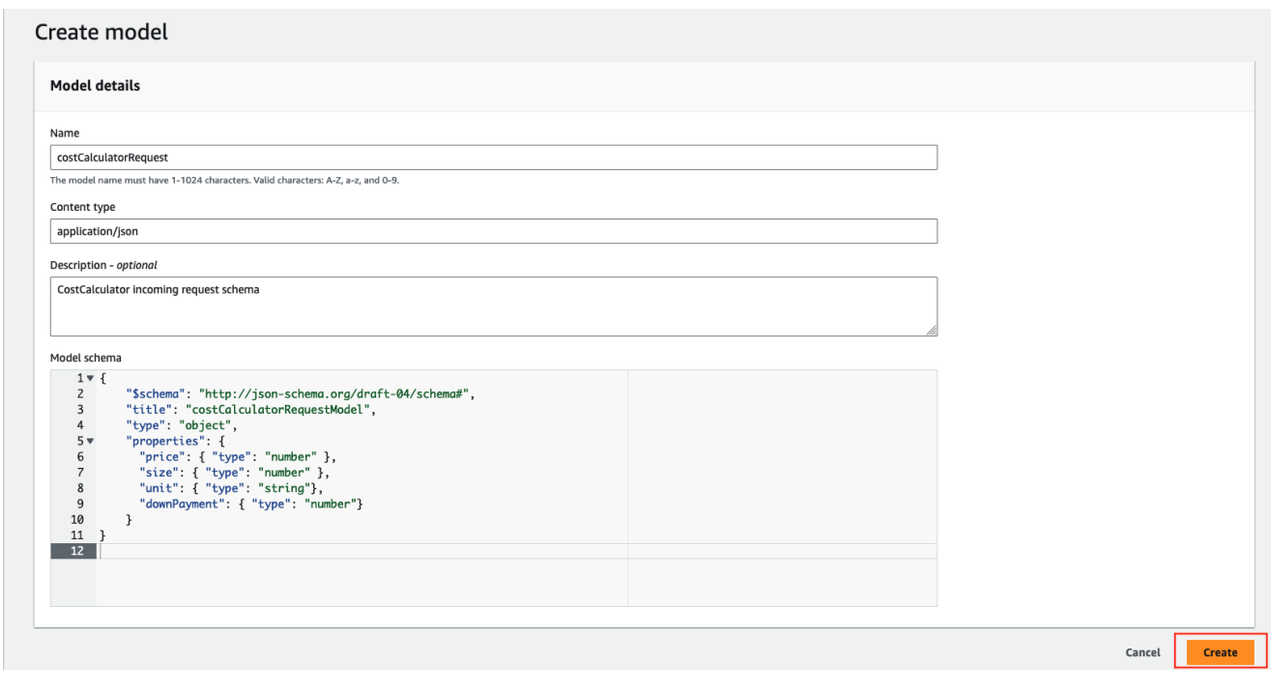
Building a Model

1. Select **Models** from the navigation menu under *calculatePrice* API.



2. Click **Create model**.
3. Enter the following values under *Model details*:
   - Name: `costCalculatorRequest`
   - Content type: `application/json`
   - 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"}
    }
}
```

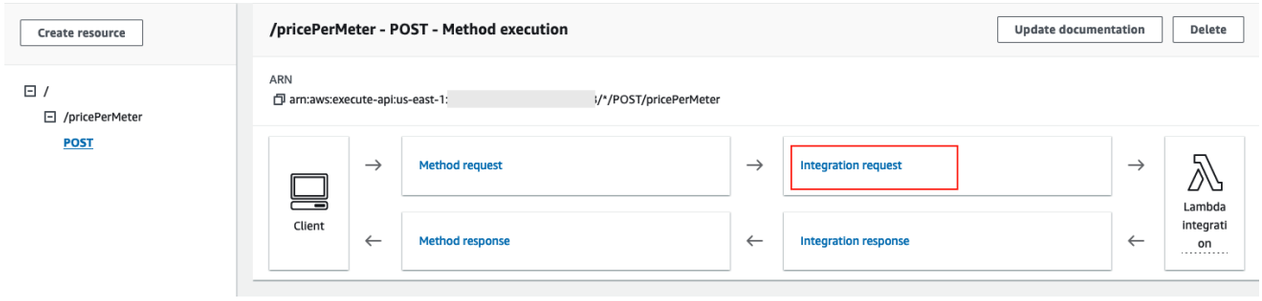


4. Click **Create**.

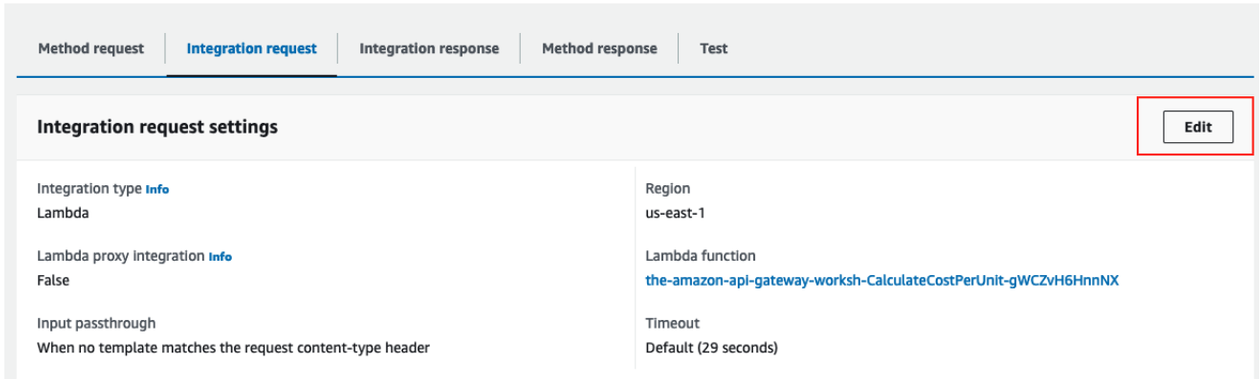
Now that the models are created, the next step is to transform the incoming request and response messages.

Transform Request Payload

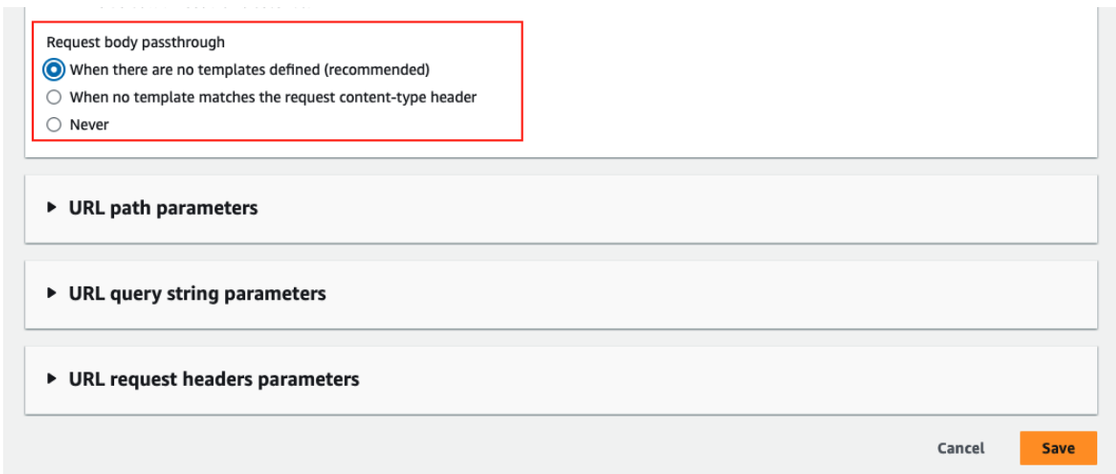
1. Go back to the */pricePerMeter* API resource by clicking on **Resources** section from the left navigation menu.
2. Click on the **POST** action.
3. From the Method Execution pane, click on **Integration request**.



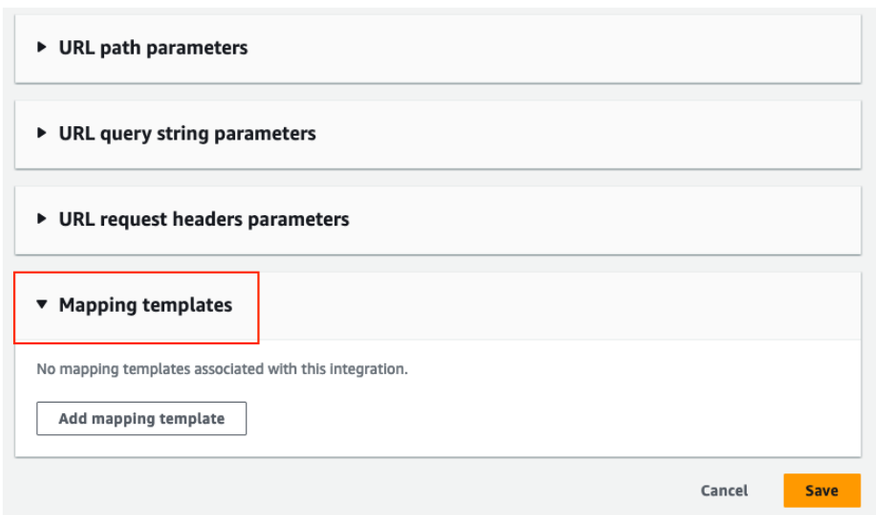
4. Under *Integration request settings*, click on **Edit**



5. Select *Request body passthrough* as **When there are no templates defined (recommended)**



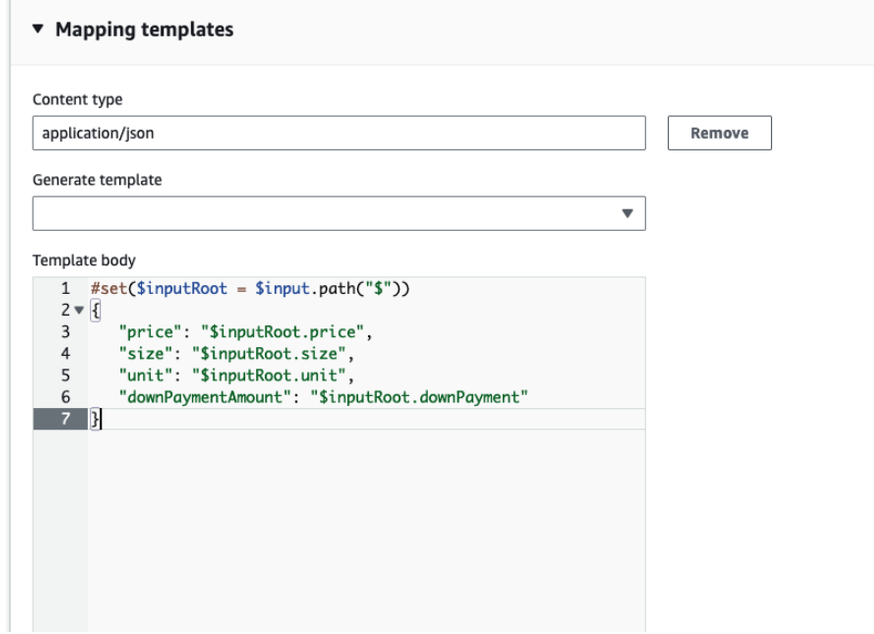
6. Scroll down to **Mapping templates** section. Click **Add mapping template**



7. Provide the following values under Mapping Templates.

- *Content type*: Type `application/json`
- *Generate template*: Leave Blank
- Copy and Paste the following content in the **Template body** section

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



8. 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**
> The last statement is mapping the value of downPayment to a new attribute called downPaymentAmount.

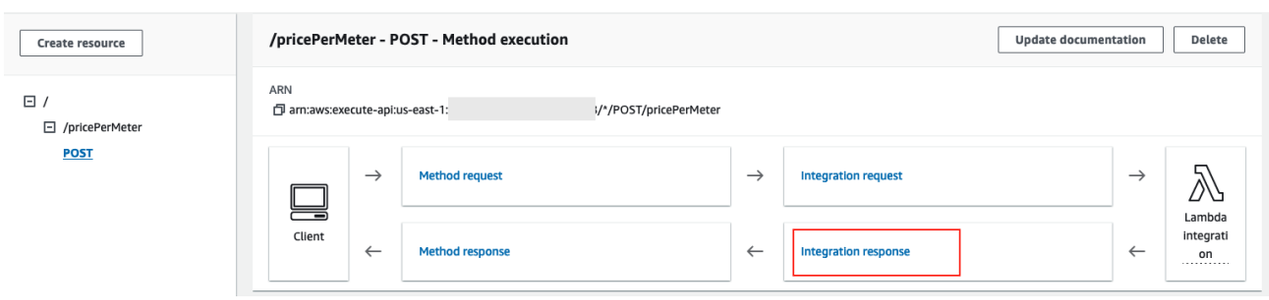
The downstream lambda function uses the downPaymentAmount to calculate the total price. (i.e. `totalPrice = price + downPaymentAmount`).

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 VTL mapping template

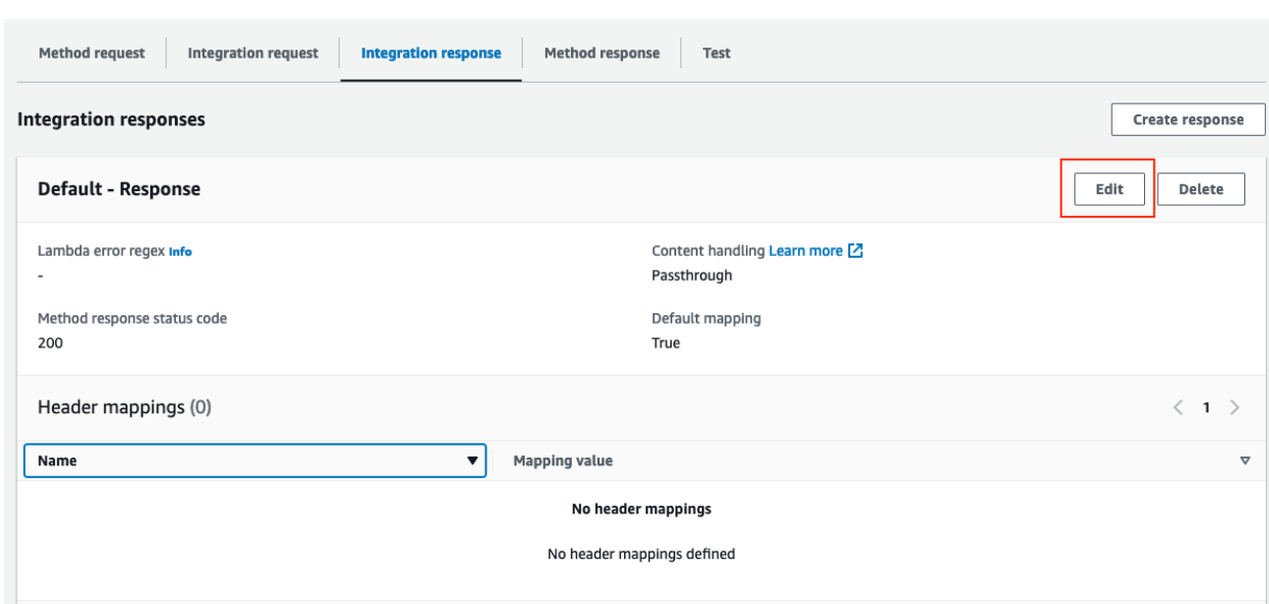
Transform Response Payload

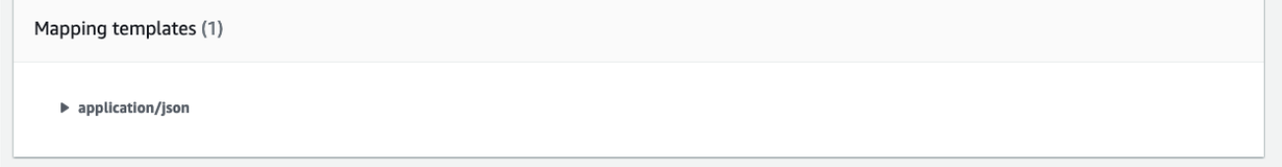
Similar to the request mapping, create the mapping template for the response payload. Perform the following:

1. Go back to the */pricePerMeter* API resource by clicking on **Resources** section from the left navigation menu.
2. Click on the **POST** action.
3. From the Method Execution pane, click on **Integration response**.



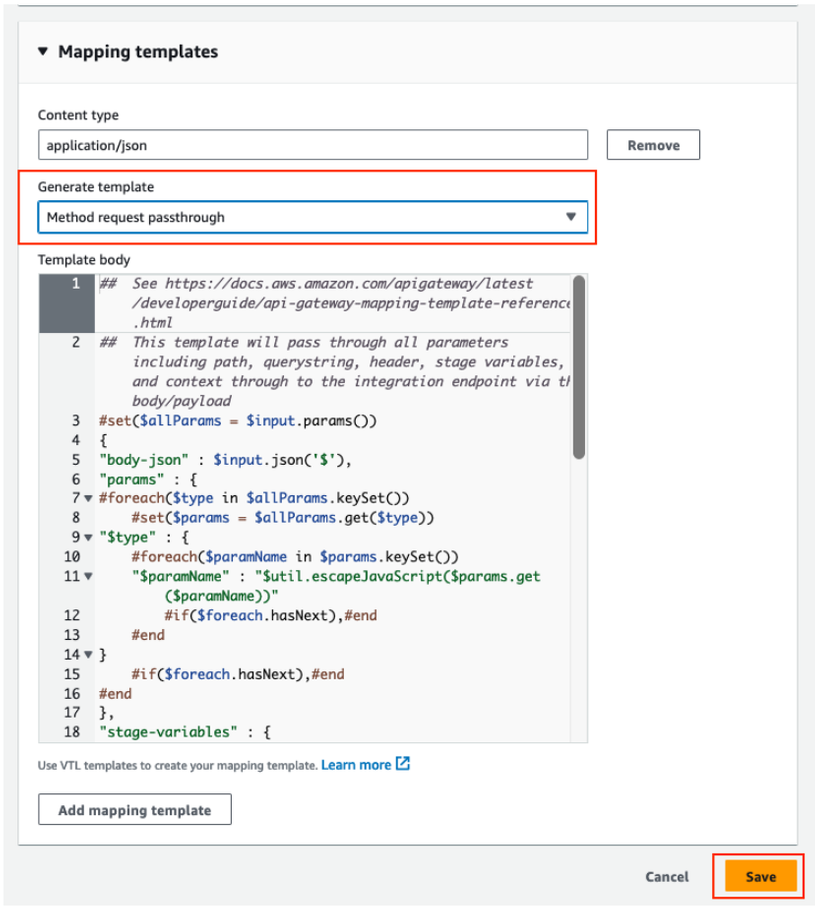
4. Under *Integration responses* pane, click **Edit**





5. Scroll down to **Mapping templates** section. Provide the following values under Mapping Templates.

   - *Content type*: Type `application/json`
   - *Generate template*: Choose **Method Request Passthrough**
   - Leave the Template body as it is



6. Click **Save**

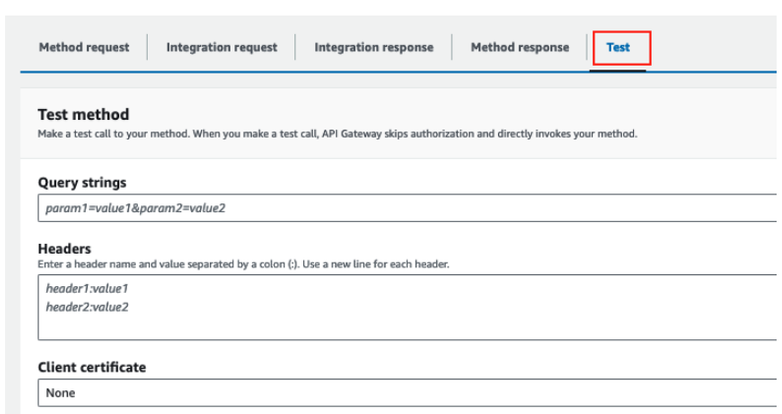
Test the API

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 **/pricePerMeter** API resource and click on the *POST* method to test your API.
2. Click on the **Test** tab to provide a sample request message.

   - Copy and Paste the following JSON Sample in the *Request body* section

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



Request body

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

Test

3. Click on **Test**

/pricePerMeter - POST method test results

Request	Latency	Status
/pricePerMeter	31	200

Response body


```
{
"body-json" : {"statusCode":200,"body":"{\"pricePerUnit\":\"250.0\",\"medianPrice\":\"320000.0\",\"totalCost\":\"400020.0\"}","headers":{"X-Powered-By":"AWS API Gateway & Lambda
Serverless"},"isBase64Encoded":false},
"params" : {
"path" : {
    }
    ,"querystring" : {
    }
    ,"header" : {
    }
    },
"stage-variables" : {
},
"context" : {
    "account-id" :
    "api-id" : "          ",
    "api-key" : "test-invoke-api-key",
    "authorizer-principal-id" : "",
    "caller" : "                                    ",
    "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" : "                                    ",
    "user-agent" : "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:109.0) Gecko/20100101 Firefox/115.0",
    "user-arn" : "                                                                                    ",
    "request-id" : "1b73e589-06fe-45b8-a6ff-0de02d7dbf64",
    "resource-id" : "ap555i",
    "resource-path" : "/pricePerMeter"
    }
}
```


Response headers


```
{
  "Content-Type": "application/json",
  "X-Amzn-Trace-Id": "Root=1-651ebab0-0575291d374287b6b7e5cbb6;Sampled=0;lineage=078d0308:0"
}
```


Notice

The `downPayment` amount was added to the price to calculate the `totalCost` output. The response also contains the header and context attributes.

Congratulations! You have successfully completed this module.

Select your cookie preferences

We use essential cookies and similar tools that are necessary to provide our site and services. We use performance cookies to collect anonymous statistics, so we can understand how customers use our site and make improvements. Essential cookies cannot be deactivated, but you can choose “Customize” or “Decline” to decline performance cookies.

If you agree, AWS and approved third parties will also use cookies to provide useful site features, remember your preferences, and display relevant content, including relevant advertising. To accept or decline all non-essential cookies, choose “Accept” or “Decline.” To make more detailed choices, choose “Customize.”

Accept

Decline

Customize

Request Validation

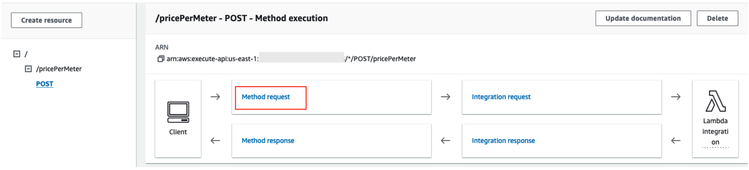
Request validation is used to ensure that the incoming request message is properly formatted and contains the proper attributes. You can set up request validators in an API's Swagger definition file and then import the Swagger definitions into API Gateway. You can also set them up in the API Gateway 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:

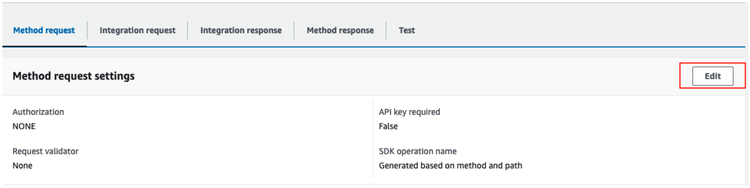
- **Validate body:** This is the body-only validator.
- **Validate query string parameters and headers:** This is the parameters-only validator.
- **Validate body, query string parameters, and headers:** This validator is for both body and parameters validation.

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 under *pricePerMeter* API resource.
2. Choose **Method request** from the Method execution pane.

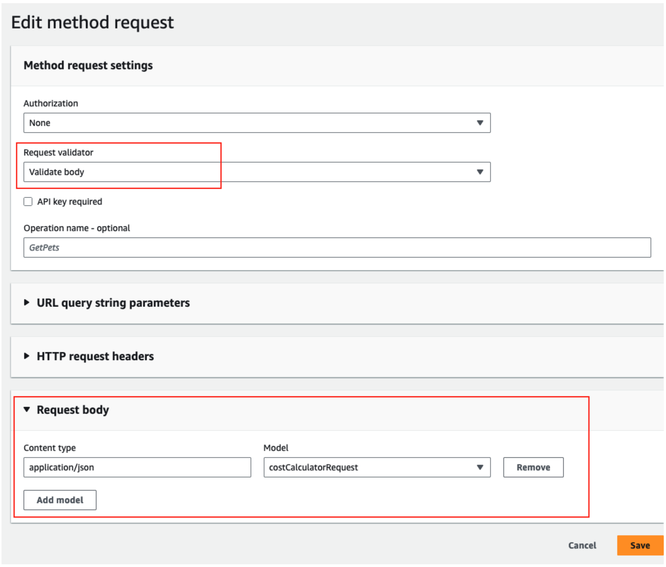


3. Under *Method request settings*, click on **Edit**



4. Choose *Request validator* as **Validate body** from the dropdown.
5. **Expand** Request Body
6. Click **Add model**
7. Provide the following values.

- *Content Type*: Type `application/json`
- *Model*: Choose **costCalculatorRequest** model from the dropdown



8. Click **Save**

Test the API

1. Go back to your **/pricePerMeter** API resource and click on the *POST* method to test your API.
2. Click on the **Test** tab to provide a sample request message.
3. Copy and Paste the following JSON Sample in the *Request body* section

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

4. Click on **Test**

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

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

5. Click on the **Test** tab to provide a sample request message.
6. Copy and Paste the following JSON Sample in the *Request body* section

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

7. Click on **Test**

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

Authentication and Authorization with Cognito

At this point, our API is publicly exposed and you 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.

Below are the steps to enable authentication and authorization to the API:

- **Create Cognito User Pool**
- **Create a User**
- **Login as user**
- **Update API Gateway Authentication**

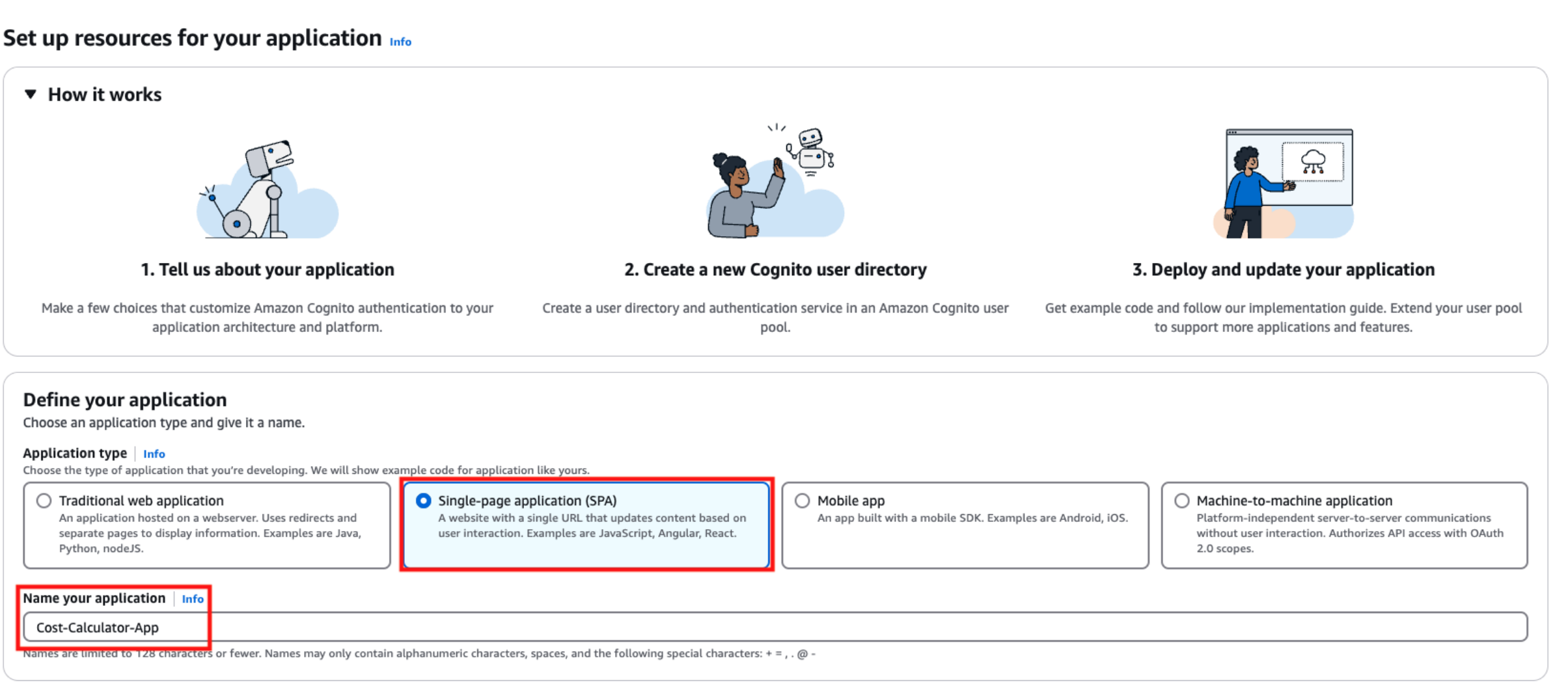
Create Cognito User Pool

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

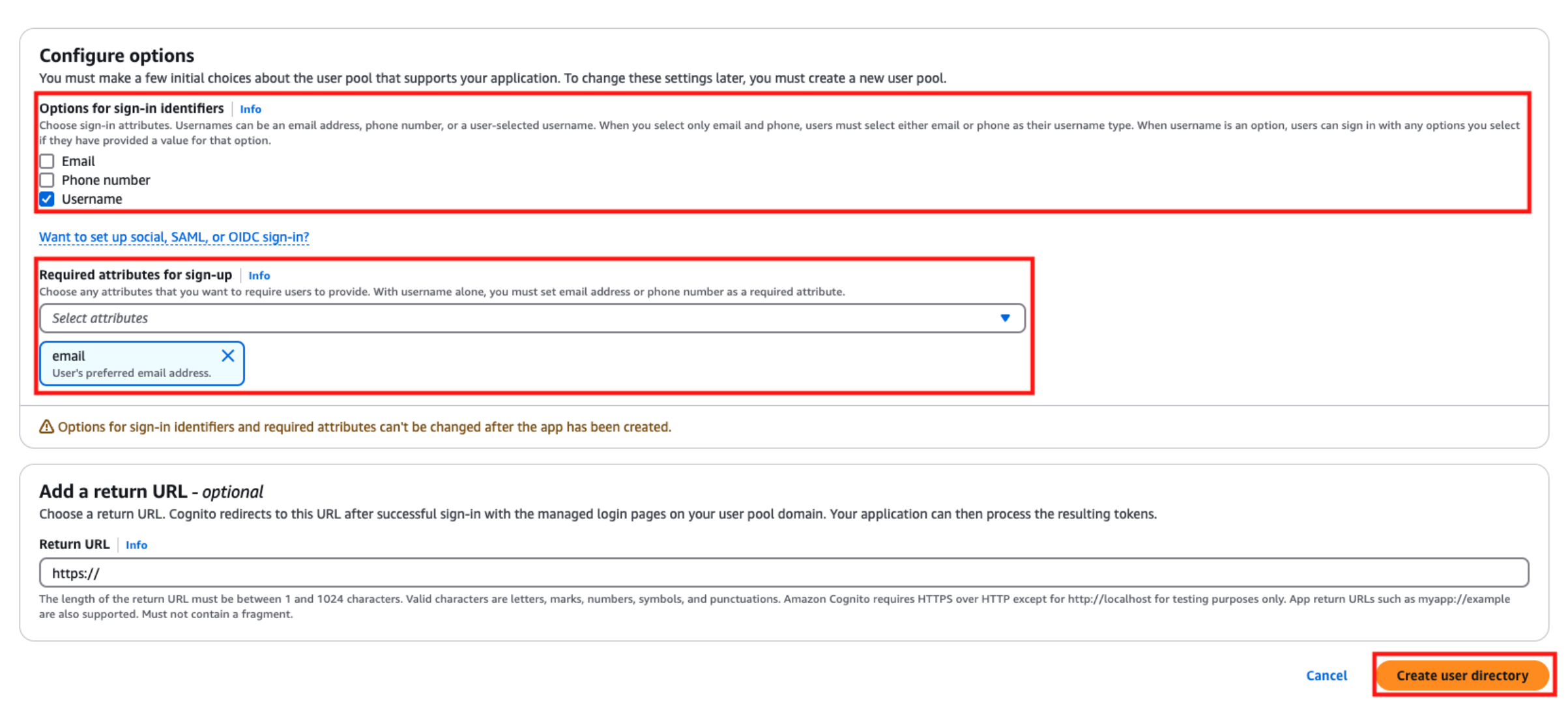
1. Open the Amazon Cognito console, select the right region if necessary and sign in.

> You should see one User Pool named `RealEstateUserPool` that is already created by the event to be used at the module 6. You can leave it as it is.

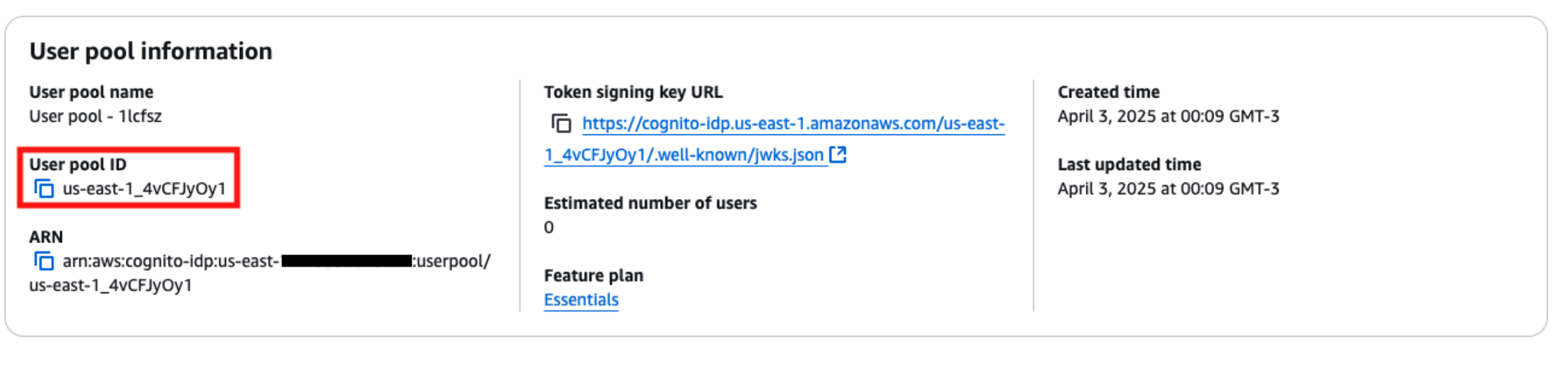
2. Click on **Create a user pool**
3. In **Set up resources for your application**, select **Single-page application (SPA)**
4. In **Name your application** type `Cost-Calculator-App`



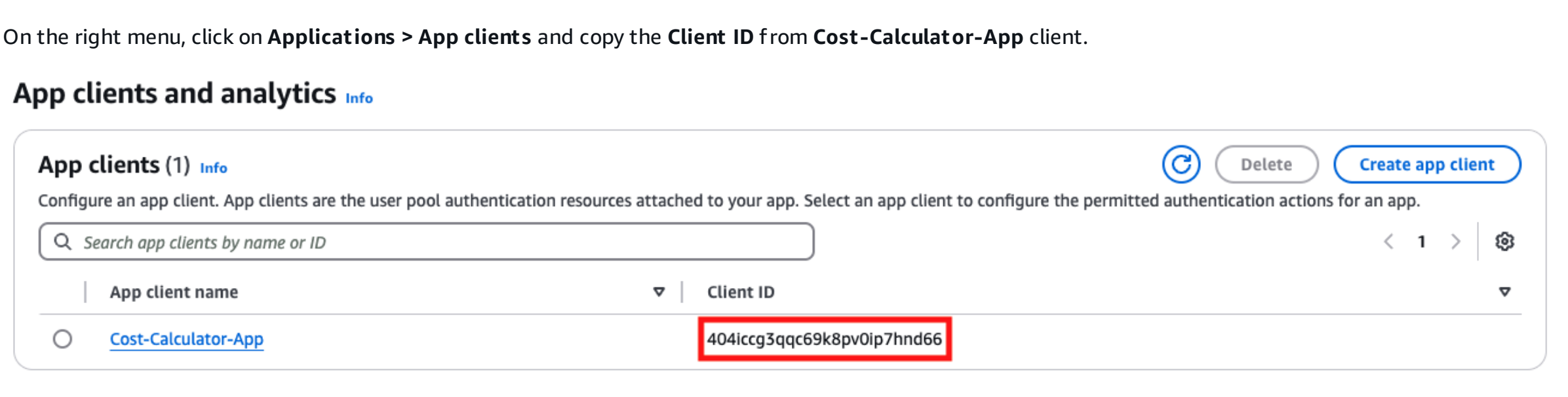
5. In **Configure options**, mark **Username**, in **Required attributes for sign-up** select **email** and click **Create user directory**.



6. Review the next page and click in **Go to overview**
7. On the overview page, copy the **User Pool ID** into a text editor, we will use this later.

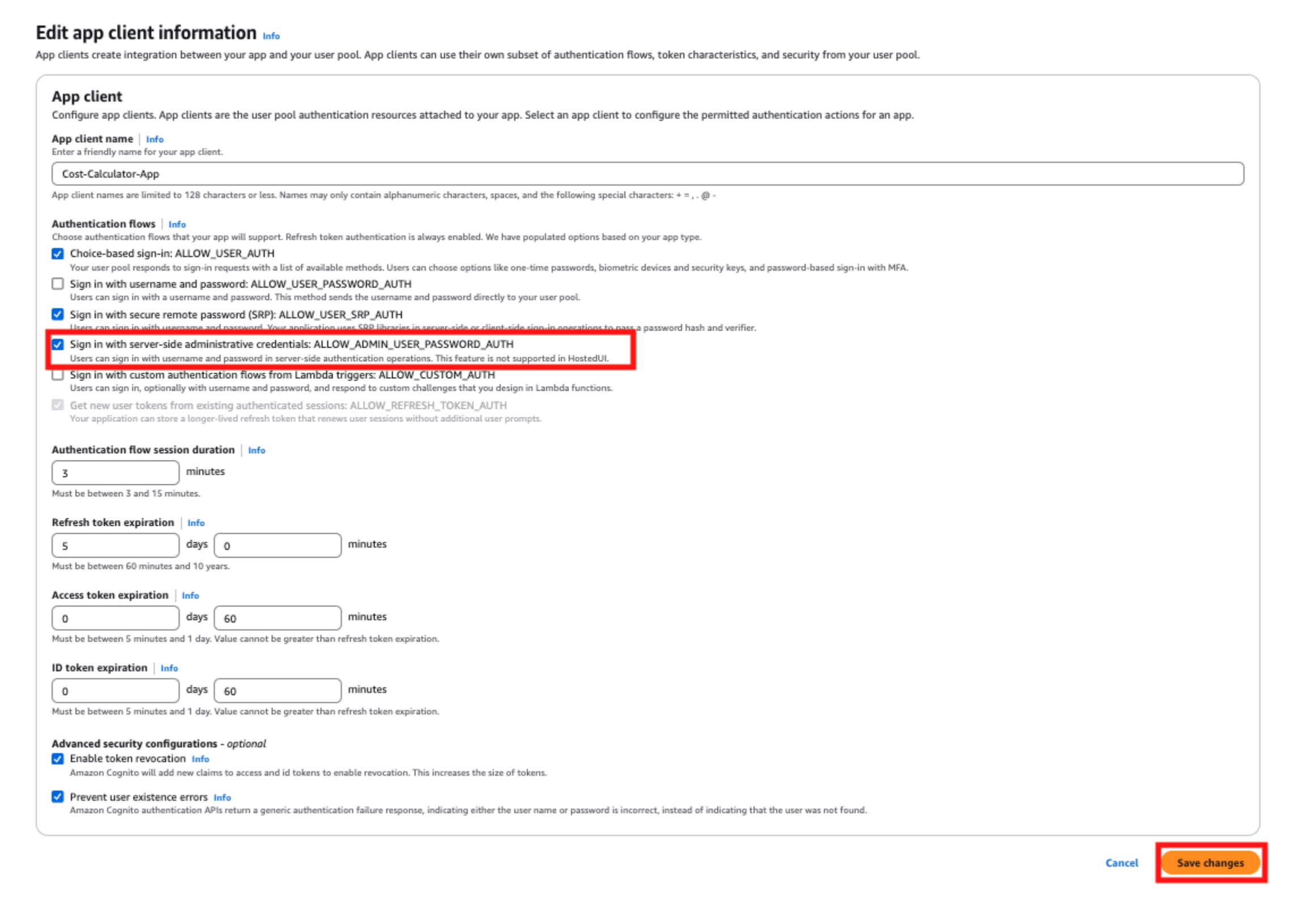


8. On the right menu, click on **Applications > App clients** and copy the **Client ID** from **Cost-Calculator-App** client.



> For the purposes of this workshop, we will also enable the authentication flow `ALLOW_ADMIN_USER_PASSWORD_AUTH`, which will allow the admin to change passwords and authenticate users.

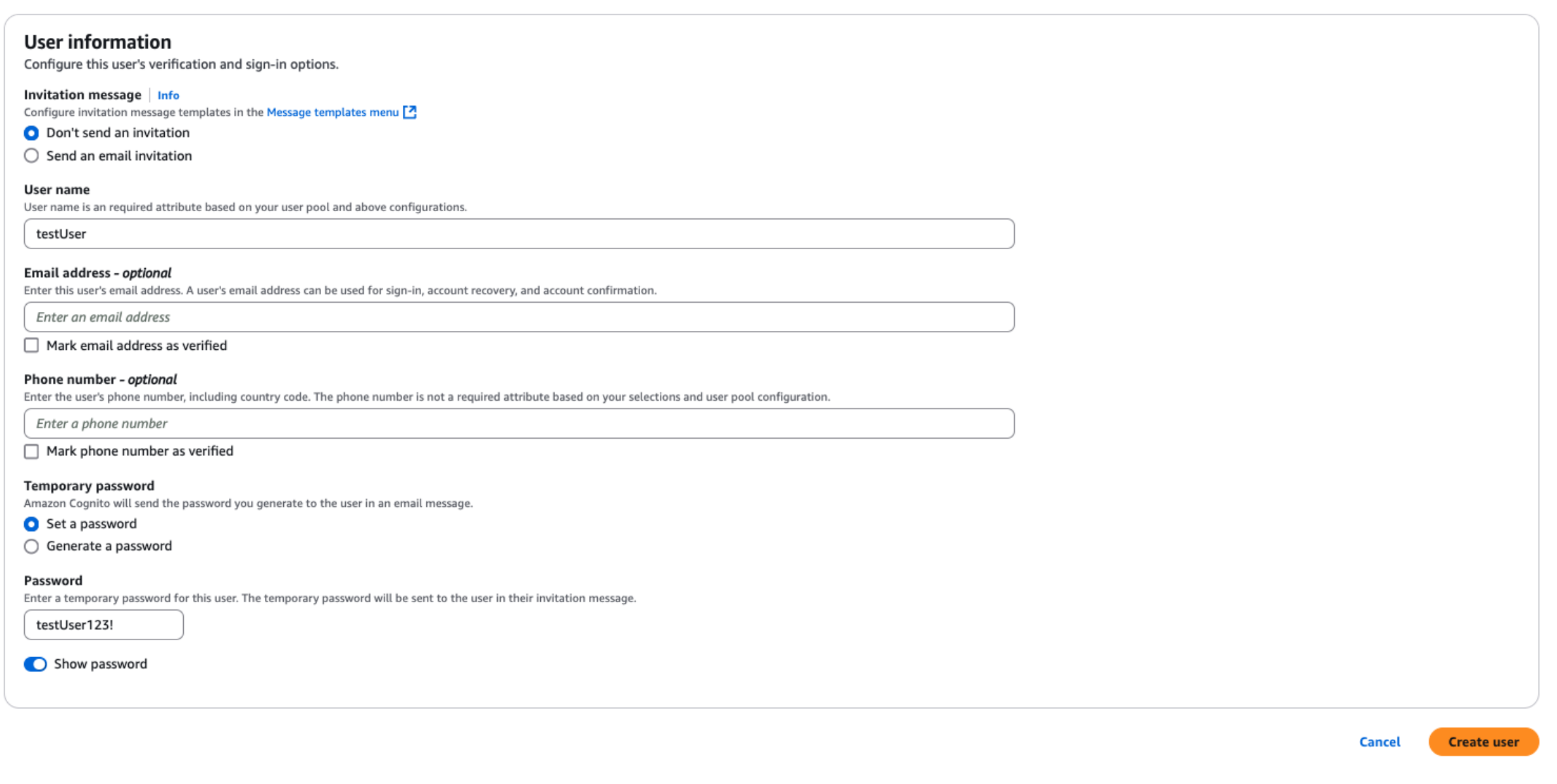
9. Click on the **Cost-Calculator-App** App client and then on the **Edit** button.
10. Select the option **Sign in with server-side administrative credentials: ALLOW_ADMIN_USER_PASSWORD_AUTH** and than **Save changes** button.



Create a User

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

1. Still in **User pool Overview** screen click on **User management > Users** tab and then **Create User**
2. Enter the following values:
   - On Invitation Message let **Don't send an invitation selected**.
   - UserName: `testUser`
   - On Temporary password let **Set a password selected**.
   - In password paste: `testUser123!`



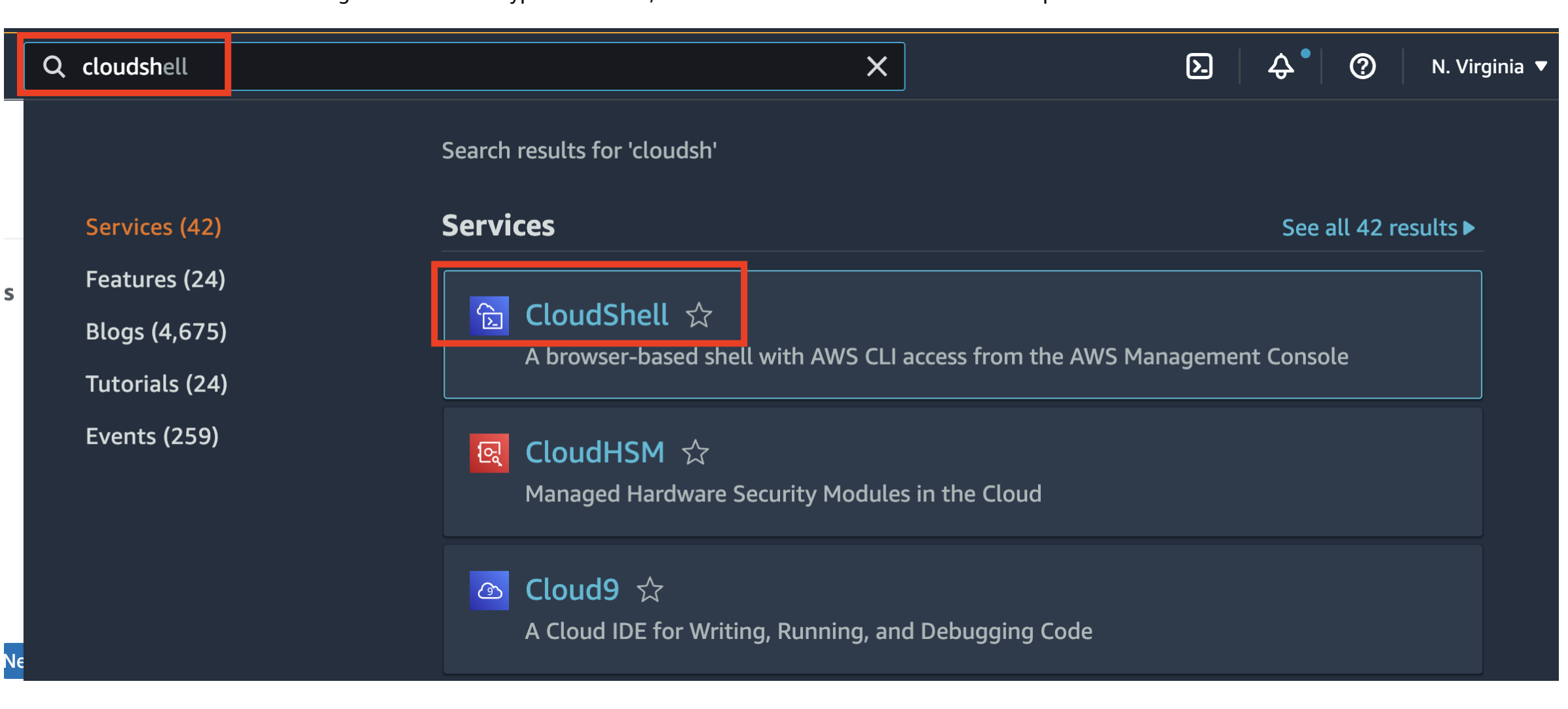
3. Click **Create User**

Login as user

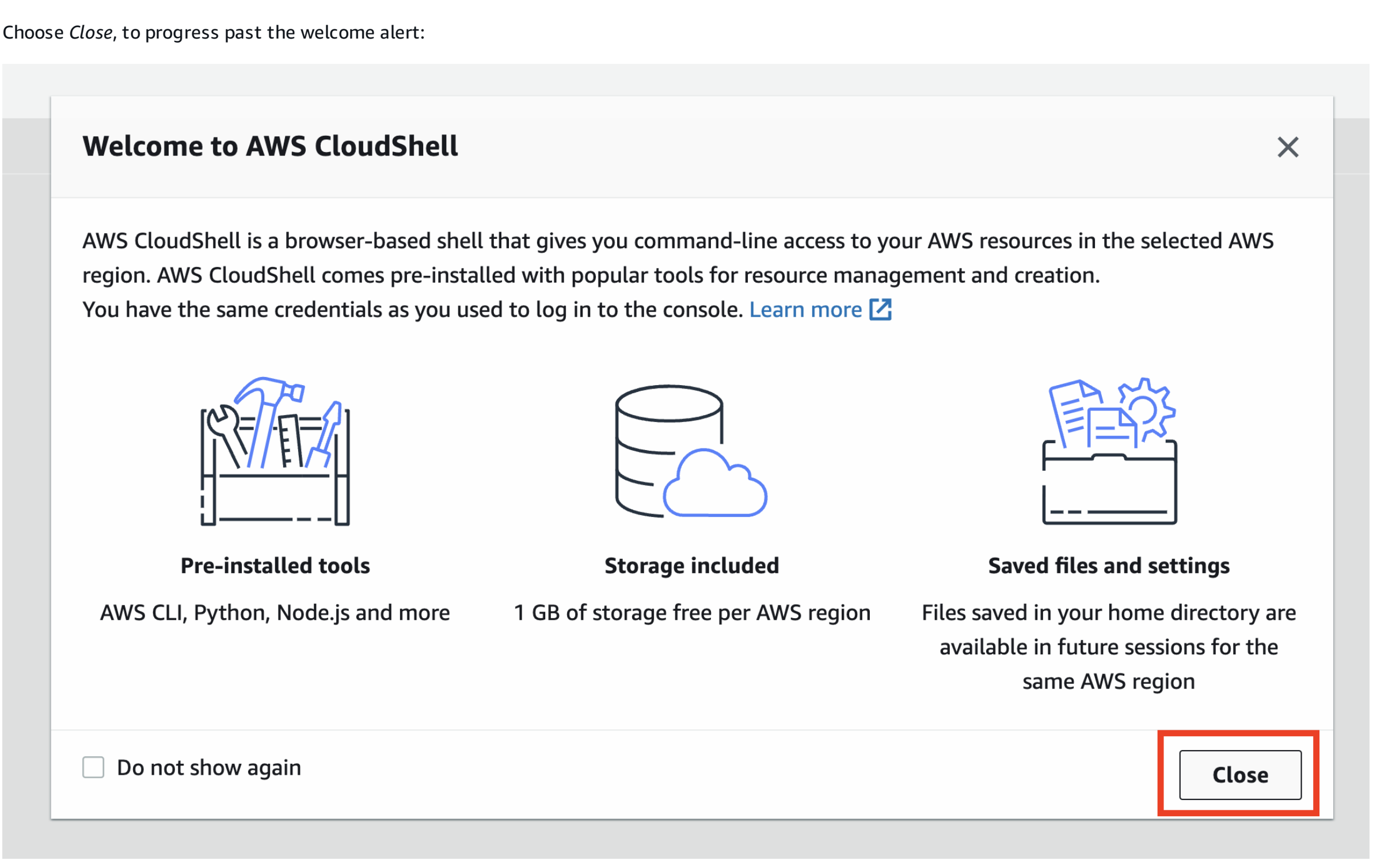
Now that the user pool is configured, use will use a service called AWS CloudShell and AWS CLI to simulate a login for the user.

> **Tip**
> AWS CloudShell is a browser-based shell terminal that makes it easy to securely manage, explore, and interact with your AWS resources to run the API command.

1. In the Search Bar of the AWS Management Console type CloudShell, and chose CloudShell from the search options:



2. Choose *Close*, to progress past the welcome alert:



3. Enter the following command. Replace the **user-pool-id, client-id** 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=test
```

4. 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",
  "Session": "...",
  "ChallengeParameters": {
    "USER_ID_FOR_SRP": "testUser",
    "requiredAttributes": "[]",
    "userAttributes": "{}"
  }
}
```

> Make note of the Session value from the response above.

5. Use the CLI to set the final password for the user. Replace the pool ID, client ID 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 --challeng
```

> **Warning**
> Be aware when changing variables to use the correct double quotes symbol " and not “, these may cause incorrect responses when running the cli command

> You may get an invalid session response if run the above command more than a minute or so after the command in step 2.
>
> *An error occurred (NotAuthorizedException) when calling the AdminRespondToAuthChallenge operation: Invalid session for the user, session is expired.*
>
> If you do run the command from step 2 again, copy the new Session value and rerun the command in step 3.

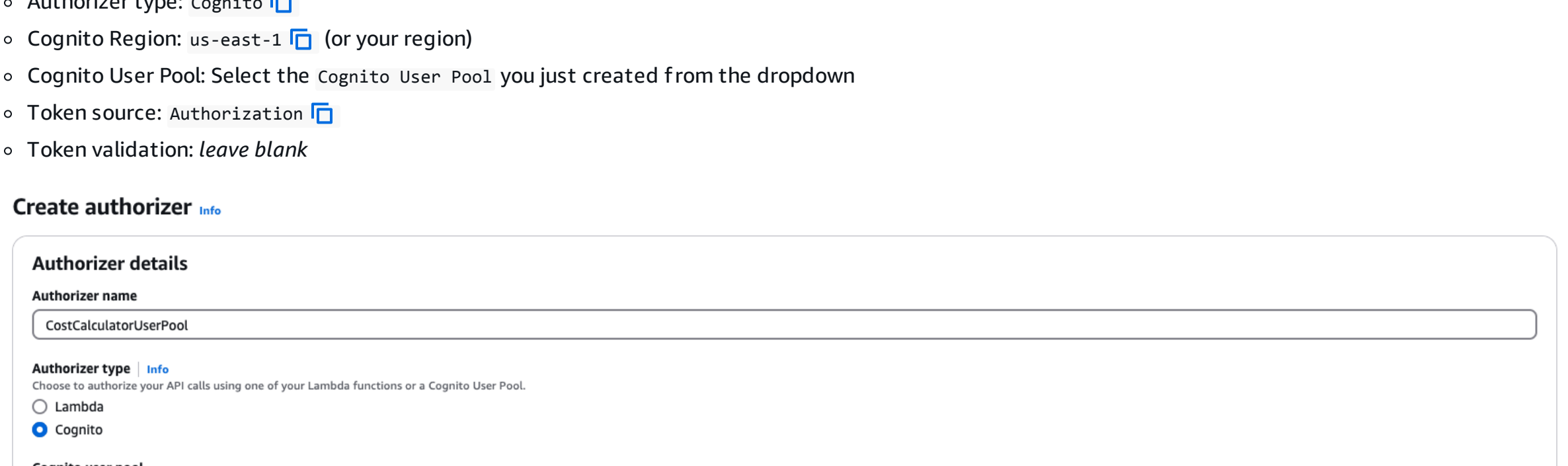
6. The output from this call should be a valid session

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

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 `RefreshToken`, you can use this token with the User Pool APIs to fetch a new identity and access token.

Create API Gateway Authorizer

1. Go back to API Gateway Service or go to https://console.aws.amazon.com/apigateway
2. Click on **calculatePrice** API
3. Click **Authorizers** on the navigation menu
4. Click **Create authorizer**
5. Enter the following values under *Authorizer details*:
   - Authorizer name: `CostCalculatorUserPool`
   - Authorizer type: `Cognito`
   - Cognito Region: `us-east-1` (or your region)
   - Cognito User Pool: Select the `Cognito User Pool` you just created from the dropdown
   - Token source: `Authorization`
   - Token validation: *leave blank*

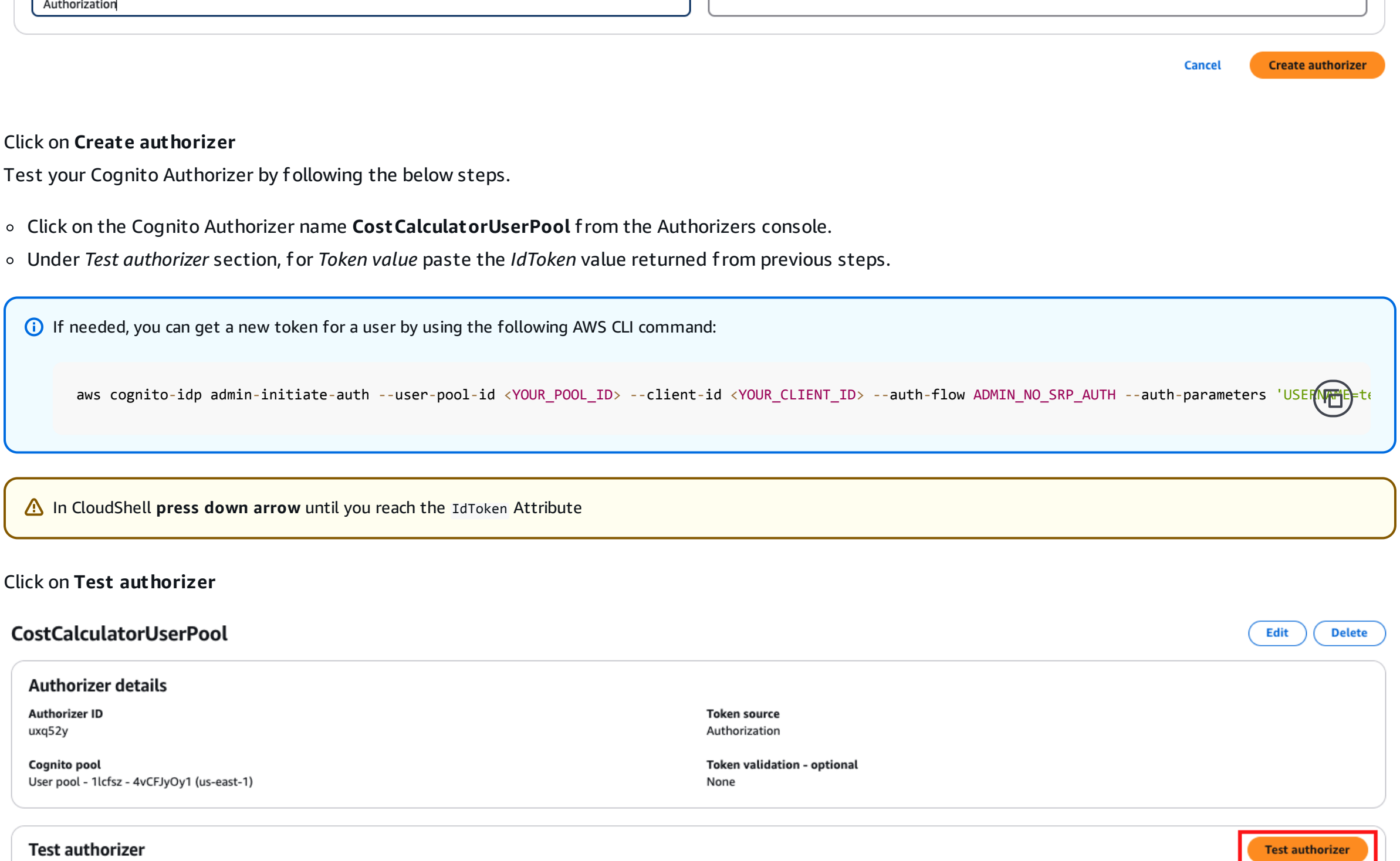


6. Click on **Create authorizer**
7. Test your Cognito Authorizer by following the below steps.
   - Click on the Cognito Authorizer name **CostCalculatorUserPool** from the Authorizers console.
   - Under *Test authorizer* section, for *Token value* paste the *IdToken* value returned from previous steps.

> If needed, you can 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 'USER
> ```

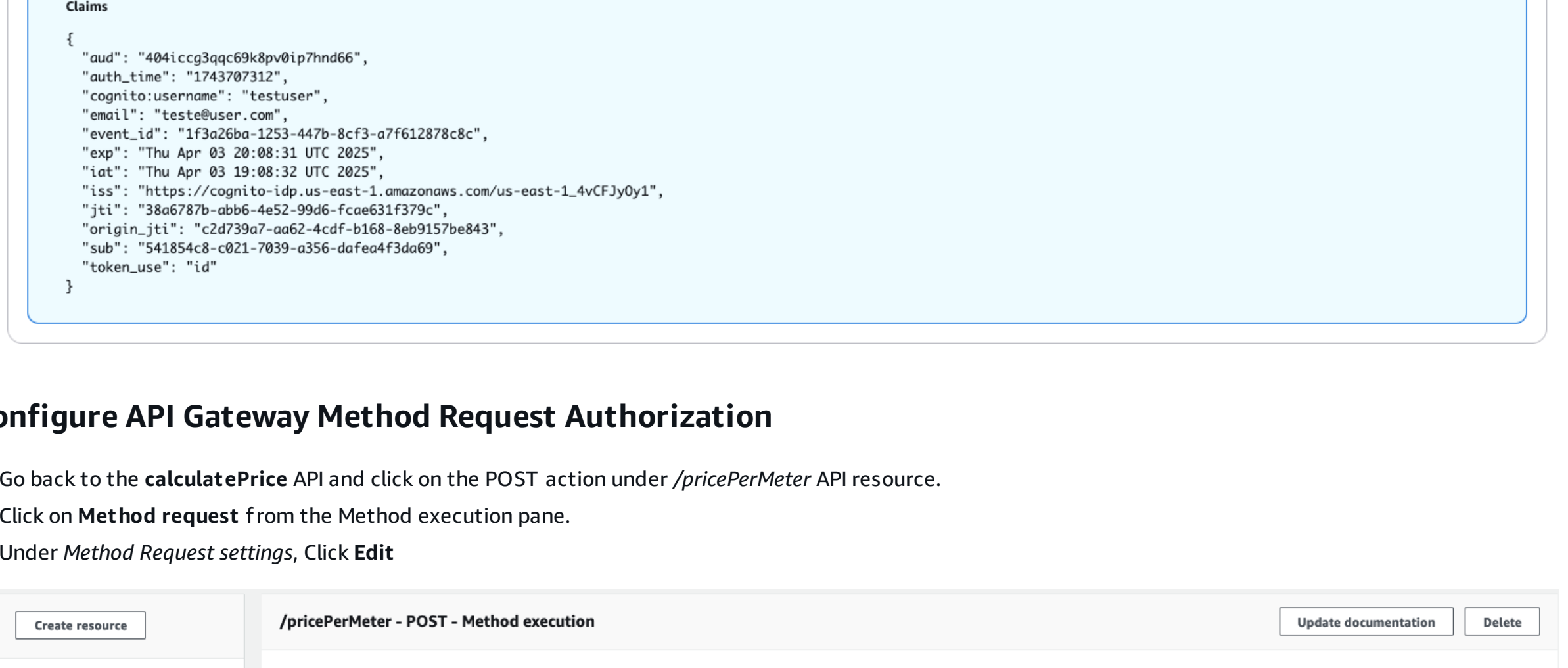
> In CloudShell **press down arrow** until you reach the `IdToken` Attribute

8. Click on **Test authorizer**



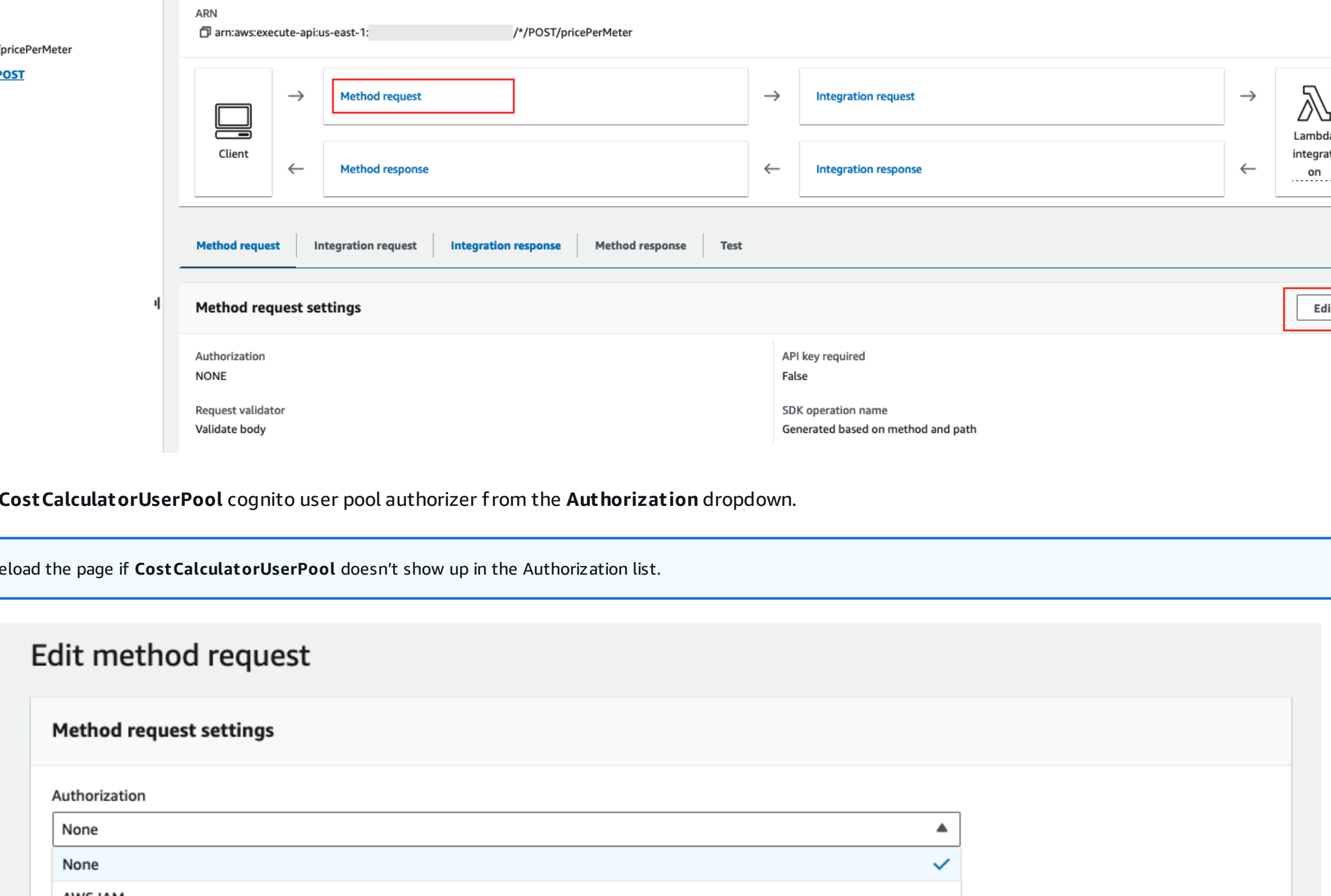
Configure API Gateway Method Request Authorization

1. Go back to the **calculatePrice** API and click on the POST action under */pricePerMeter* API resource.
2. Click on **Method request** from the Method execution pane.
3. Under *Method Request settings*, Click **Edit**



4. Select **CostCalculatorUserPool** cognito user pool authorizer from the **Authorization** dropdown.

> Reload the page if **CostCalculatorUserPool** doesn't show up in the Authorization list.



5. Click **Save**.

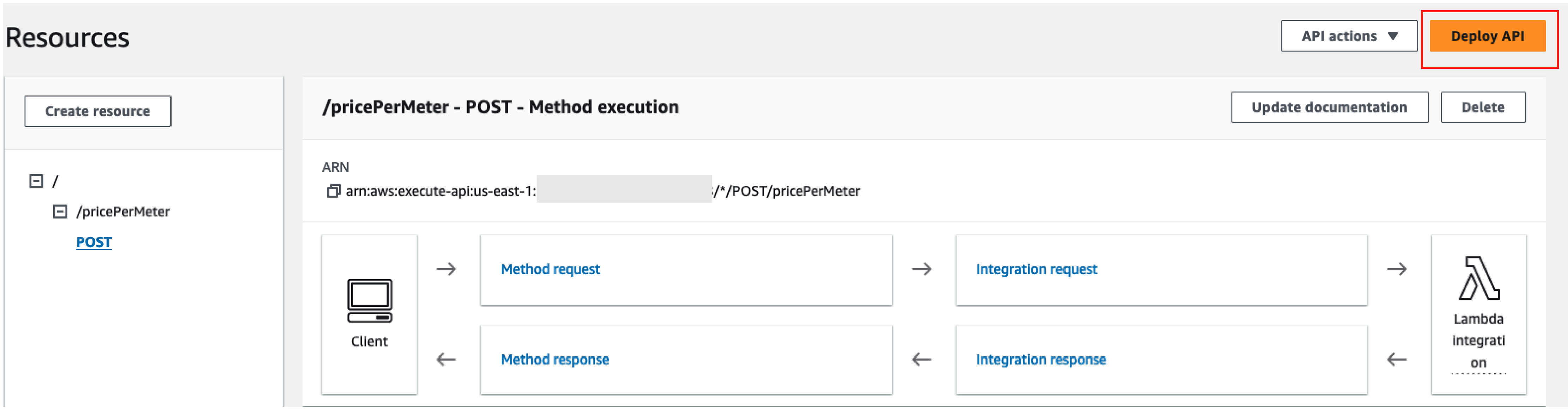
The authorization will be tested on the API in the next section.

The Amazon API Gateway Workshop > Module 1 - Introduction to Amazon API Gateway > API Deployment

API Deployment

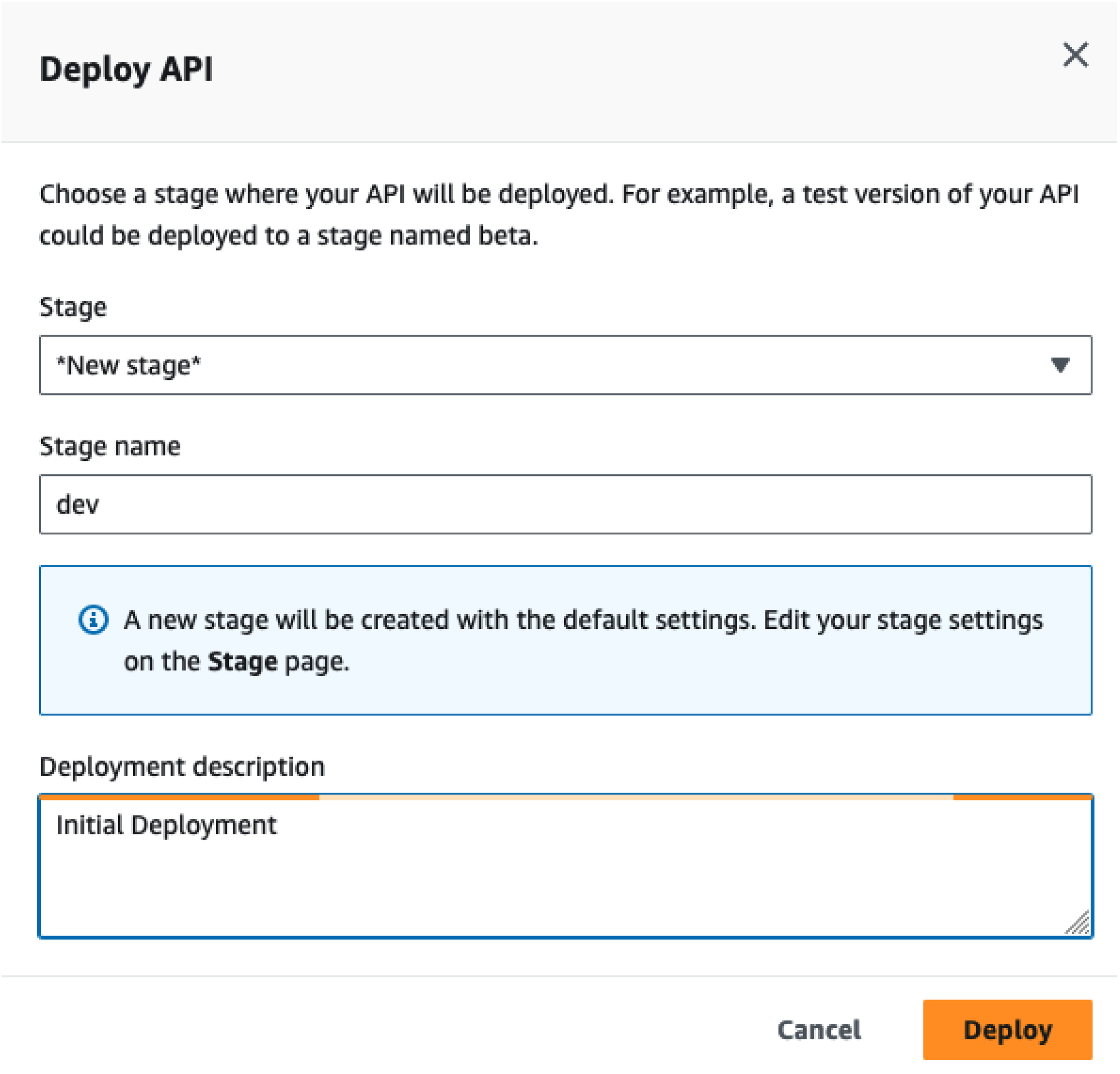
At this point, we have been testing our API using the API Gateway console. Now that we have made the necessary configuration for authorization, we are ready to deploy our API.

1. Select **calculatePrice** API and choose the *POST* method under *pricePerMeter* API resource.
2. Click on **Deploy API**.



3. Select **New stage** from the Stage drop down
4. Enter the following values:

- Stage name: `dev`
- Deployment description: `Initial Deployment`



5. Click **Deploy**
6. Expand the **dev** stage, and click on the *POST* method of the *pricePerMeter* API resource.

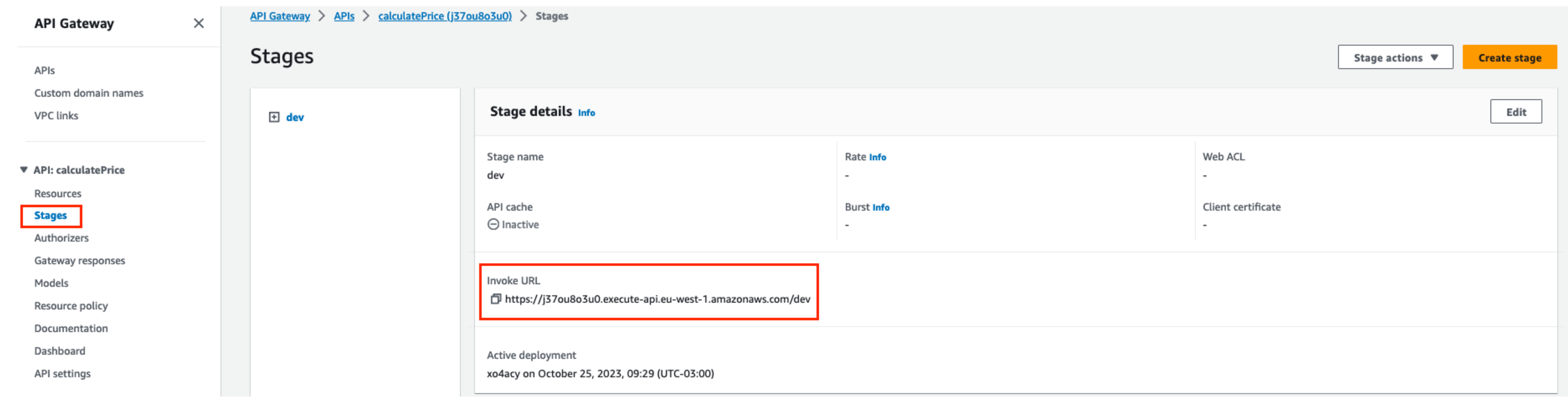
Note the API/Invoke URL. Copy this value as it will be used in the subsequent steps.

Test the API

After we finish the configuration of the Cognito User Pool as the authorizer, it's time to test it.

Let's test the API request using curl.

1. Go to **Stages** and copy the **Invoke URL**:



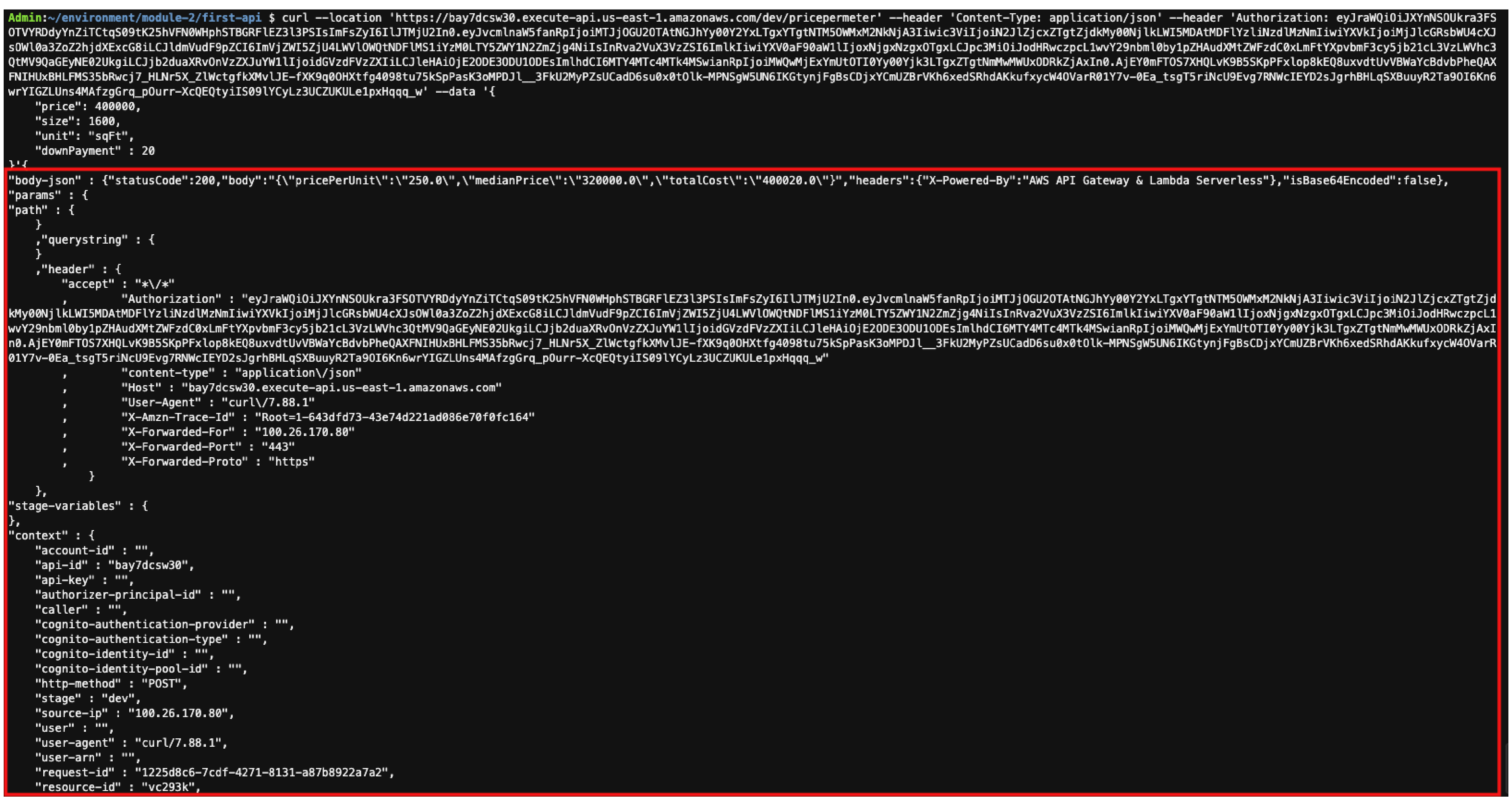
2. Navigate to Cloud9 in your AWS console or to AWS CloudShell and run the following command to execute the `/pricePerMeter` method:

```
curl --location '[insert your invoke URL here]/pricePerMeter' \
--header 'Content-Type: application/json' \
--header 'Authorization: [insert your IdToken here]' \
--data '{
"price": 400000,
"size": 1600,
"unit": "sqFt",
"downPayment" : 20
}'
```

The `IdToken` can be found and the previous section: Authentication and Authorization with Cognito.

Example
curl --location 'https://bay7dcsw30.execute-api.us-east-1.amazonaws.com/dev/pricepermeter'
--header 'Content-Type: application/json'
--header 'Authorization:
eyJraWQiOiJXYnNSOUkra3FSOTVYRDdyYnZiTCtqS09tK25hVFN0WHphSTBGRFlEZ3l3PSIsImFsZyI6IlJTMjU2In0.eyJvcmlnaW5fanRpIjoiMTJjOGU2OTAtNGJhYy00Y2YxLTgxYTgtNTM5
OWMxM2NkNjA3Iiwic3ViIjoiN2JlZjcxZTgtZjdkMy00NjlkLWI5MDAtMDFlYzliNzdlMzNmIiwiYXVkIjoiMjJlcGRsbWU4cXJsOWl0a3ZoZ2hjdXExcG8iLCJldmVudF9pZCI6ImVjZWI5ZjU4LWVlO
WQtNDFlMS1iYzM0LTY5ZWY1N2ZmZjg4NiIsInRva2VuX3VzZSI6ImlkIiwiYXV0aF90aW1lIjoxNjgxNzgxOTgxLCJpc3MiOiJodHRwczpcL1wvY29nbml0by1pZHAudXMtZWFzdC0xLmFtYXp
vbmF3cy5jb21cL3VzLWVhc3QtMV9QaGEyNE02UkgiLCJjb2duaXRvOnVzZXJuYW1lIjoidGVzdFVzZXIiLCJleHAiOjE2ODE3ODU1ODEsImlhdCI6MTY4MTc4MTk4MSwianRpIjoiMWQwMjExY
mUtOTI0Yy00Yjk3LTgxZTgtNmMwMWUxODRkZjAxIn0.AjEY0mFTOS7XHQLvK9B5SKpPFxlop8kEQ8uxvdtUvVBWaYcBdvbPheQAXFNIHUxBHLFMS35bRwcj7_HLNr5X_ZlWctgfkXMvlJE-
fXK9q0OHXtfg4098tu75kSpPasK3oMPDJl__3FkU2MyPZsUCadD6su0x0tOlk-MPNSgW5UN6IKGtynjFgBsCDjxYCmUZBrVKh6xedSRhdAKkufxycW4OVarR01Y7v-
0Ea_tsgT5riNcU9Evg7RNWcIEYD2sJgrhBHLqSXBuuyR2Ta9OI6Kn6wrYIGZLUns4MAfzgGrq_pOurr-XcQEQtyiIS09lYCyLz3UCZUKULe1pxHqqq_w'
--data '{ "price": 400000, "size": 1600, "unit": "sqFt", "downPayment" : 20 }'

After the request you should receive a response like this:



Notice that you were able to make the request to the API only because you sent a valid authentication token to the `/pricepermeter` method. Try to perform the request again without sending the token in the Authorization header to check that the response will be **401 - Unauthorized**

The Amazon API Gateway Workshop

- Introduction
- Getting Started
- Module 1 - Introduction to Amazon API Gateway
  - Module Goals
  - Create Your First API
  - Message Transformation
  - Request Validation
  - Authentication and Authorization with Cognito
  - API Deployment
  - Message Caching
  - Usage Plans and Message Throttling
  - Authentication and Authorization with IAM
  - Clean Up
- Module 2 - Deploy your first API with IaC
- Module 3 - API Gateway REST Integrations
- Module 4 - Observability in API Gateway
- Module 5 - WebSocket APIs
- Module 6 - Enable fine-grained access control for your APIs
- Clean up
- Resources

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 backend and also improve the latency of the requests to your API. When you enable caching for a stage, API Gateway caches responses from your backend 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.

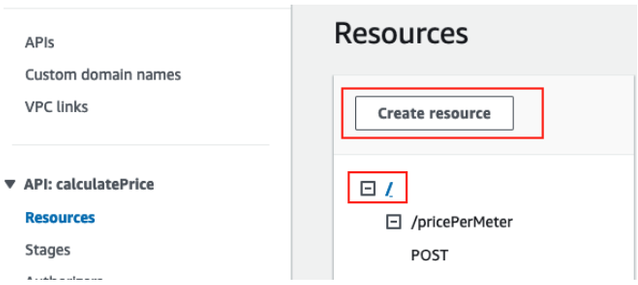
API Gateway enables caching at the stage or method level. In this lab, we will create a new resource called **medianPriceCalculator**. This service returns the median price of houses in US or Canada. Since these regional prices do not change often, we can cache the results.

In this section we will:

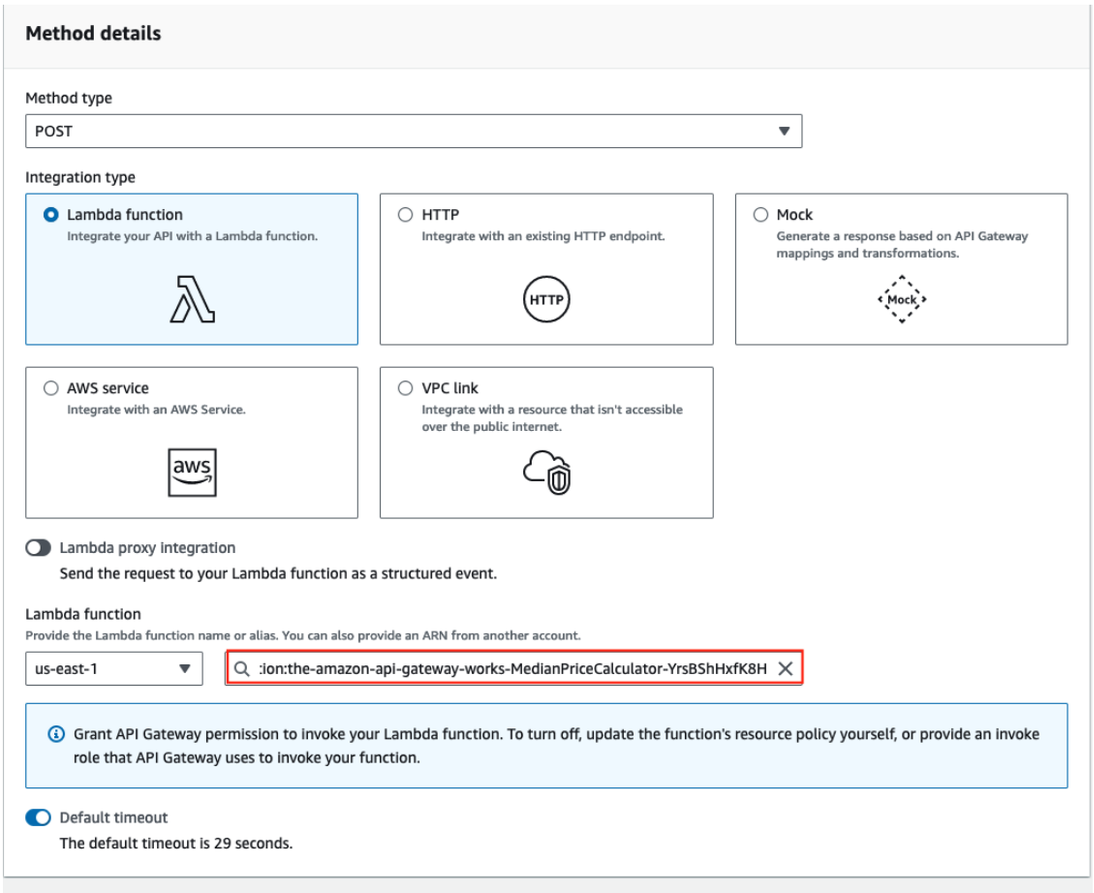
- **Create a new resource**
- **Enable Caching**
- **Deploy the API**
- **Testing Your Cached Resource**

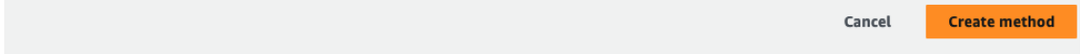
Create a new resource

1. From **API Gateway Console**, select the root element (/) of **calculatePrice** API
2. Click **Create resource**



3. Enter the following values
   - Resource Path: Choose default (/)
   - Resource Name: `medianPriceCalculator`
4. Click **Create resource**
5. Under **Methods** tab, click **Create method**
6. Provide the following values under **Method details**.
   - Method type: Choose **POST**
   - Integration Type: `Lambda function`
   - Lambda Region: `us-east-1` (or your chosen region)
   - Lambda Function: `-MedianPriceCalculator-` (Select the function which comes up.)





7. Click **Create method**

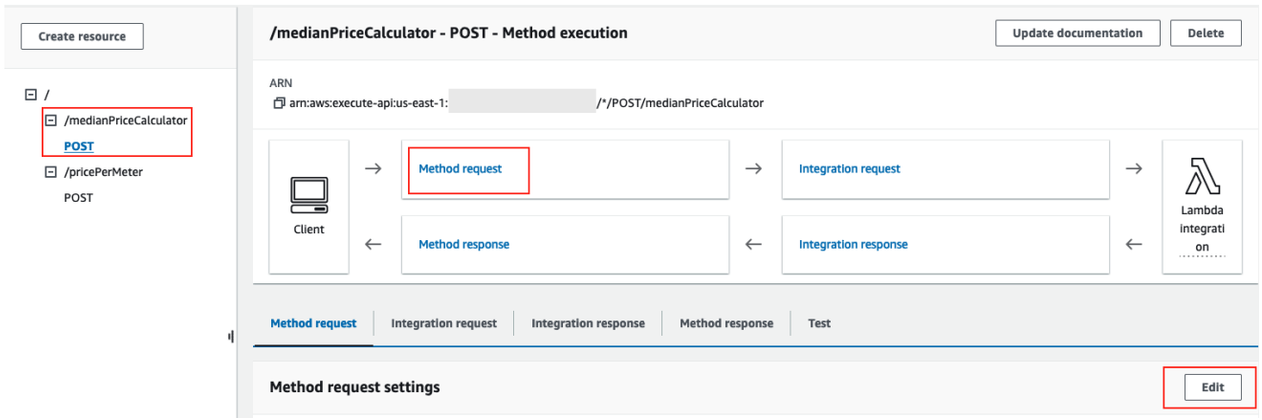
> **Notice**
> By default, API Gateway is granted permission to invoke the configured Lambda function.

Enable Caching

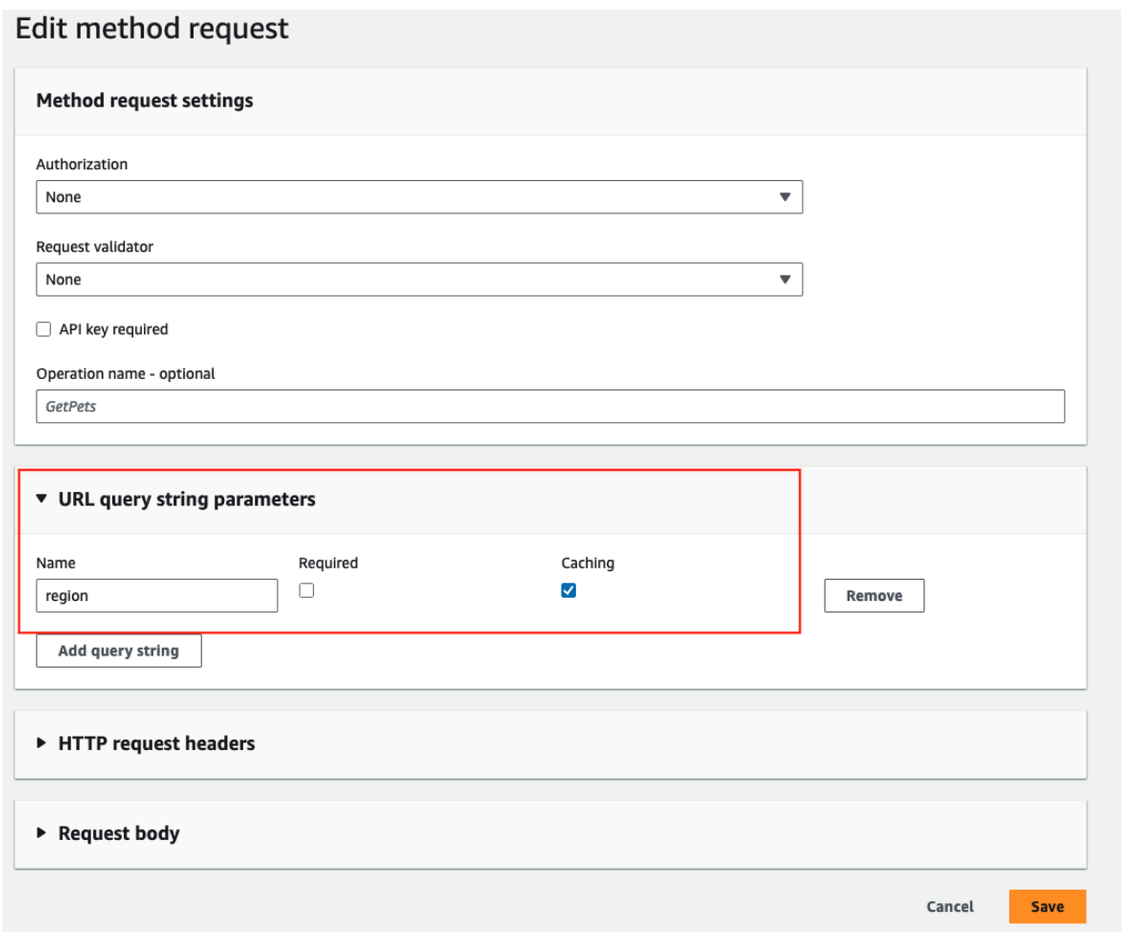
Now that the API is created, we can enable caching at the method level. Since the median price for Canada and US are different, it would make sense to cache based on the chosen region. When a cached method or integration has parameters, which can take the form of `custom headers`, `URL paths`, or `query 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 on **POST** method of the **medianPriceCalculator** API resource.
2. Click on **Method request** from the *Method Execution* pane.
3. Under *Method request settings*, click on **Edit**

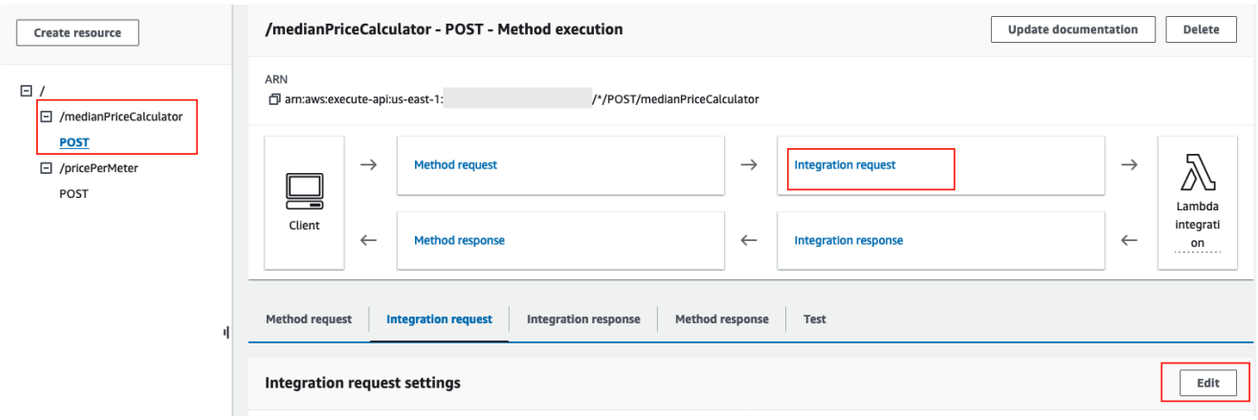


4. Expand **URL query string parameters** section and perform below steps.
   - Click **Add query string**
   - Type `region` for the *Name*
   - Check **Caching**
5. Click **Save**

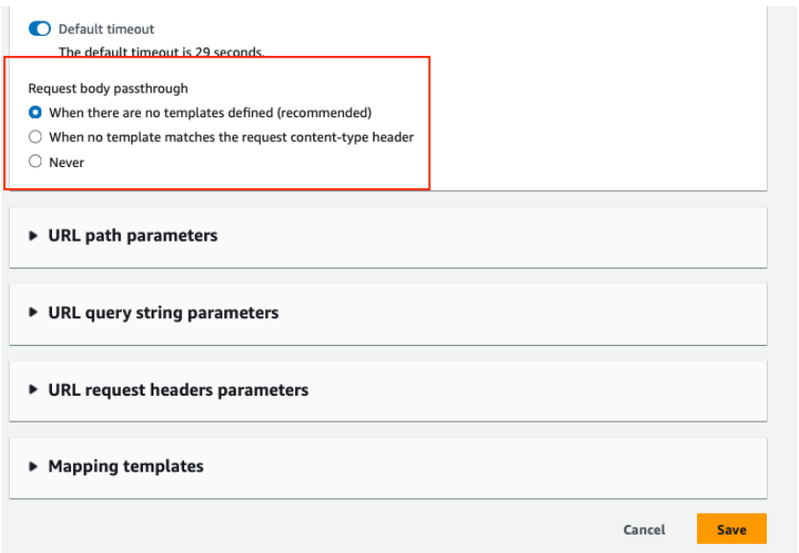


Now that we are passing the region as a query parameter, we need to map it to the integration payload.

1. Click on **POST** method of the **medianPriceCalculator** API resource.
2. Click on **Integration request** from the *Method Execution* pane.
3. Under *Integration request settings*, click on **Edit**



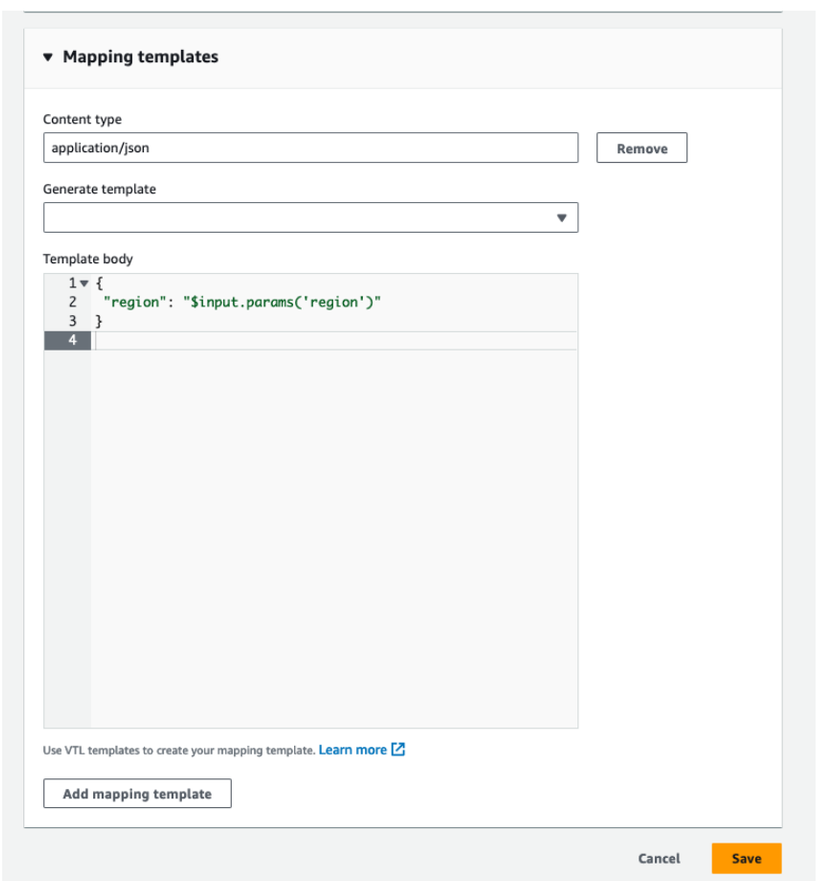
4. Select *Request body passthrough* as **When there are no templates defined (recommended)**



5. Scroll down to *Mapping templates* section. Click **Add mapping template**
6. Provide the following values under Mapping Templates.

- *Content type*: Type `application/json`
- *Generate template*: Leave blank
- Copy and Paste the following content in the **Template body** section

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



7. Click **Save**

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

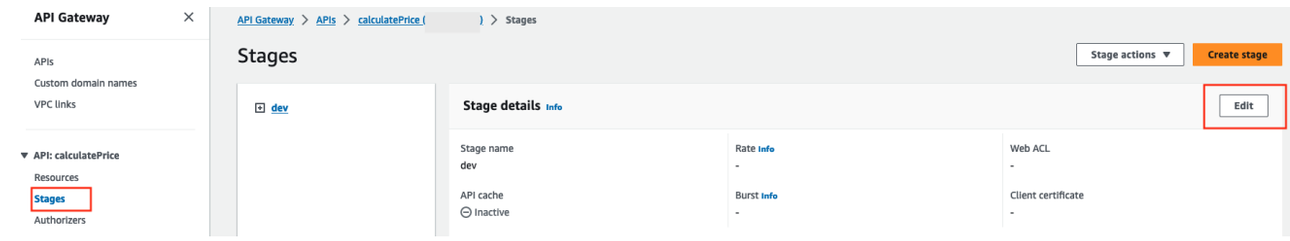
Deploy the API

Now, it's time to deploy our new API.

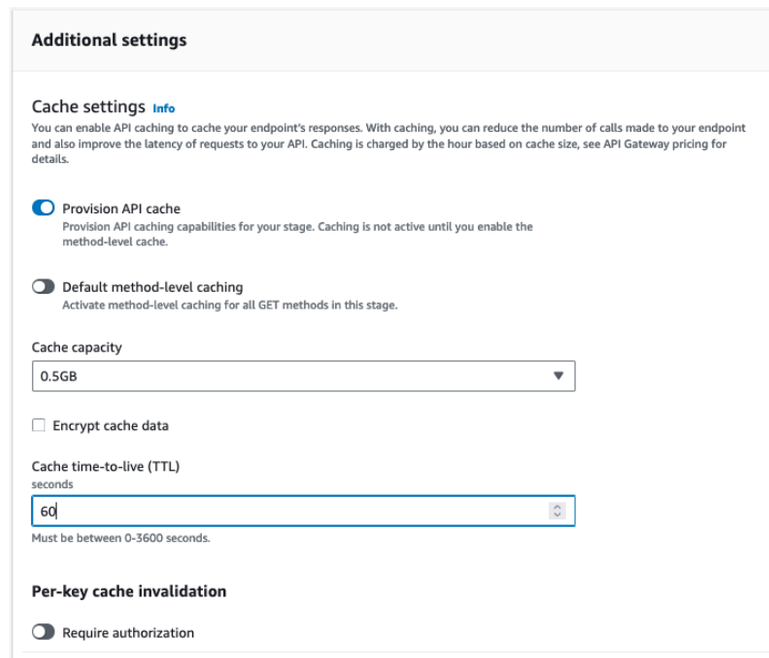
1. Click on **POST** method of the **medianPriceCalculator** API resource
2. Click on **Deploy API**
3. Choose **dev** stage
4. Enter Deployment description `Add new method with caching` .
5. Click **Deploy**

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

1. Click **Stages** under the **calculatePrice** API navigation menu.
2. Click on the **dev** stage.
3. Click **Edit** across *Stage details* section.



4. Enable **API cache** under *Cache settings* and choose the below values.
   - Cache capacity: `0.5GB`
   - Cache time-to-live (TTL): `60`
   - Leave all other fields as default



5. Click **Save**

Notice that the API cache status changes from `InActive` to `Active` on the Stage details section.

> **Important**
> Creating or deleting a cache takes a few minutes for API Gateway to complete. You can monitor the cache creation status on the stage page.

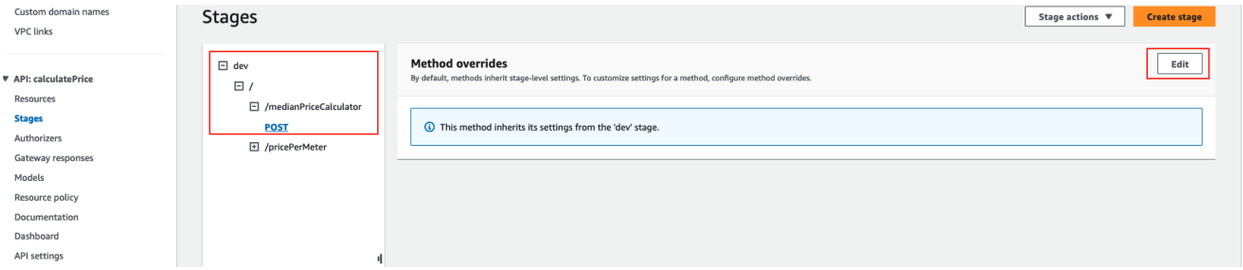
> **Important**
> When you enable caching for a stage, only **GET** methods have caching enabled by **default**. This helps to ensure the safety and availability of your API. You can enable caching for other methods by **overriding** method settings.

Overriding Method Settings

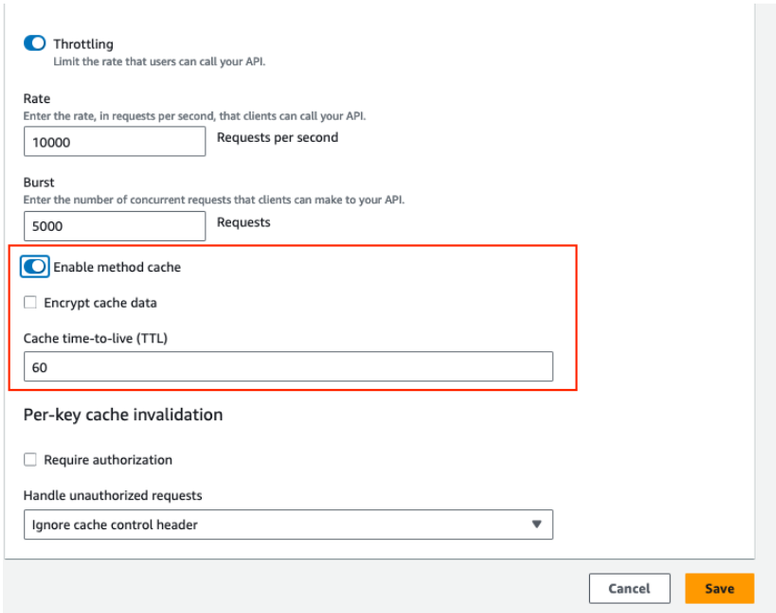
Since we enabled caching at the stage, it **will not automatically apply to POST methods deployed under the stage**. We need to **override API Gateway stage-level caching** for enable method caching.

> You will need to wait a few minutes for the Cache status to become **Active** before continuing.

1. Expand **dev** stage, click on **POST** method under **medianPriceCalculator** API resource.
2. Click on **Edit** across *Method overrides* section.



3. Toggle **Enable method cache** (this is disabled by default)
4. Set *Cache time-to-live (TTL)* as `60`



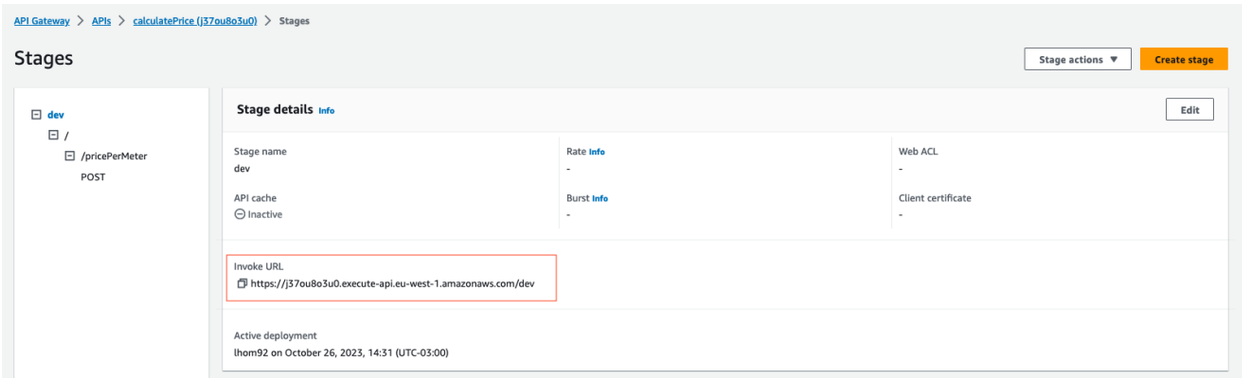
5. Click **Save**

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`.

Let's test the API request using curl.

1. Go to the **medianPriceCalculator** *POST* method from the *dev* stage and copy the **Invoke URL**.



2. Navigate to Cloud9 in your AWS console or to AWS CloudShell and run the following command to execute the **/medianPriceCalculator** method:

```
curl --location '[insert your invoke URL here]/medianPriceCalculator?region=US' \
--header 'Content-Type: application/json' \
--data '{}' \
-w '\n===\nTotal time: %{time_total}s\n'
```

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. You will notice that the first invocation takes slightly longer than subsequent invocations.

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

Congratulations! You have successfully completed this module.

Select your cookie preferences

We use essential cookies and similar tools that are necessary to provide our site and services. We use performance cookies to collect anonymous statistics, so we can understand how customers use our site and make improvements. Essential cookies cannot be deactivated, but you can choose “Customize” or “Decline” to decline performance cookies.

If you agree, AWS and approved third parties will also use cookies to provide useful site features, remember your preferences, and display relevant content, including relevant advertising. To accept or decline all non-essential cookies, choose “Accept” or “Decline.” To make more detailed choices, choose “Customize.”

Accept
Decline
Customize

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 quotas.

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.

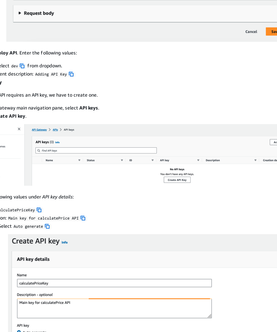
In the subsequent sections of the lab, we are going to setup API Keys to track our API Callers. The API keys will then be used to setup usage plans and restrict each client based on their tier level (platinum, gold, silver, etc.).

In this section we will:

- **Setting up API Keys**
- **Setting Up Usage Plans**
- **Testing API with Usage Plan**

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, select the *calculatePrice* API by clicking on the name
3. Then click the **POST** method of **medianPriceCalculator** API resource
4. Click **Method request** from *Method execution* pane.
5. Under the *Method request settings* section, click **Edit**
6. Select the checkbox for **API key required**
7. Click **Save**

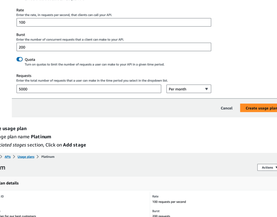


8. Click on **Deploy API**. Enter the following values:
   - Stage: Select `dev` from dropdown.
   - Deployment description: `Adding API key`
9. Click **Deploy**

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

1. In the *API Gateway* main navigation pane, select **API keys**.
2. Click on **Create API key**.

3. Provide following values under *API key details*:
   - Name: `calculatePriceKey`
   - Description: `Main key for calculatePrice API`
   - API key: Select `Auto generate`



4. Click **Save**

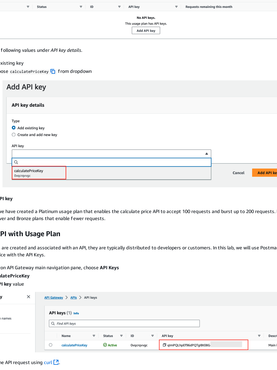
> You may need to click on **APIs** and select the *calculatePrice* API to get back the full navigation menu.

Setting Up Usage Plans

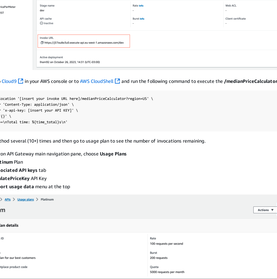
Once the API keys are created, the next step is to create a Usage Plan.

1. In the *Amazon API Gateway* main navigation pane, Select **Usage plans**
2. Click **Create usage plan**.

3. Provide the following values under *Usage plan details*:
   - Name: `Platinum`
   - Description: `Exclusive plan for our best customers`
   - Toggle Throttling to enable
   - Throttling > Rate: `100`
   - Throttling > Burst: `200`
   - Toggle Quota to enable
   - Quota: `5000` requests **Per month**



4. Click **Create usage plan**
5. Click on *Usage plan* name **Platinum**
6. Under *Associated stages* section, Click on **Add stage**



7. Select **calculatePrice** API from drop down
8. Select **dev** stage
9. Expand **Method-level throttling** section and click on **Add method**. Use the following values.
   - Resource: `/medianPriceCalculator`
   - Method: `POST`
   - Rate: `100`
   - Burst: `200`

10. Click **Add to usage plan**
11. Click on **Associated API keys** tab
12. Click **Add API key**

13. Provide the following values under *API key details*.
   - Type: Add existing key
   - API key: Choose `calculatePriceKey` from dropdown

14. Click **Add API key**

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. Copy the **API key** value

Let's test the API request using curl

4. Go to the **medianPriceCalculator** *POST* method from the *dev* stage and copy the **Invoke URL.**

5. Navigate to Cloud9 in your AWS console or to AWS Cloud9 Ide and run the following command to execute the */medianPriceCalculator* method:

```
curl --location '[insert your invoke URL here]/medianPriceCalculator?region=US' \
--header 'Content-Type: application/json' \
--header 'x-api-key: [insert your API KEY]' \
--data '{}' \
-w '\n==>Total time: %{time_total}\n'
```

Invoke the method several (10+) 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 **Associated API keys** tab
4. Select **calculatePriceKey** API Key
5. Click on **Export usage data** menu at the top

6. Notice the usage statistics.

> Usage data is delayed by up to an hour, so would suggest taking a break for an hour and coming back to check it.

7. Click on **Grant usage extension** to observe the number of requests that are still available for the API Key.

Authentication and Authorization with IAM

In this lab, we will create a new API in the API Gateway to be used internally with your session user, assuming a specific role with permission to consume this new API.

This lab updates our Amazon API Gateway to use IAM-based authorization. This extends our authorization capability to offer fine-grained access control authorizing differently per API operation and enhancing security via request signing. By enabling IAM-based authorization, you will use the same type of authentication, authorization, and request signing used by all AWS services and SDKs.

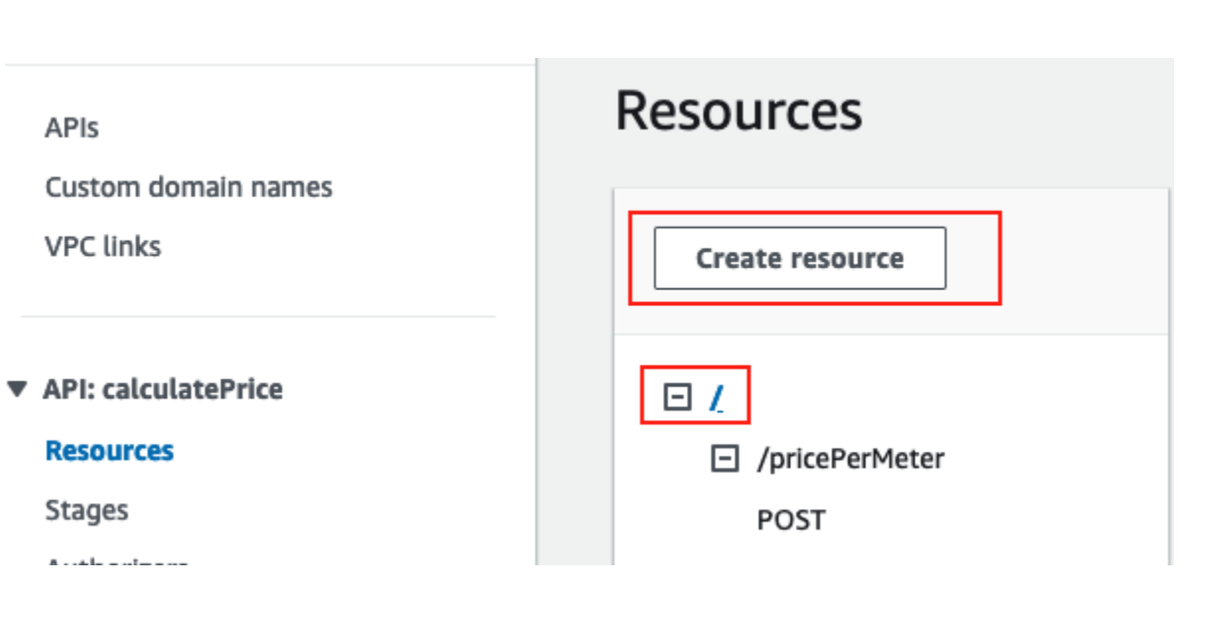
Request signing is a secure implementation of API request authentication where each API request made is signed with a signature unique to the request itself. Hence, no static API keys or bearer tokens are directly sent to the backend service and any man-in-the-middle attacks would not be able to use such API keys or bearer tokens to impersonate a valid user with the backend resources. AWS APIs and SDKs use a request signing algorithm named Signature V4 (Sigv4) which is what you will enable your API to use in this module.

Below are the steps to enable fine-grained IAM authorization in our calculatePrice API:

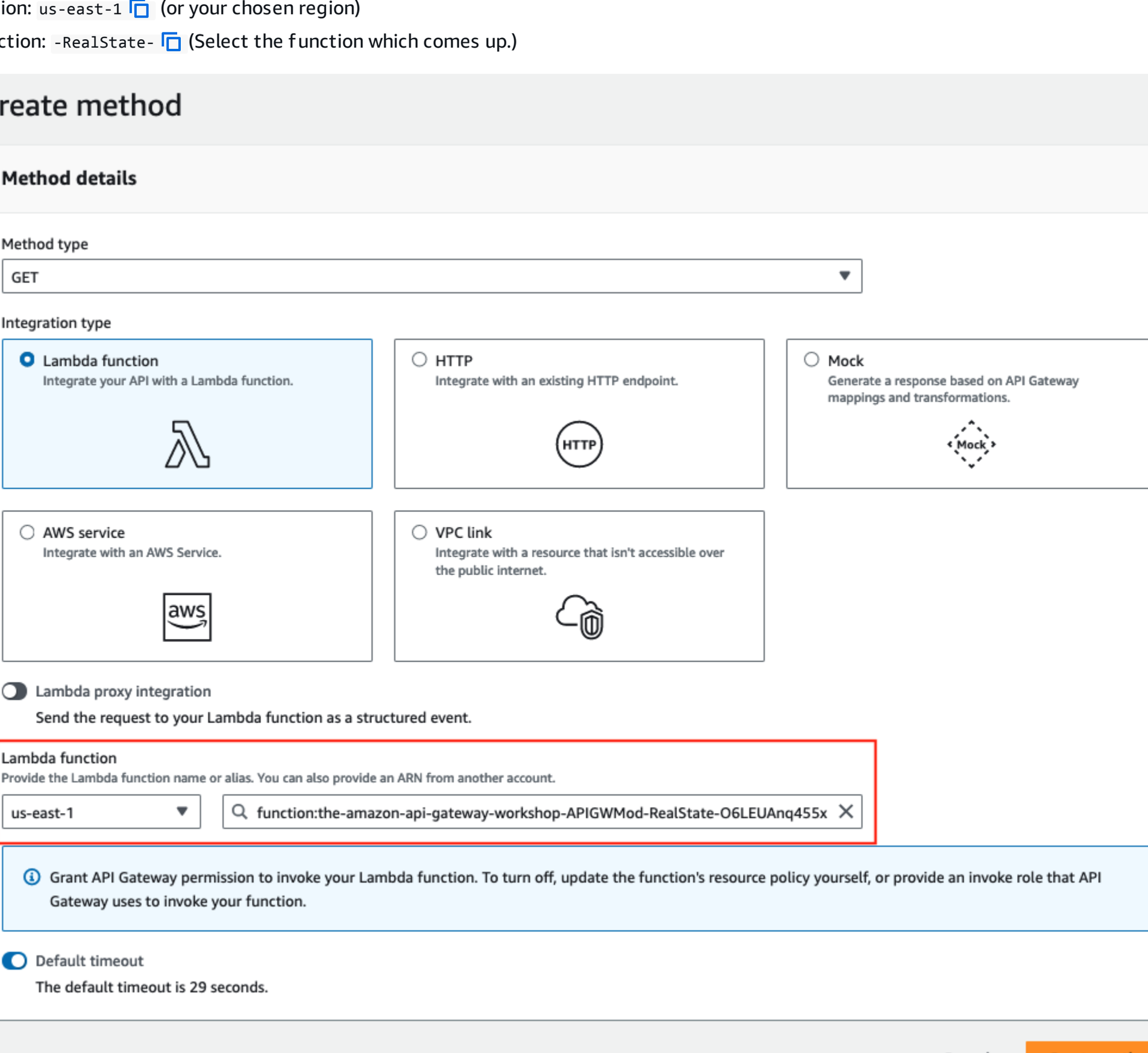
- **Create a new resource**
- **Associate an API Gateway with an account role**
- **Update API Gateway Authentication**
- **Testing the API**

Create a new resource

1. From **API Gateway Console**, select the root element (/) of *calculatePrice* API
2. Click **Create resource**



3. Enter the following values
   - Resource Path: Choose default (/)
   - Resource Name: `realState`
4. Click **Create resource**
5. Under **Methods** tab, click **Create method**
6. Choose the following values under **Method details**.
   - Method type: Choose **GET**
   - Integration Type: `Lambda function`
   - Lambda Region: `us-east-1` (or your chosen region)
   - Lambda Function: `-RealState-` (Select the function which comes up.)



7. Click **Create method**

> **Notice**
> By default, API Gateway is granted permission to invoke the configured Lambda function.

Associate an API Gateway with an account IAM role

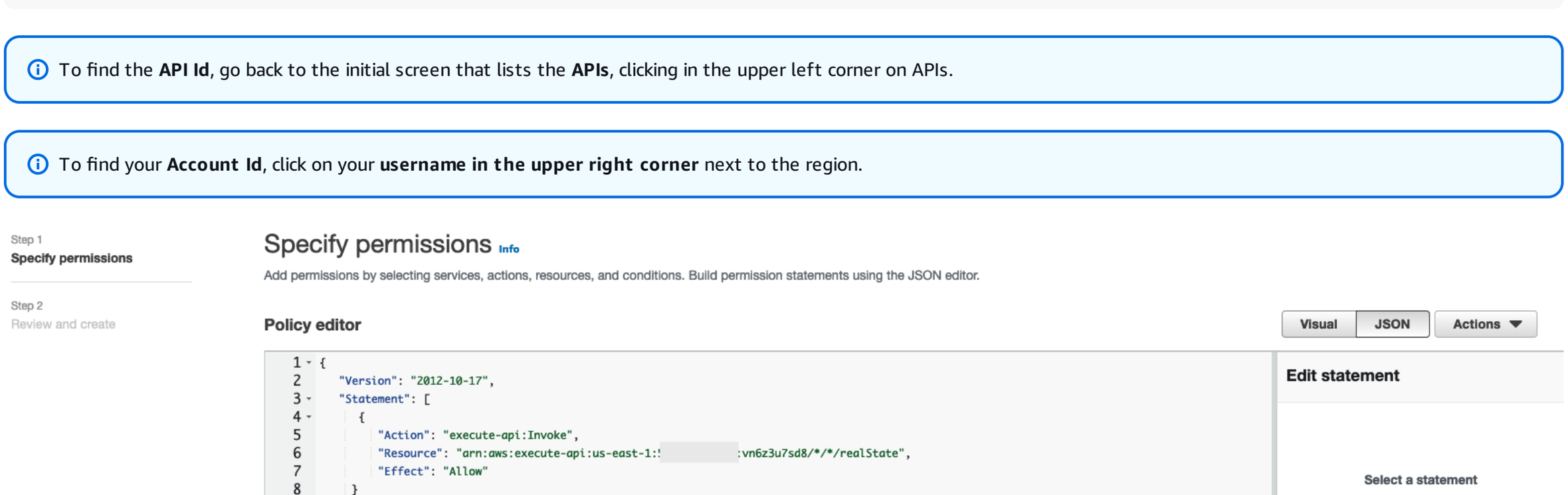
For us to be able to use request signing and IAM-based fine-grained access control, we'll first need to associate an IAM policy that provides permissions to invoke API operations for your API Gateway deployment. For further details, you can review controlling access to an API with IAM permissions documentation.

1. Go the AWS Management Console, click **Services** then select **IAM** under Security, Identity, and Compliance.
2. Choose **Policies**.
3. Click **Create policy**.
4. Select the **JSON** tab, then copy and paste the following policy into the box, replacing the values with the appropriate ones. Replace the **region, account_id, and api_gw_id** attributes:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": "execute-api:Invoke",
      "Resource": "arn:aws:execute-api:<REGION>:<ACCOUNT_ID>:<API_GW_ID>/*/*/realState",
      "Effect": "Allow"
    }
  ]
}
```

> To find the **API Id**, go back to the initial screen that lists the **APIs**, clicking in the upper left corner on APIs.

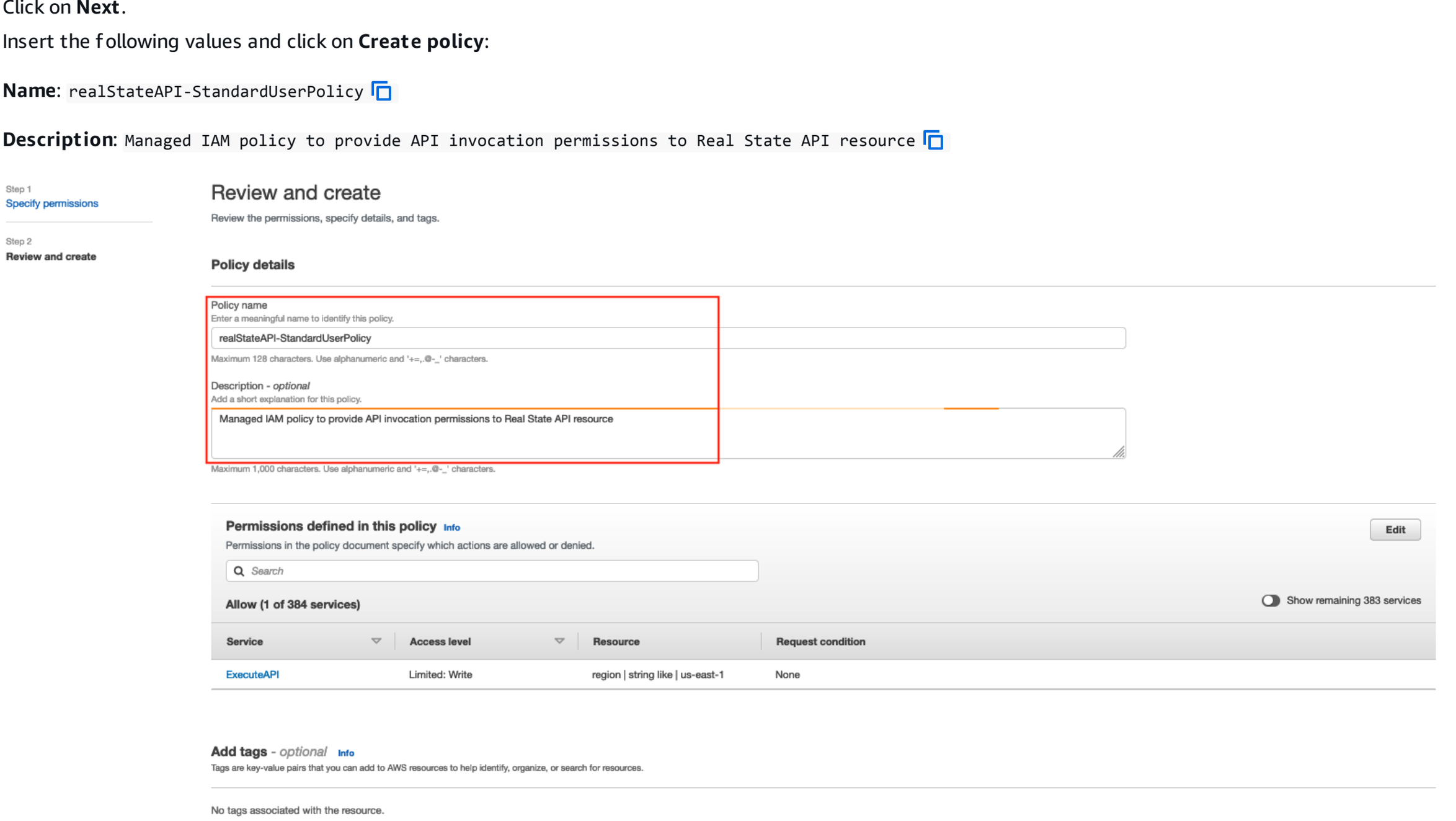
> To find your **Account Id**, click on your **username in the upper right corner** next to the region.



5. Click on **Next**.
6. Insert the following values and click on **Create policy**:

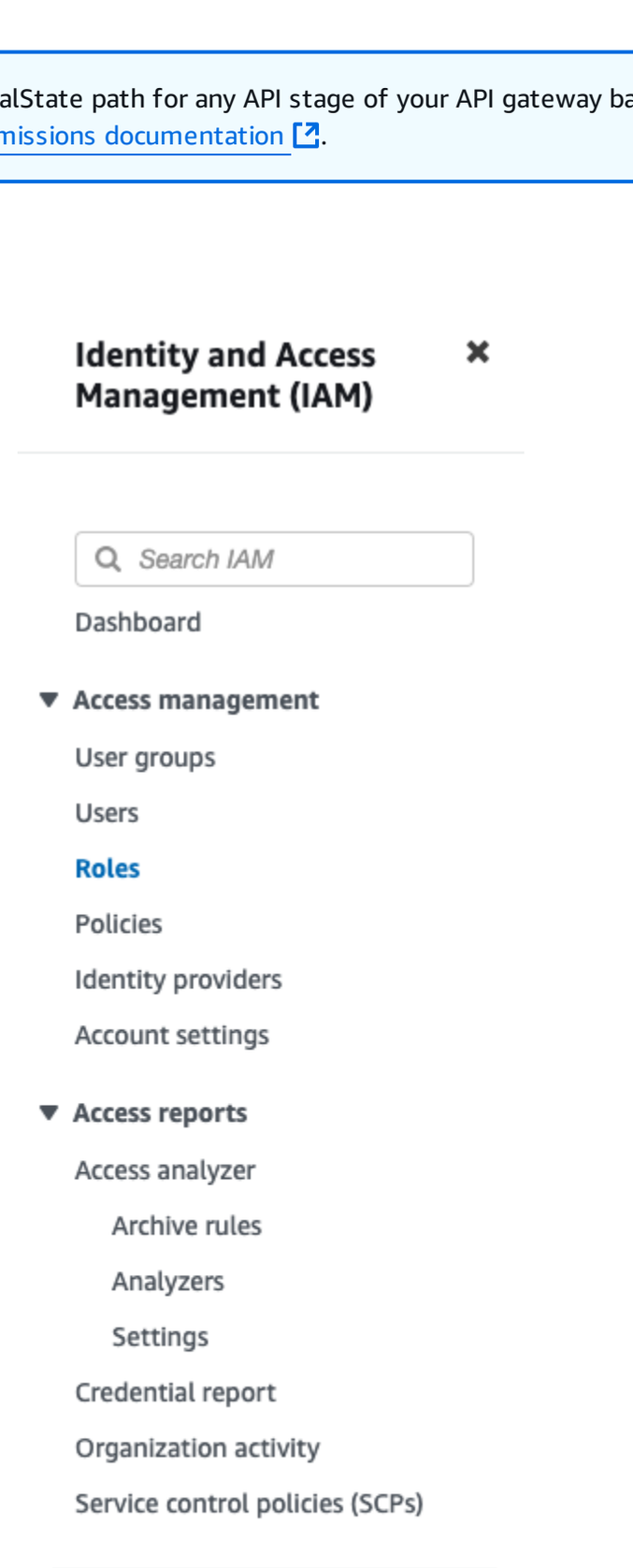
   Name: `realStateAPI-StandardUserPolicy`

   Description: `Managed IAM policy to provide API invocation permissions to Real State API resource`

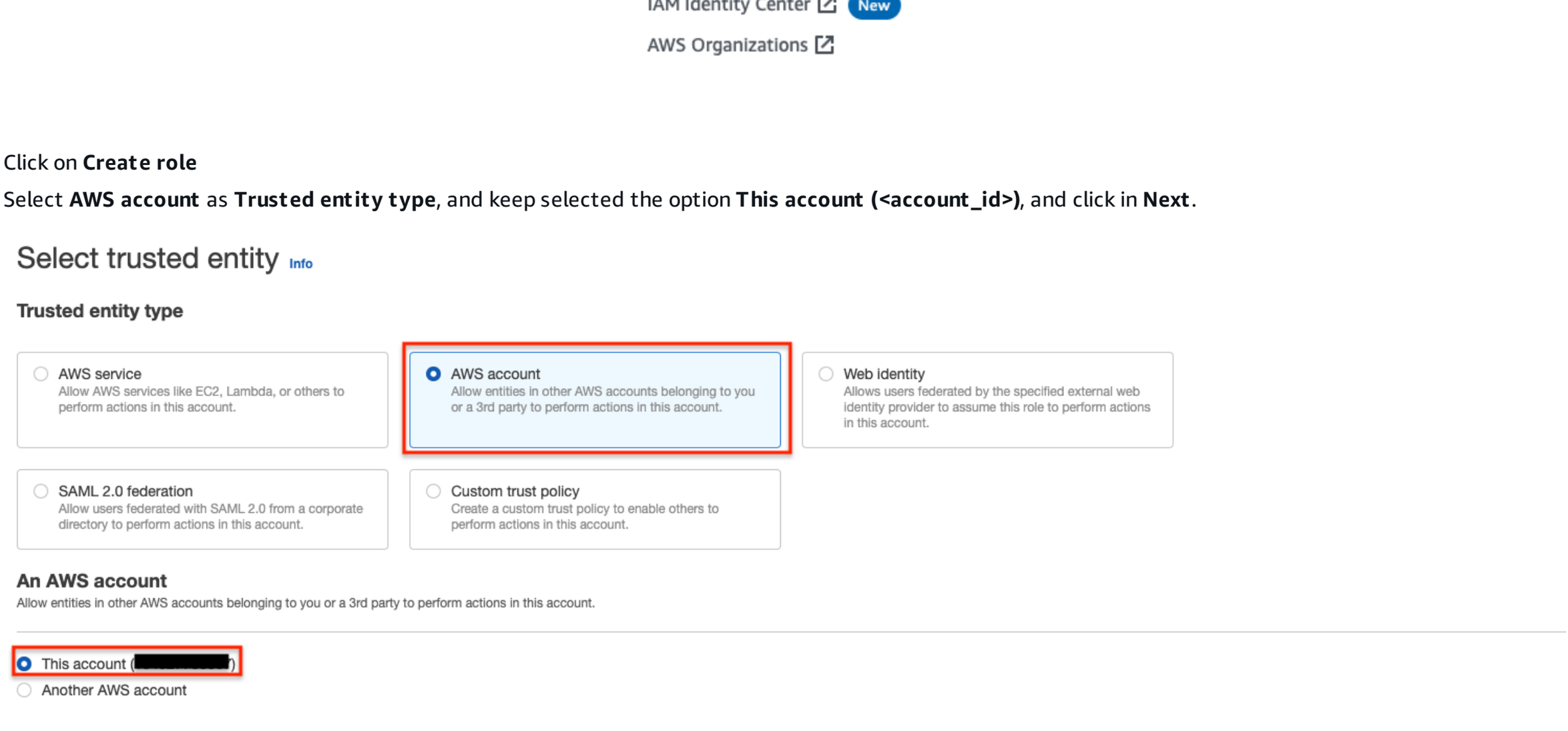


> **Info** This policy allows access to invoke any method on the /realState path for any API stage of your API gateway backend. For more details about authoring IAM policies for API Gateway, visit the controlling access to an API with IAM permissions documentation.

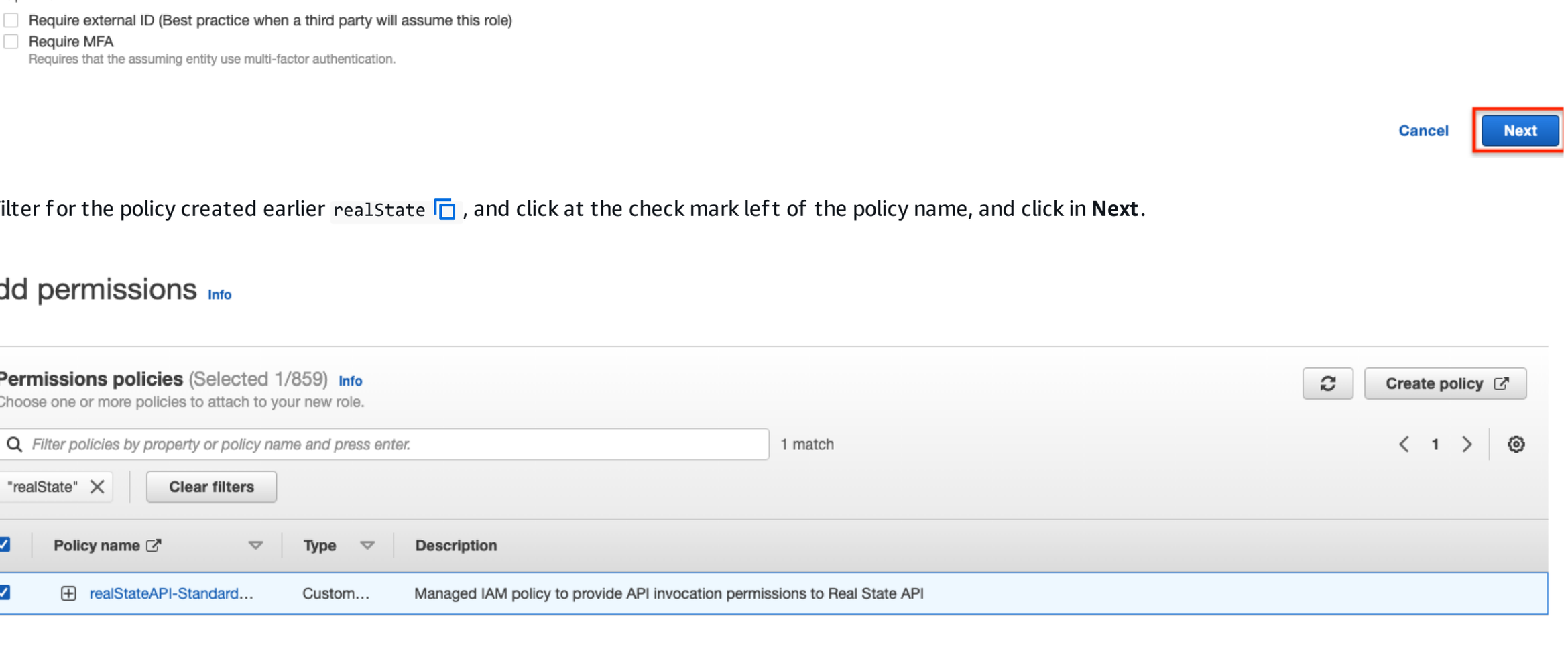
7. Choose **Role** in the left menu.



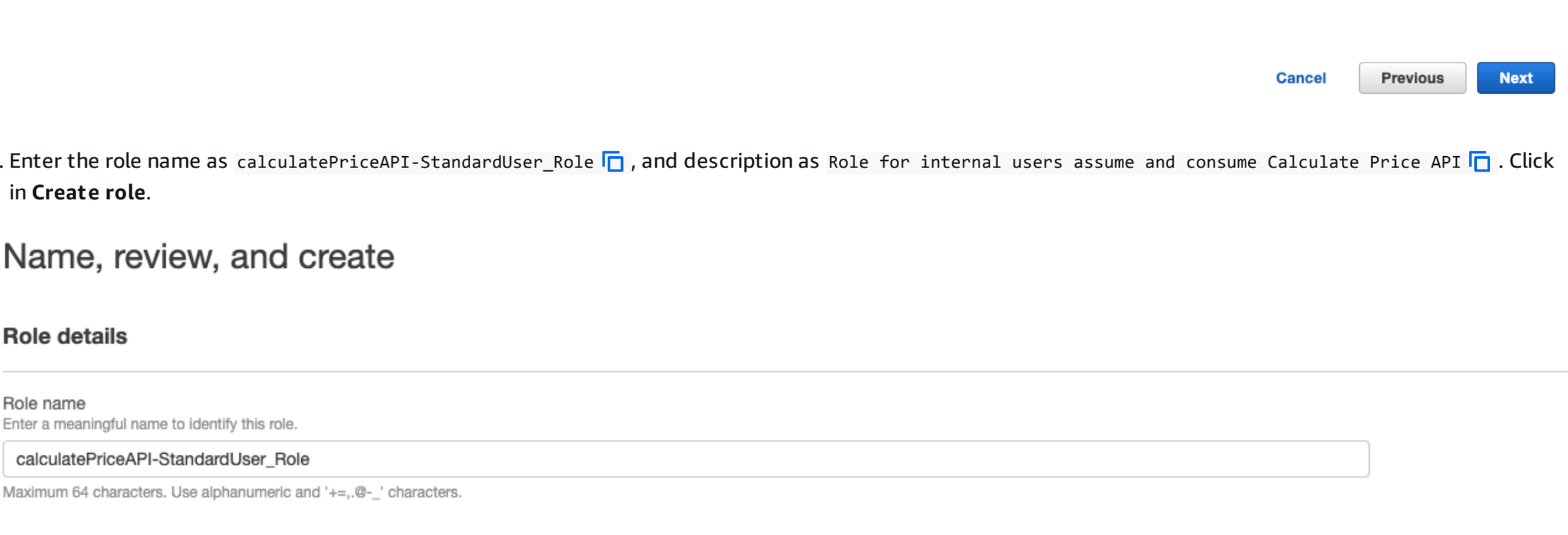
8. Click on **Create role**
9. Select **AWS account** as **Trusted entity type**, and keep selected the option **This account (<account_id>)**, and click in **Next**.



10. Filter for the policy created earlier `realState`, and click at the check mark left of the policy name, and click in **Next**.

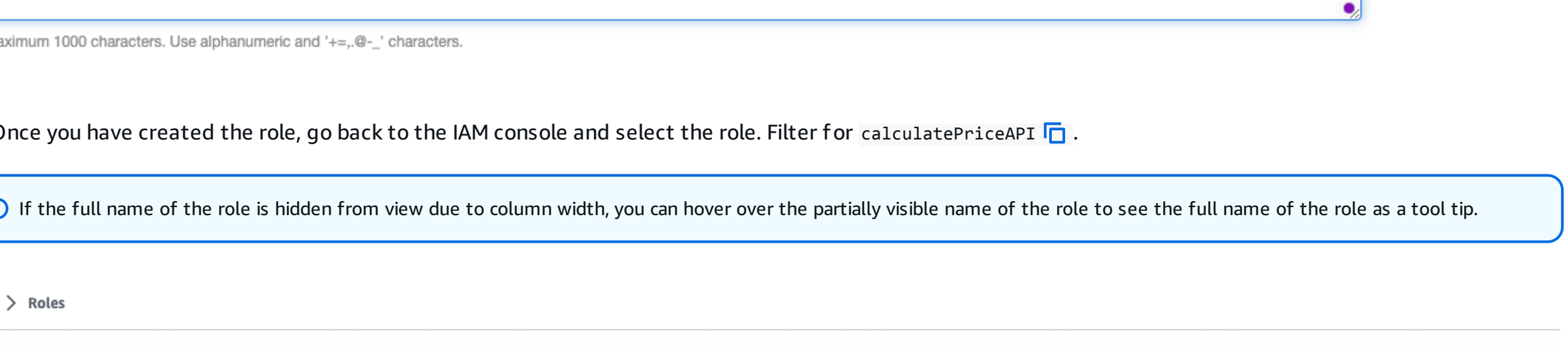


11. Enter the role name as `calculatePriceAPI-StandardUser_Role`, and description as `Role for internal users assume and consume Calculate Price API`. Click in **Create role**.

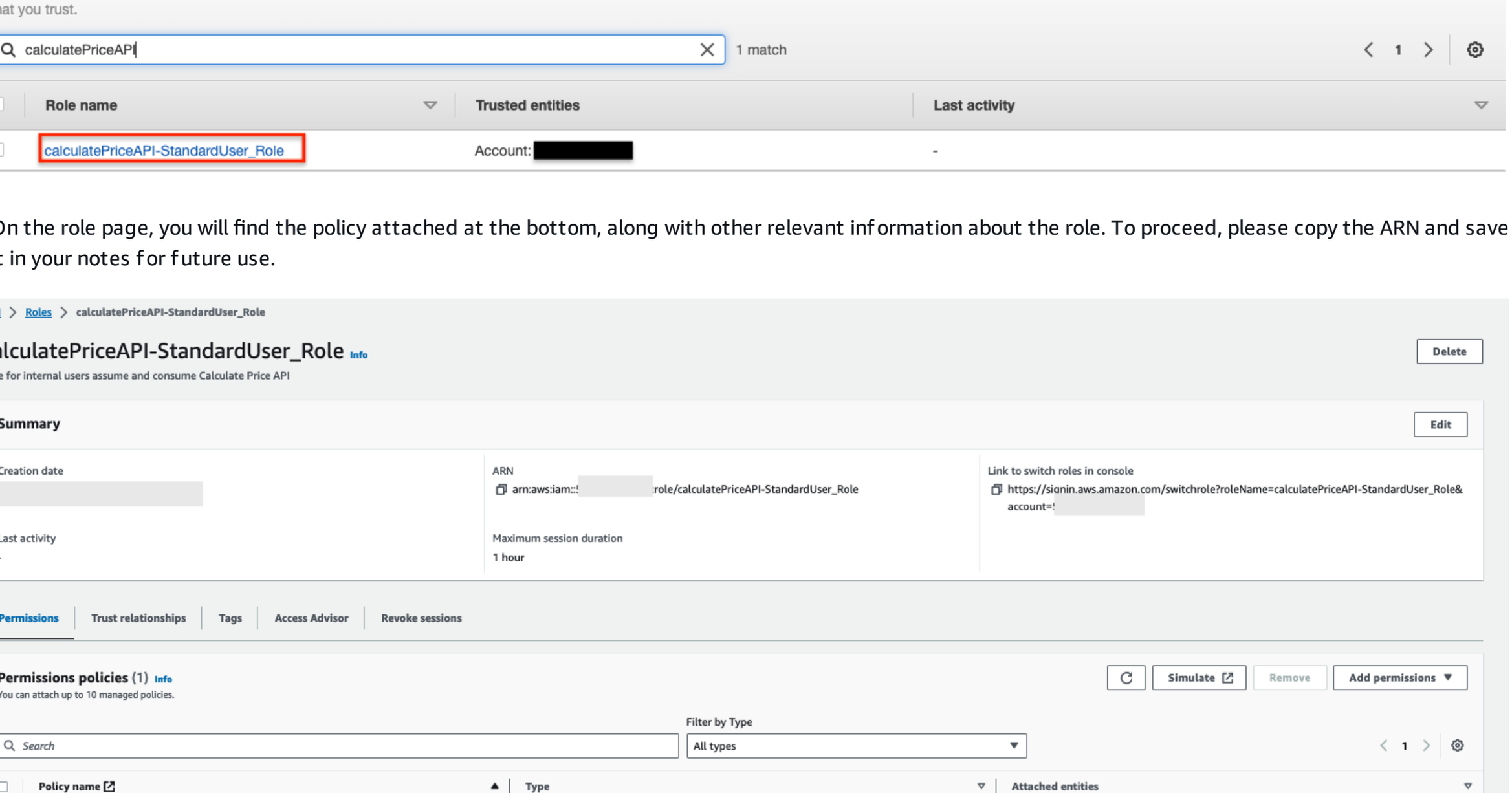


12. Once you have created the role, go back to the IAM console and select the role. Filter for `calculatePriceAPI`.

> If the full name of the role is hidden from view due to column width, you can hover over the partially visible name of the role to see the full name of the role as a tool tip.



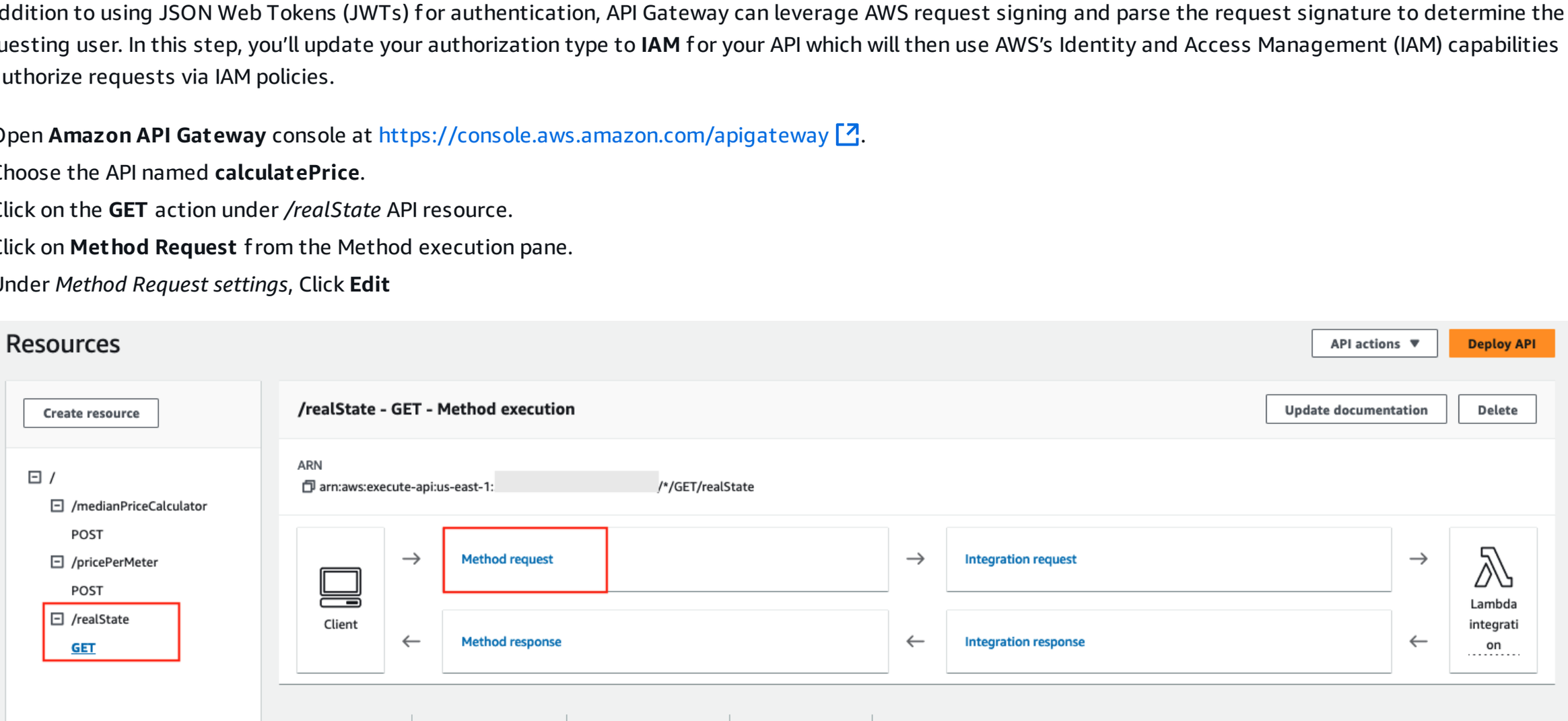
13. On the role page, you will find the policy attached at the bottom, along with other relevant information about the role. To proceed, please copy the ARN and save it in your notes for future use.



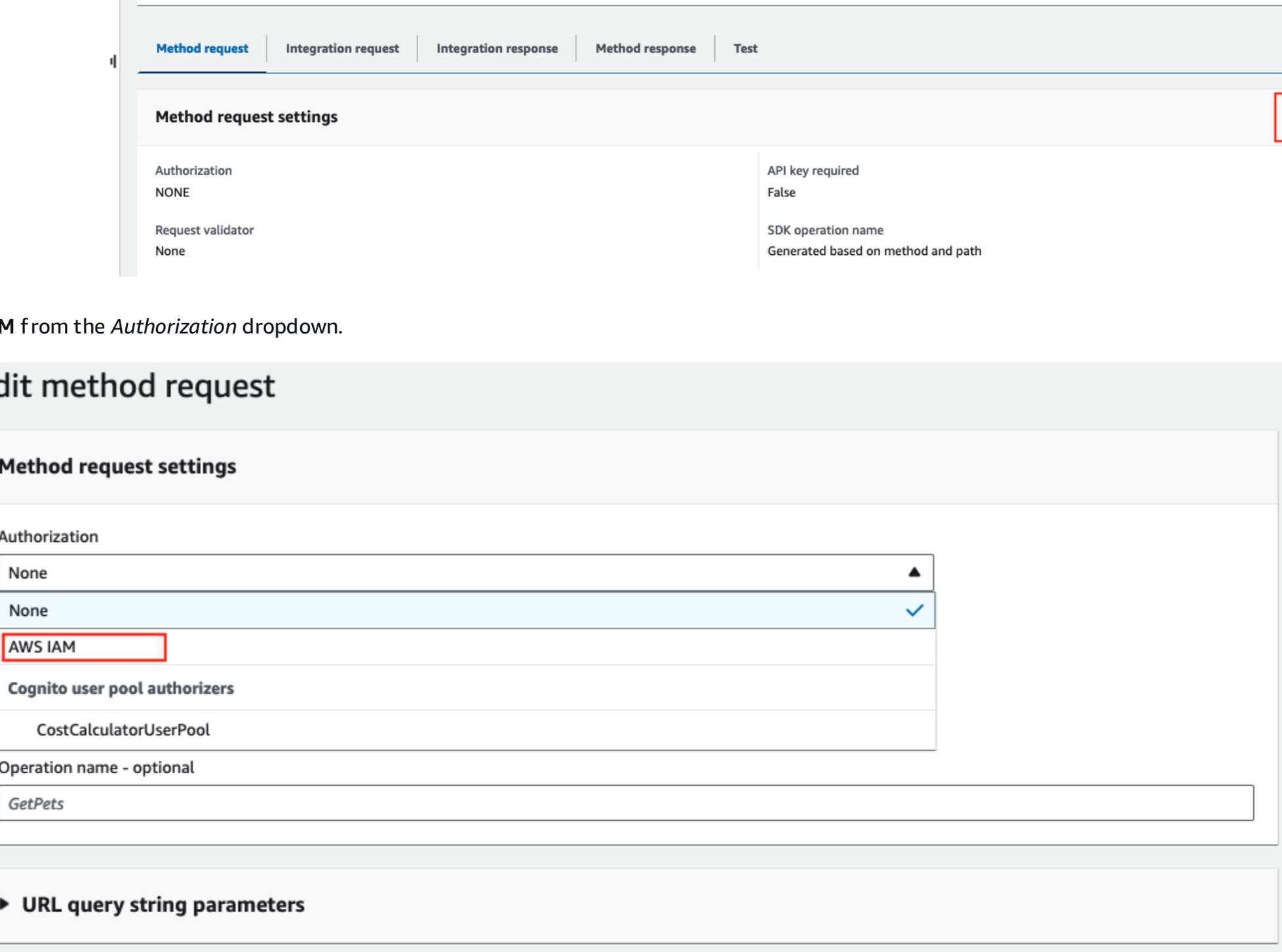
Update API Gateway Authentication

In addition to using JSON Web Tokens (JWTs) for authentication, API Gateway can leverage AWS request signing and parse the request signature to determine the requesting user. In this step, you'll update your authorization type to **IAM** for your API which will then use AWS's Identity and Access Management (IAM) capabilities to authorize requests via IAM policies.

1. Open **Amazon API Gateway** console at https://console.aws.amazon.com/apigateway.
2. Choose the API named **calculatePrice**.
3. Click on the **GET** action under */realState* API resource.
4. Click on **Method Request** from the Method execution pane.
5. Under *Method Request settings*, Click **Edit**



6. Select **AWS_IAM** from the *Authorization* dropdown.

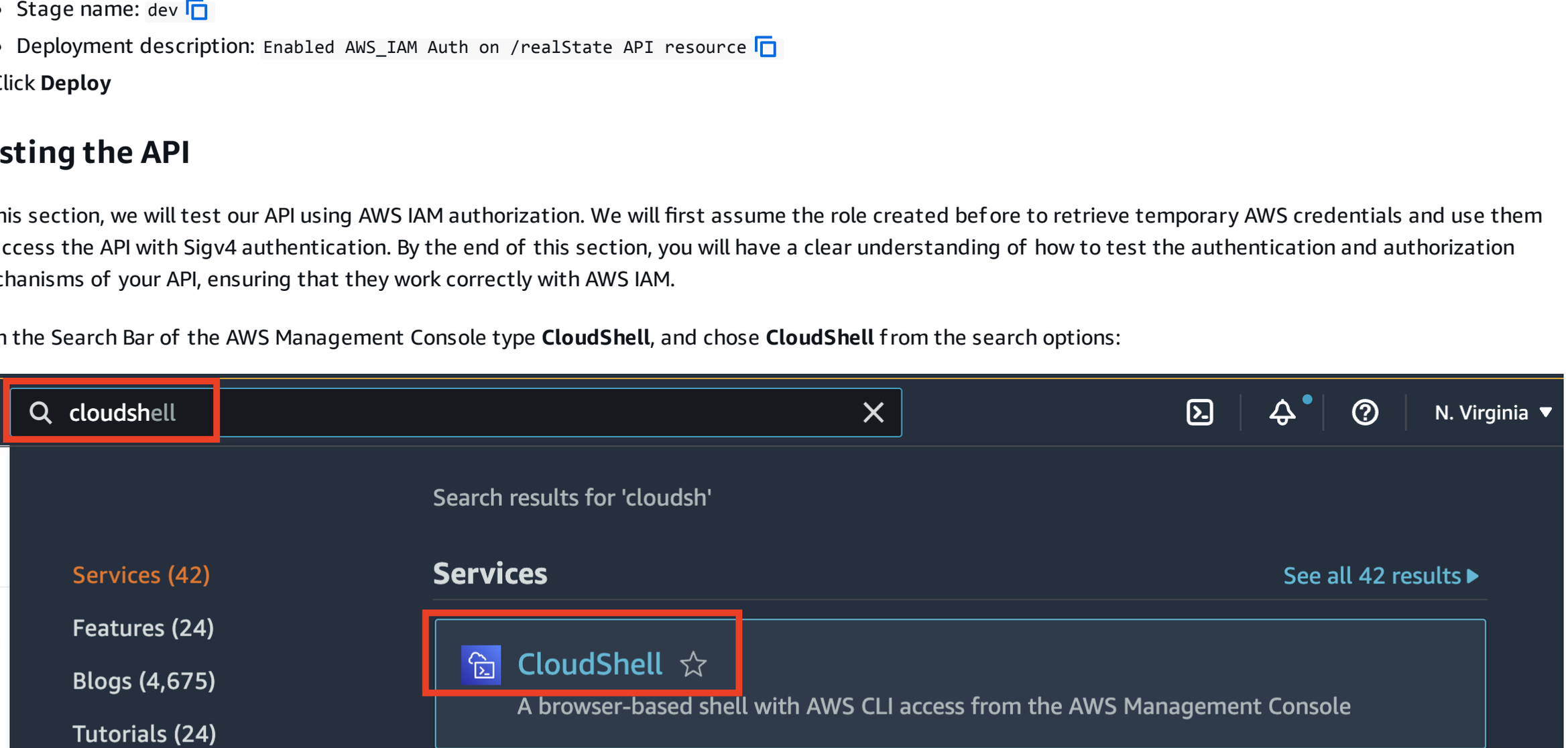


7. Click **Save**
8. From the Resources Console, Click on **Deploy API**.
9. Enter the following value:
   - Stage name: `dev`
   - Deployment description: `Enabled AWS_IAM Auth on /realState API resource`
10. Click **Deploy**

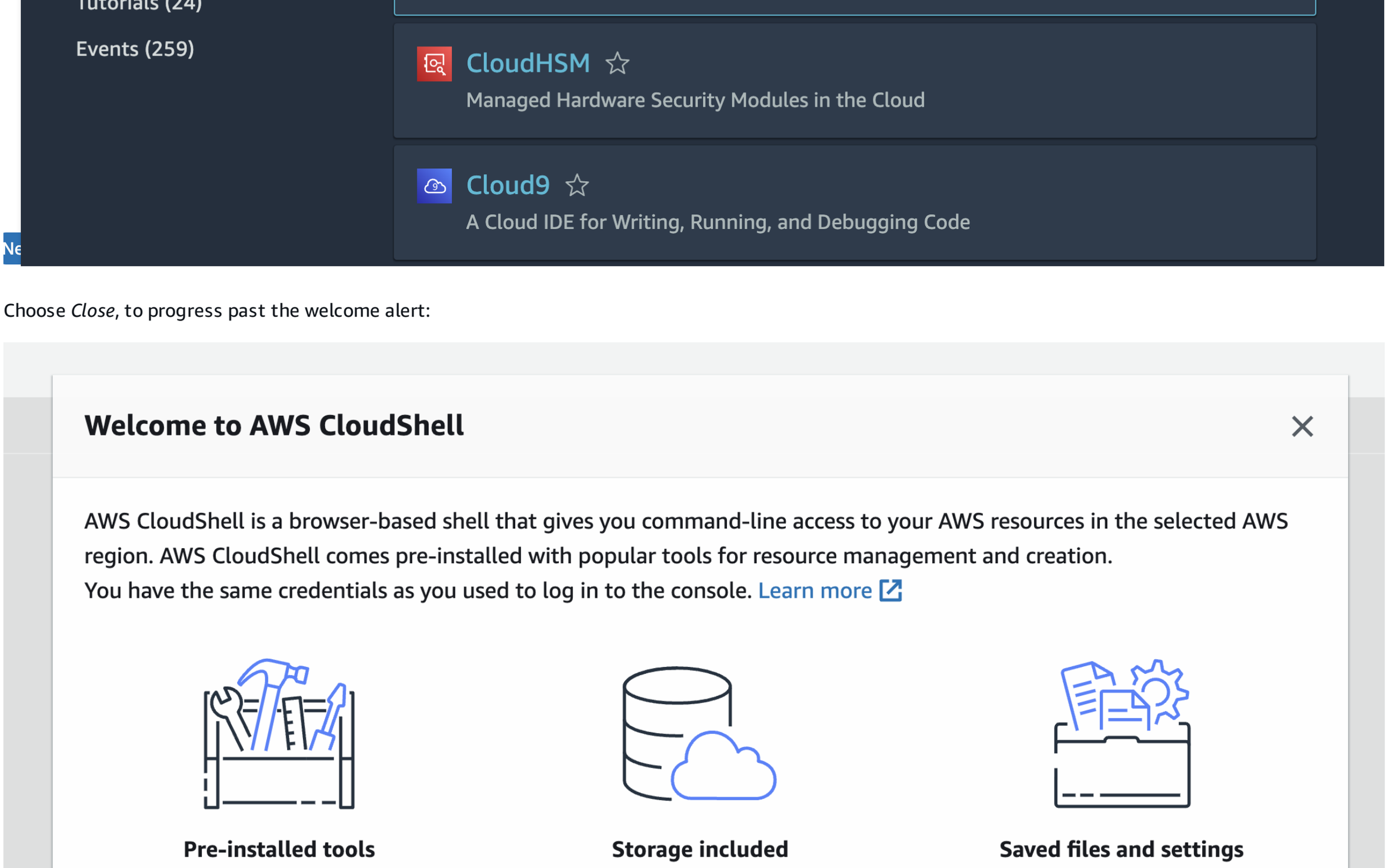
Testing the API

In this section, we will test our API using AWS IAM authorization. We will first assume the role created before to retrieve temporary AWS credentials and use them to access the API with Sigv4 authentication. By the end of this section, you will have a clear understanding of how to test the authentication and authorization mechanisms of your API, ensuring that they work correctly with AWS IAM.

1. In the Search Bar of the AWS Management Console type **CloudShell**, and chose **CloudShell** from the search options:



2. Choose *Close*, to progress past the welcome alert:



3. Enter the following command. Replace the **IAM_ROLE_ARN** attribute with the ARN of the IAM role created in previous section:

```
aws sts assume-role --role-arn <IAM_ROLE_ARN> --role-session-name execute-api
```

> **Important!** Save the credentials in your notes.

4. Navigate to Cloud9 in your AWS console. Make sure you are in the correct region.
5. Click **Open** on the **APIGatewayWorkshopWorkspace** environment.
6. Set up your testing folder

```
mkdir module-1 && cd module-1
```

7. Now you will create a sigv4 request using a python script. First create a new Python file within your Cloud 9 environment.

```
touch sigv4.py && chmod +x sigv4.py
```

8. open the `module-1/sigv4.py` file, copy/paste the following code in the file.

```
import os
import boto3
from botocore.auth import SigV4Auth
from botocore.awsrequest import AWSRequest
import requests

access_key = os.environ.get('AWS_ACCESS_KEY_ID')
secret_key = os.environ.get('AWS_SECRET_ACCESS_KEY')
session_token = os.environ.get('AWS_SESSION_TOKEN')
region = os.environ.get('AWS_REGION')
service = 'execute-api'
host = os.environ.get('API_GATEWAY_ID') + '.execute-api.' + region + '.amazonaws.com'
canonical_uri = '/dev/realState'
method = "GET"
url=f'https://{host}{canonical_uri}'

session = boto3.Session(
  aws_access_key_id=access_key,
  aws_secret_access_key=secret_key,
  aws_session_token=session_token,
  region_name=region
)

request = AWSRequest(
  method,
  url,
  headers={'Host': host}
)

SigV4Auth(session.get_credentials(), service, region).add_auth(request)

try:
  response = requests.request(method, url, headers=dict(request.headers), data={}, timeout=5)
  response.raise_for_status()
  print(f'Response Status: {response.status_code}')
  print(f'Response Body: {response.content.decode("utf-8")}')
except Exception as e:
  print(f'Error: {e}')
```

> This is designed to make an authenticated request to an Amazon API Gateway endpoint using **AWS Signature Version 4 (SigV4)** authentication. The SigV4 authentication ensures that the requests are properly authenticated and authorized by AWS, preventing unauthorized access to the API.

9. Now you will need to create a virtual environment and install `boto3` and `requests` packages.

```
python3 -m venv .venv
source .venv/bin/activate
pip3 install boto3 requests
```

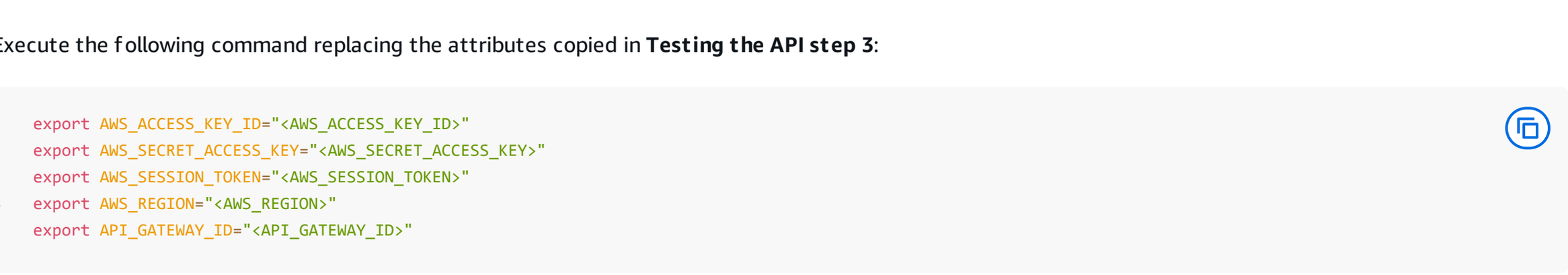
10. Execute the following command replacing the attributes copied in **Testing the API step 3**:

```
export AWS_ACCESS_KEY_ID="<AWS_ACCESS_KEY_ID>"
export AWS_SECRET_ACCESS_KEY="<AWS_SECRET_ACCESS_KEY>"
export AWS_SESSION_TOKEN="<AWS_SESSION_TOKEN>"
export AWS_REGION="<AWS_REGION>"
export API_GATEWAY_ID="<API_GATEWAY_ID>"
```

11. Execute the python script:

```
python3 sigv4.py
```

12. You should see the API result with the list of real states.



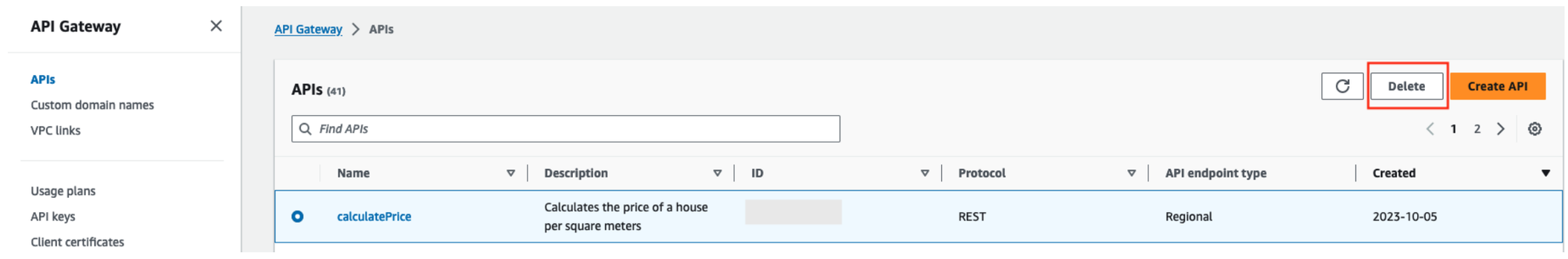
Clean Up

Important
Follow the instructions on this page only if you are executing this workshop in your own account.

Delete your API

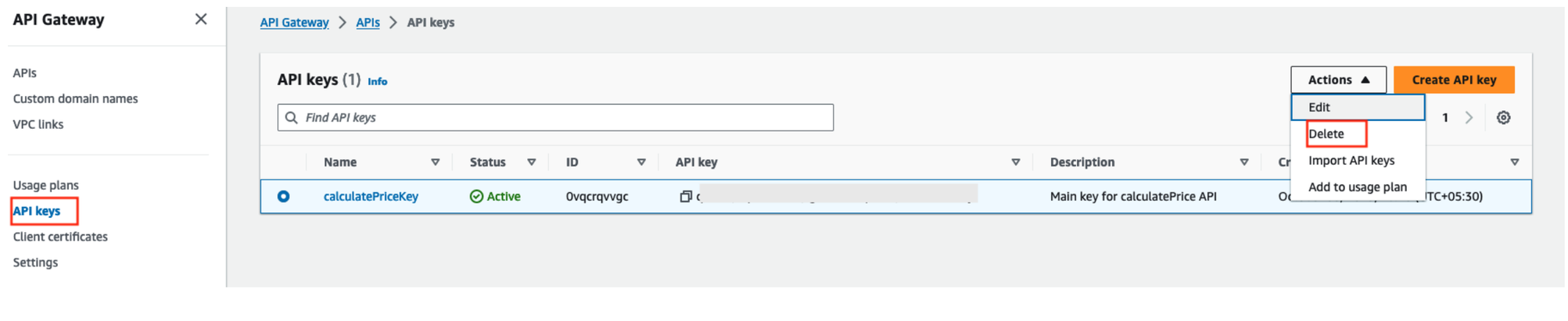
In Amazon API Gateway Console:

1. Select **calculatePrice** API by clicking on the radio button next to it
2. Click **Delete**
3. Type `confirm` in the popup textbox after verifying the API Name and click **Delete**



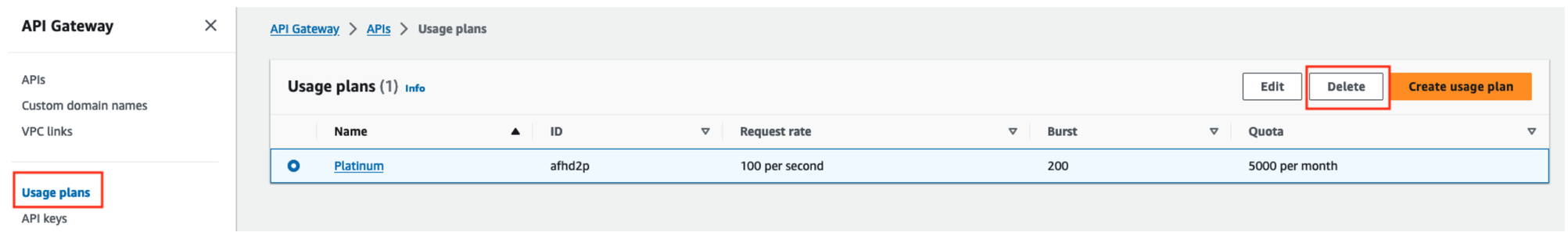
Delete your API Key

1. Click on **API keys** from the left navigation menu.
2. Select the **calculatePriceKey** by clicking on the radio button next to it.
3. Click **Delete** from the *Actions* menu
4. Click **Delete** in the popup



Delete your Usage Plan

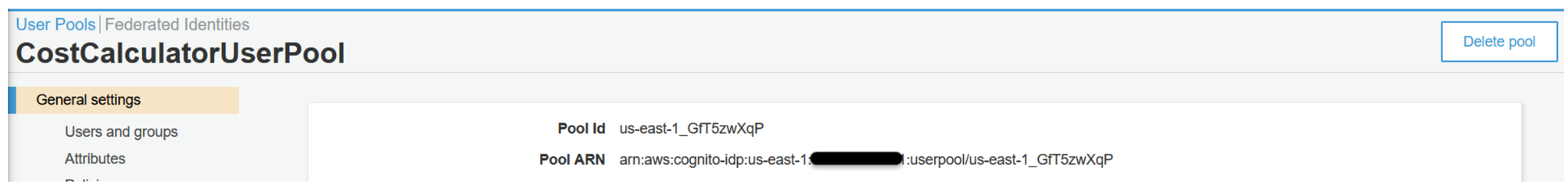
1. Click on **Usage plans** from the left navigation menu.
2. Select the **Platinum** usage plan by clicking on the radio button next to it.
3. Click **Delete**
4. Type `confirm` in the popup textbox after verifying the Usage Plan Name and click **Delete**



Delete Cognito User Pool

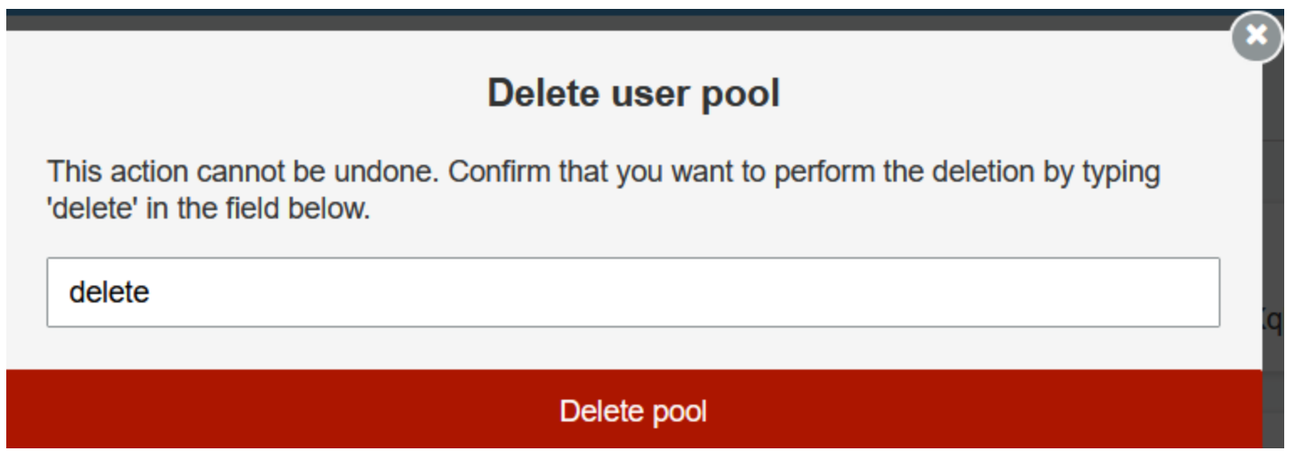
In Amazon Cognito Console

1. Click **Manage User Pools**
2. Click **CostCalculatorUserPool**
3. Click **Delete pool**



4. Type `delete` in popup and Click **Delete pool**

ou may need to refresh the page to see the pool as deleted.



Congratulations! You have successfully completed this module.

Previous Next