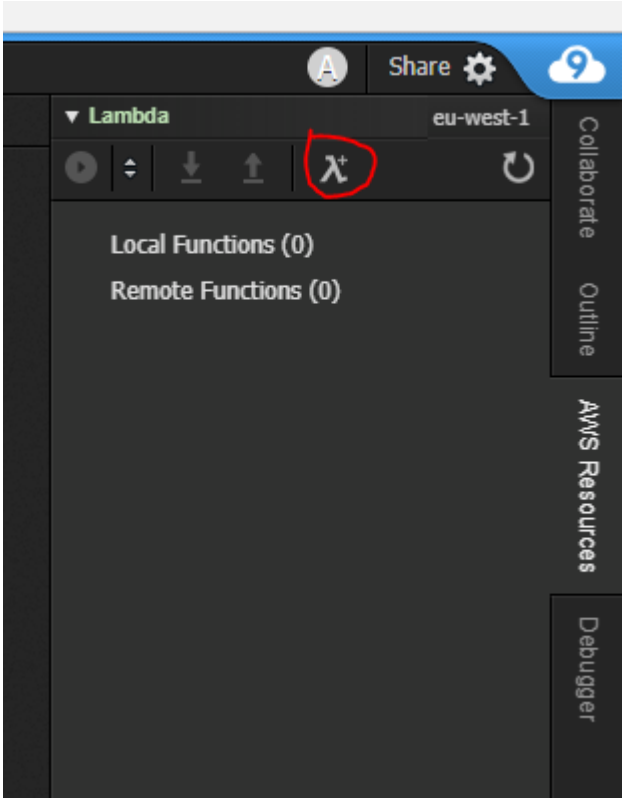
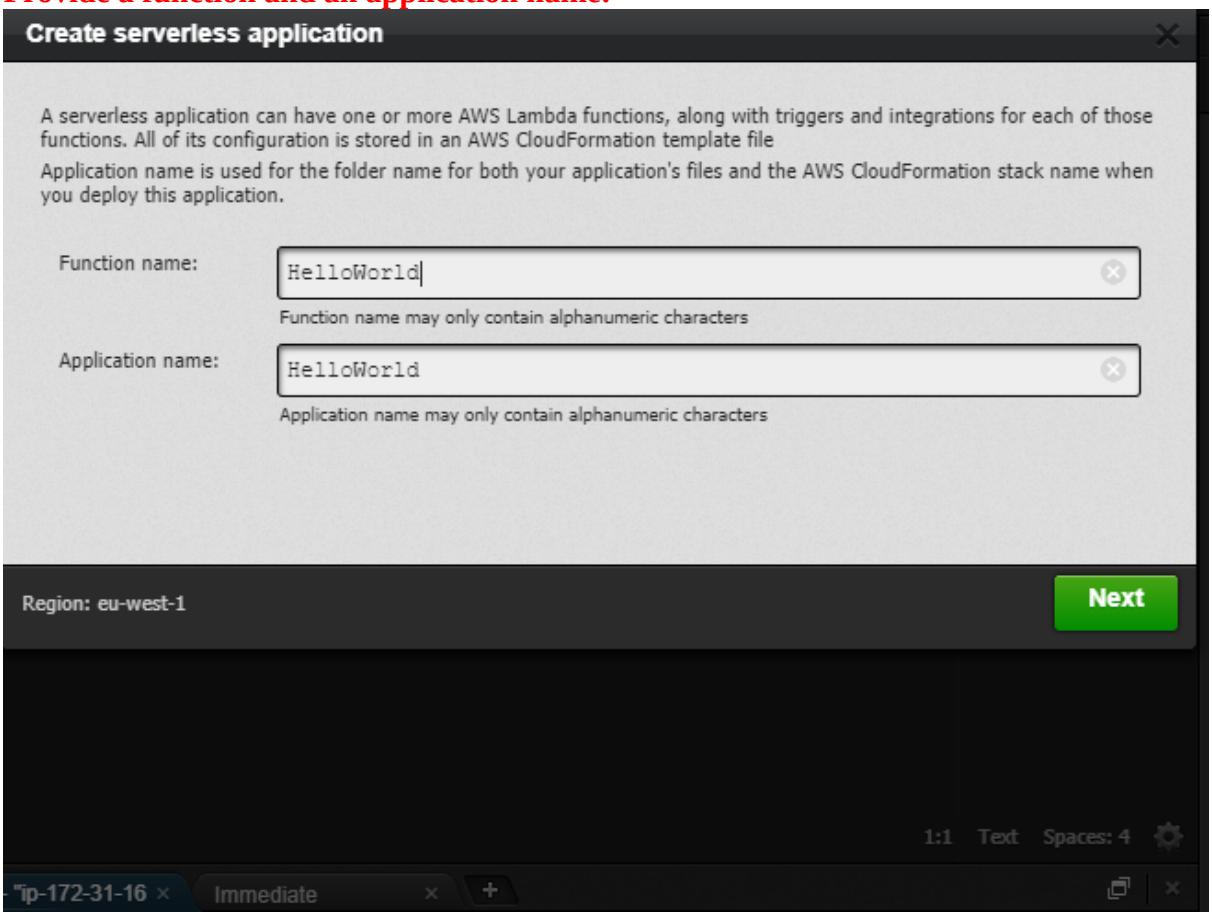
Building HelloWorld

Let's start off create a new application for the mandatory "Hello World" example:

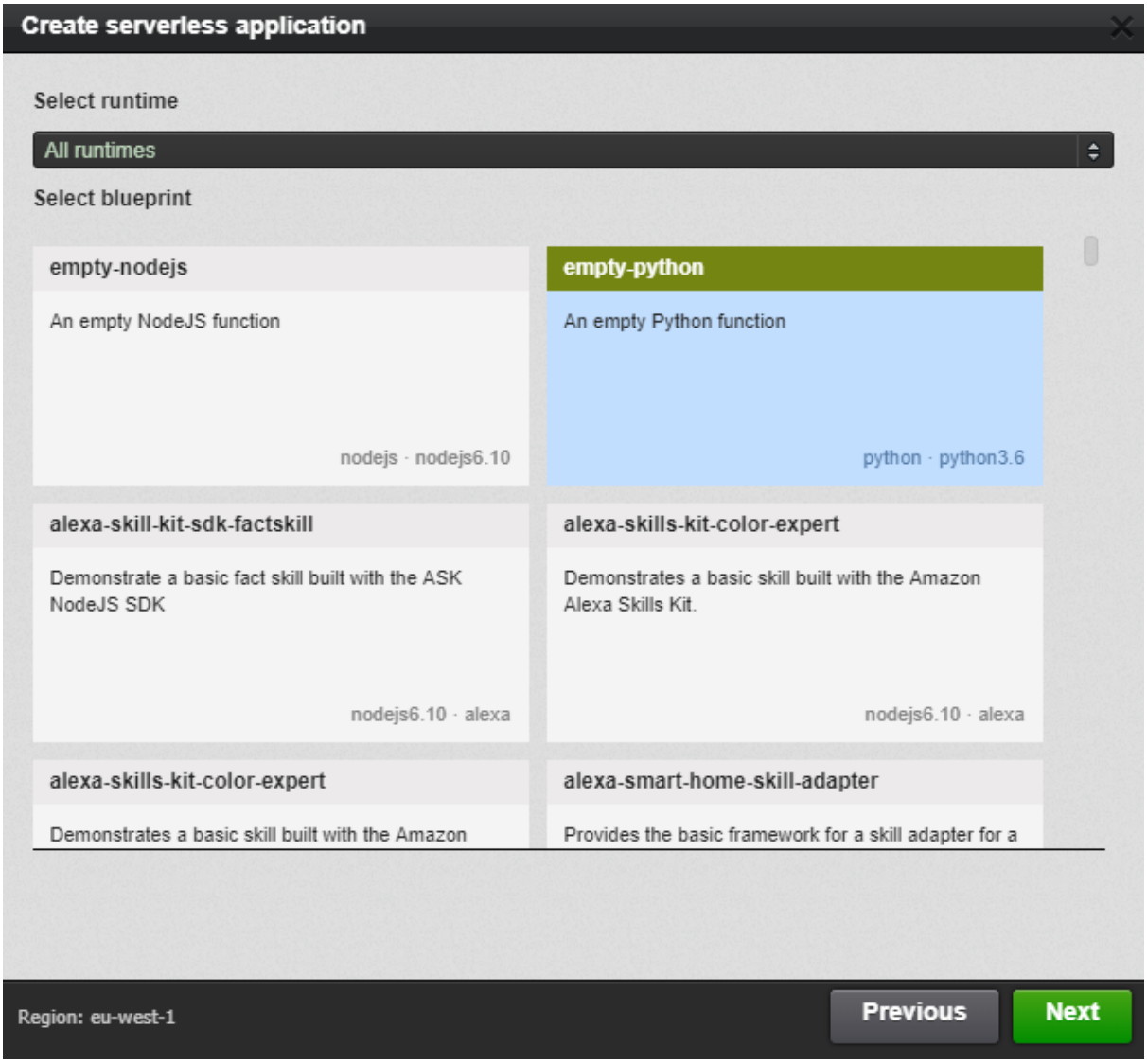
Create a new lambda function by using the AWS Resources menu and selecting the lambda+ icon:



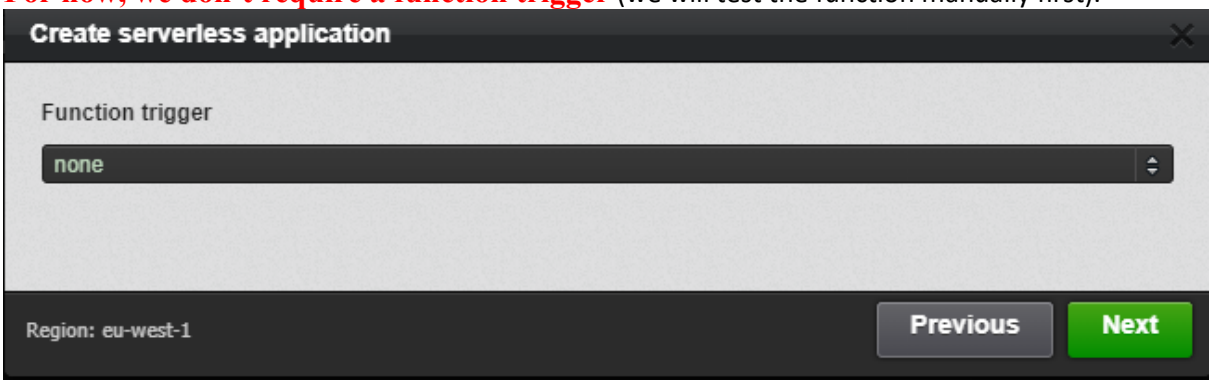
Provide a function and an application name:



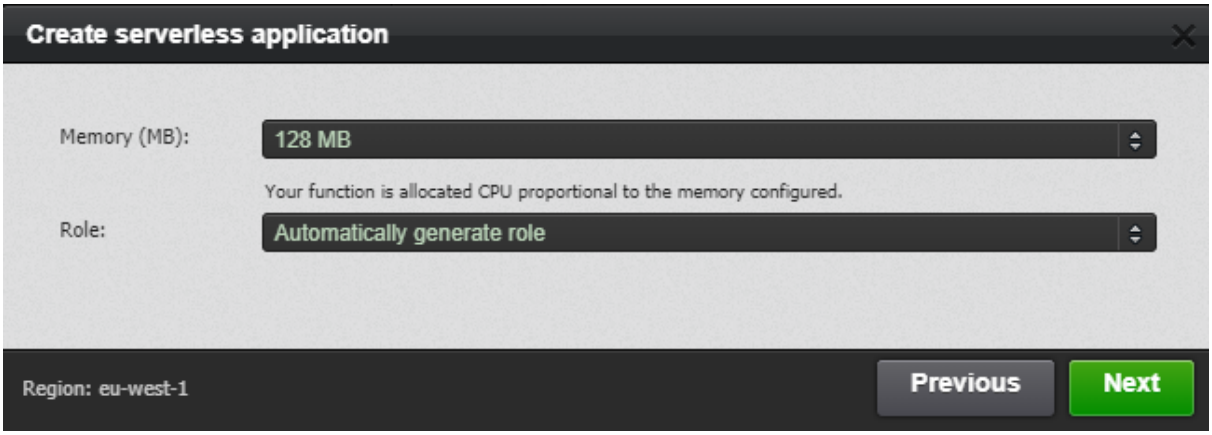
The function needs to be developed. Let's **select an empty Python 3.6+ template** to write the Hello World example as **Python code**:



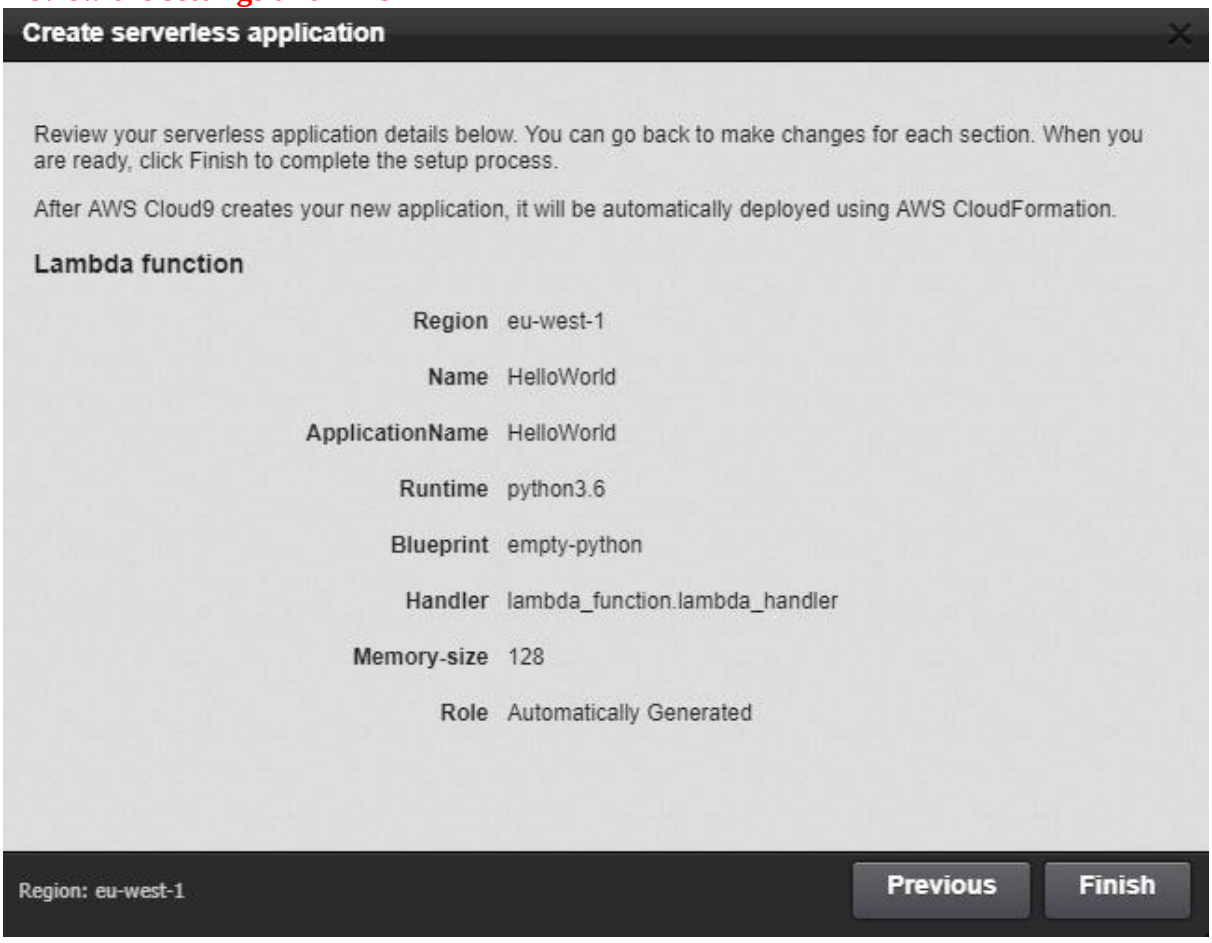
For now, we don't require a function trigger (we will test the function manually first):



Also, **default memory settings and a default generated role should be enough** - we're not doing any heavy lifting, nor are we using any asynchronous functionality:



Review the settings and finish:



From the setting above, you should note that the Handler is "lambda_function.lambda_handler". What does this mean? Well, if the function is invoked, it will look for a Python **source file** named "lambda_function" (.py, actually) and try and execute the **function** "lambda_handler" inside this file. To isolate this project from other projects in the same IDE, Cloud9 will create a so-called "Virtual Environment" for Python, where all its dependencies will be stored and not pollute the global namespace - reducing the chances of conflicts.

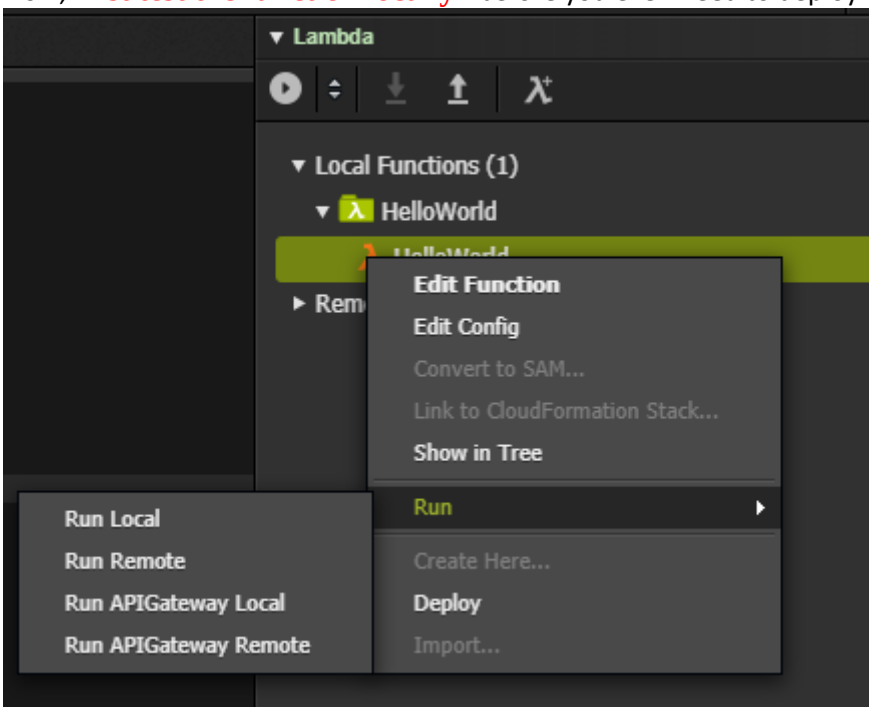
As you can see, the AWS Lambda framework will provide the function with an event and a context: the event is the event that triggered the invocation of the function, the context consists of AWS specific environment information, the so-called execution context.
Let's insert code to examine both the event and the context passed in and return a greeting using "Hello " plus the name if present in the input event, otherwise name should be defaulting to "World" ".

lambda_function.py

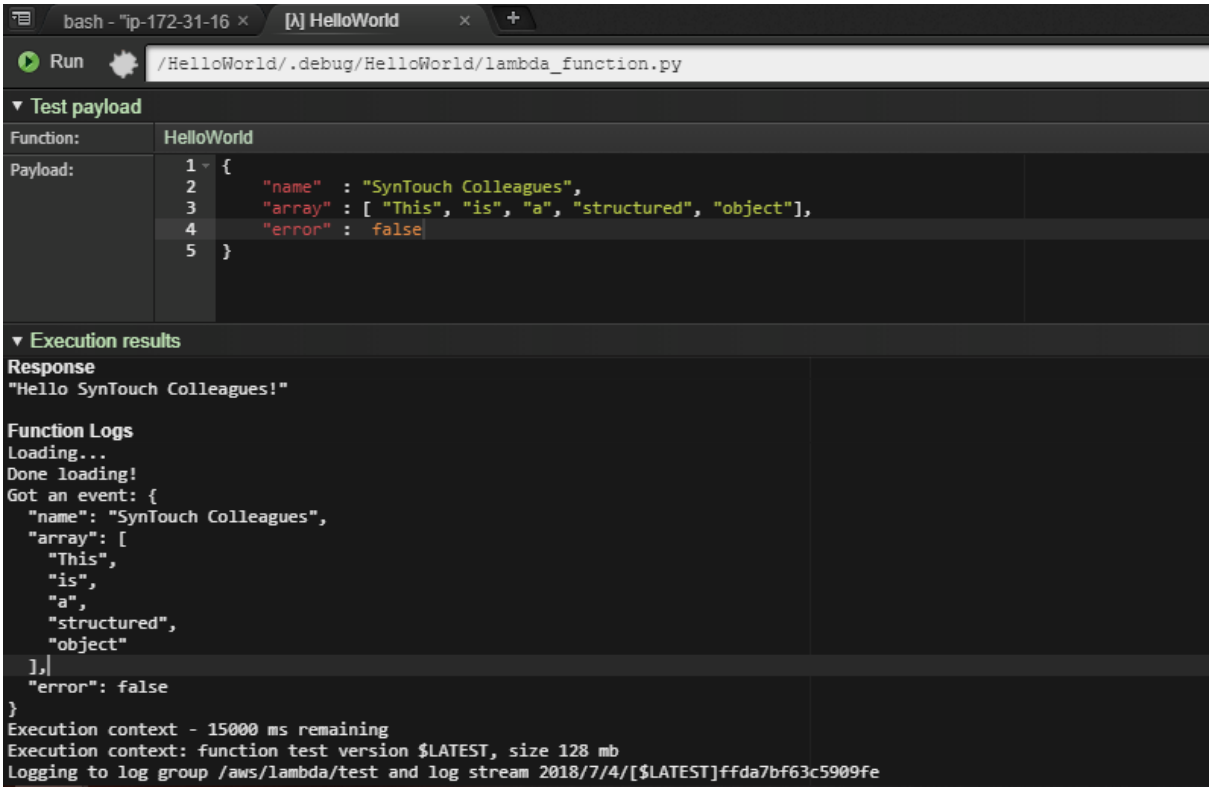
```
import json

print('Loading...')

def lambda_handler(event, context):
    # event is a dict object
    print('Got an event: ' + json.dumps(event, indent=2))

    # context is an object - https://docs.aws.amazon.com/lambda/latest/dg/python-context-object.html
    print("Execution context - {} ms remaining".format(context.get_remaining_time_in_millis()))
    print('Execution context: function {} version {}, size {} mb'.format(context.function_name, context.function_version, context.memory_limit_in_mb))
    print('Logging to log group {} and log stream {}'.format(context.log_group_name, context.log_stream_name))

    naam = 'World' if 'name' not in event else event['name']

    groet = 'Hello ' + naam + "!"
    print('We gaan groeten ... ' + groet)

    return groet

print('Done loading!')
```

Relevant code snippets (code for the handler function, or the payload for the test event) can be found in the GitHub repository https://github.com/mnuman/syntouch-aws-lambda-workshop/tree/master/2-Hello%20World.

Now, **first test the function locally** - before you even need to deploy it to the real cloud:



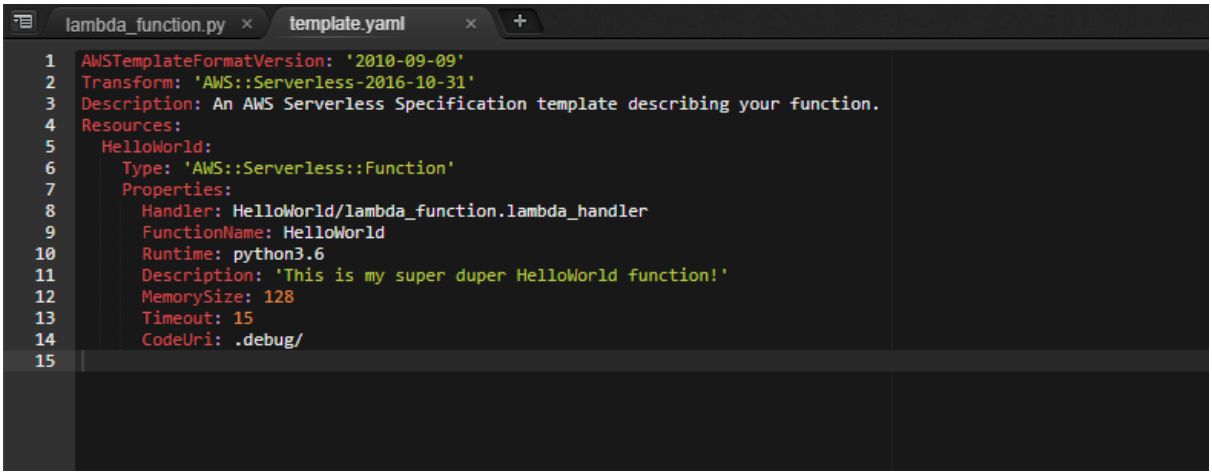
This opens a window that lets you specify an **event payload**. **As we are expecting a JSON string element called name at a minimum set this element in the event payload**:



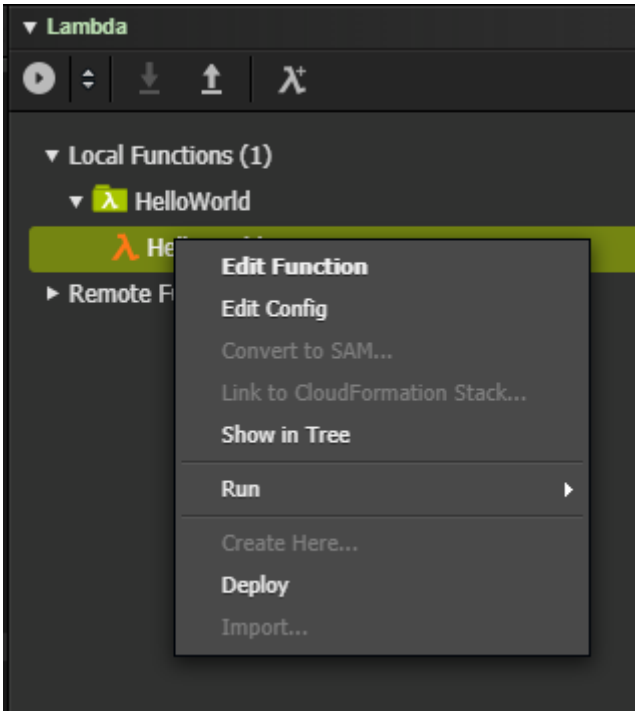
When supplying a name attribute in the JSON event, the contents are used for the greeting.
You can see the function loading (initializing), finish the initialization and then executing on the supplied event. As we're running locally, the actual names ("test" for the function name) may be different from the real run-time environment values.

What Cloud9 is doing behind the scenes, is creating a SAM (Serverless Application Model) template to provide all resources required (template.yaml). Actually, this is an extension to the Amazon CloudFormation service, which creates or updates an "application stack" based on an input template ('infrastructure' as code!)

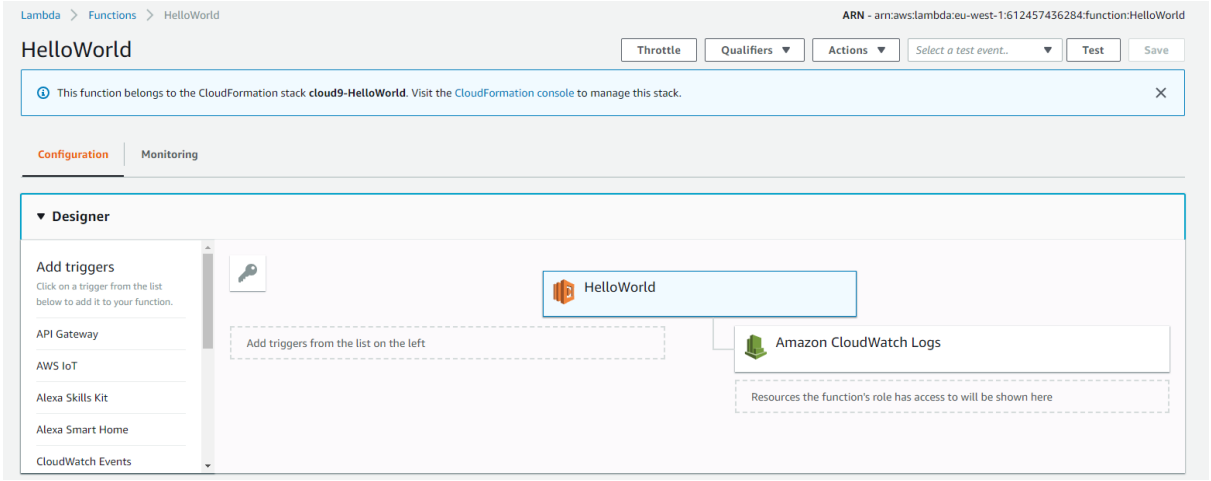
Before deployment, we need to make a small change to the template that is used to provision the stack (if we don't we cannot control the actual name assigned to the function completely): add a FunctionName attribute to the HelloWorld function and set this to HelloWorld:



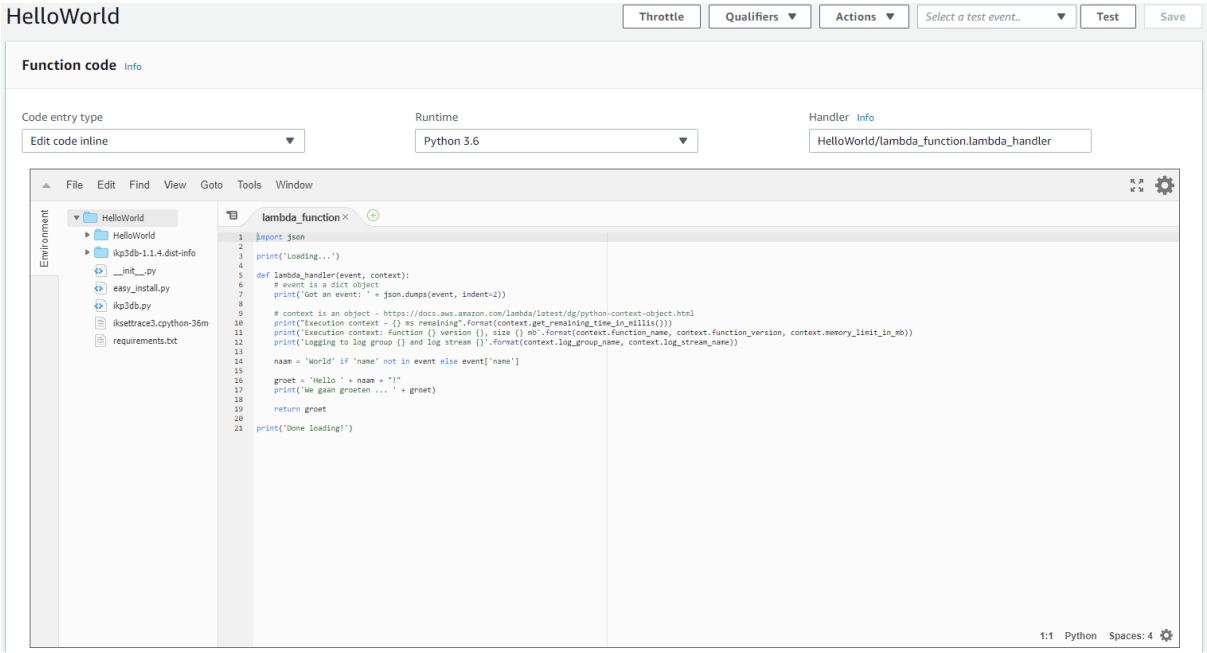
Now the code is ready to rock & roll! **Deploy your code**



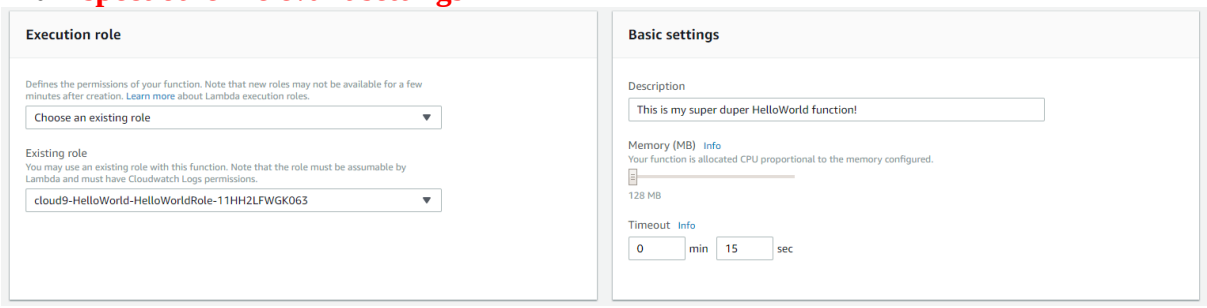
After some seconds, a stack has been provisioned using CloudFormation. **Examine the actual HelloWorld Lambda function from the AWS Lambda console**:



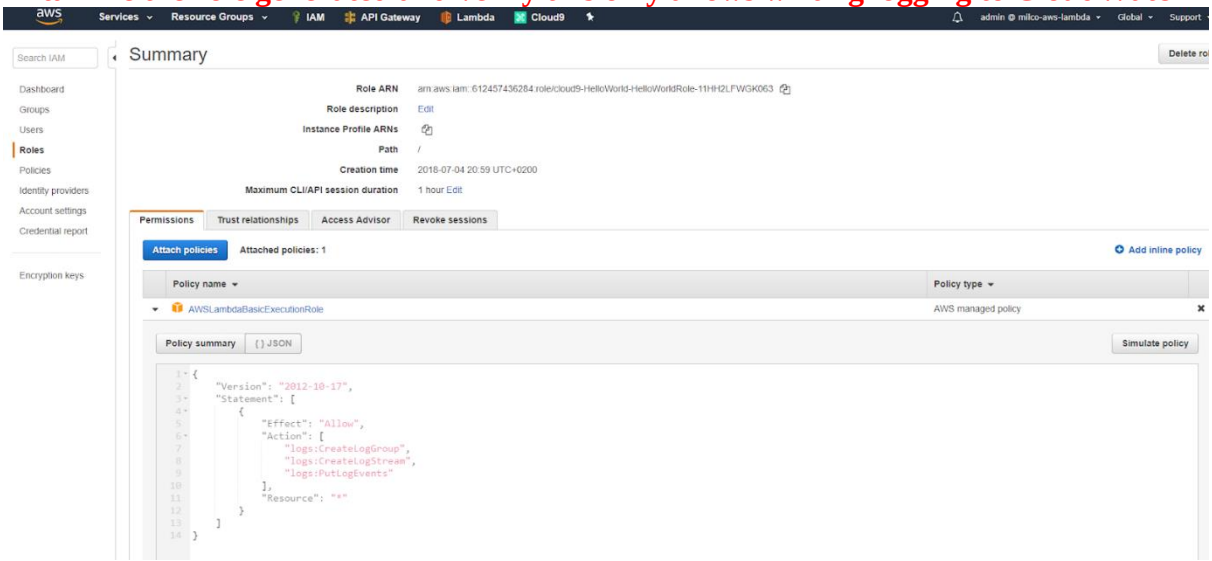
Examine the code:



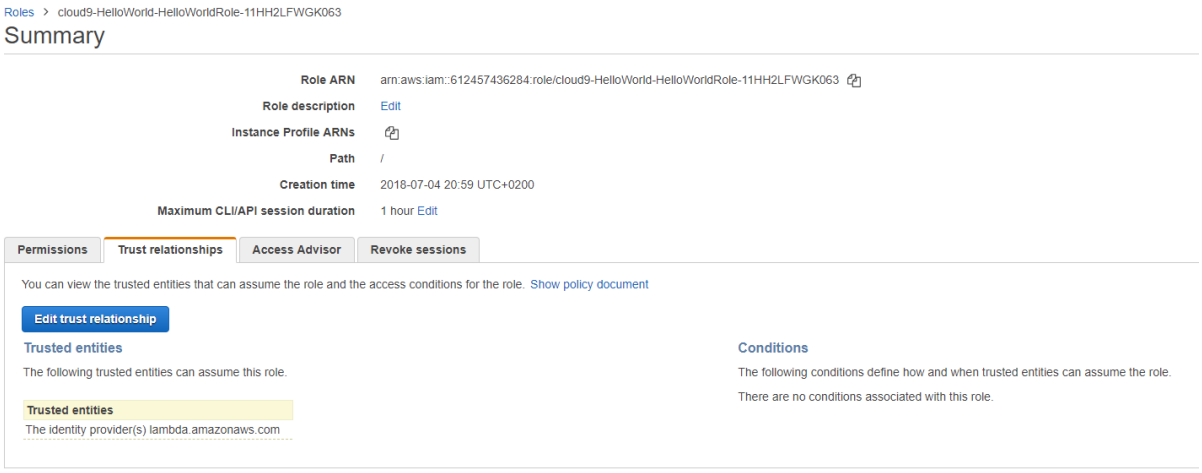
And **inspect other relevant settings**:



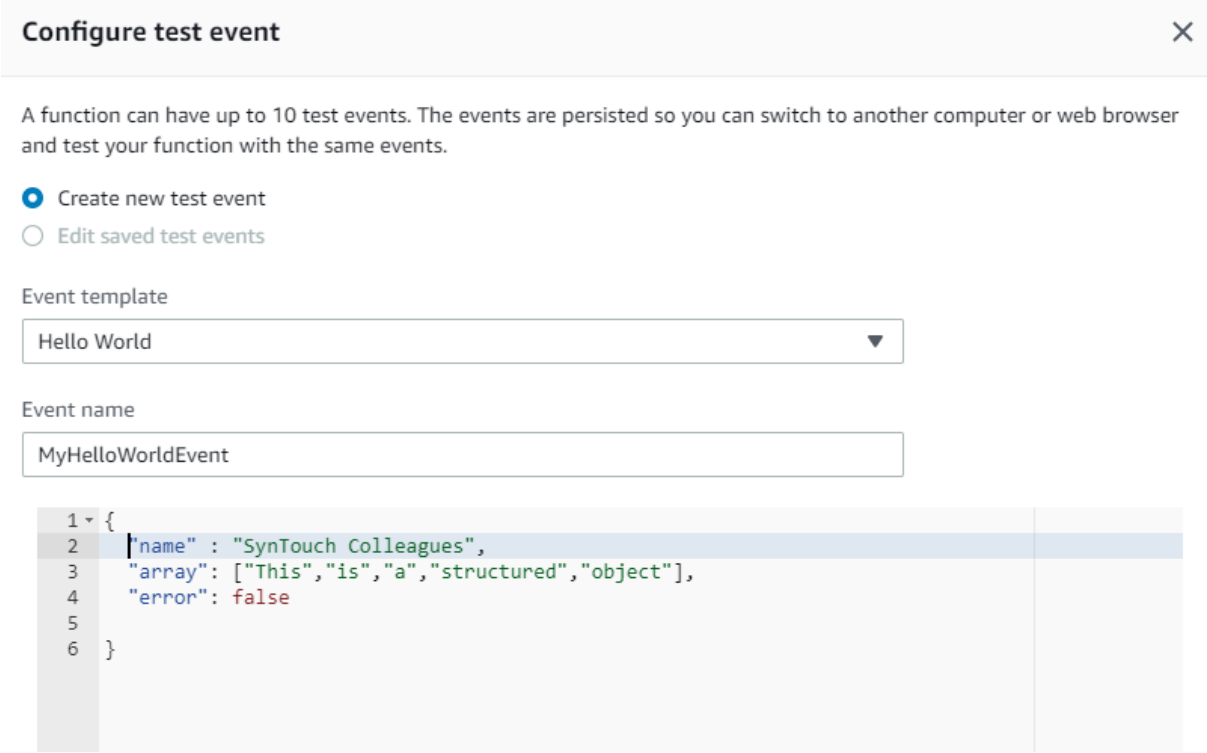
Extamine the role generated and verify this only allows writing logging to CloudWatch:



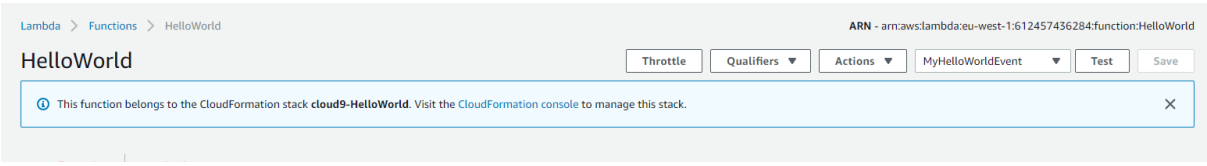
And only AWS Lambda is allows to assume this role:



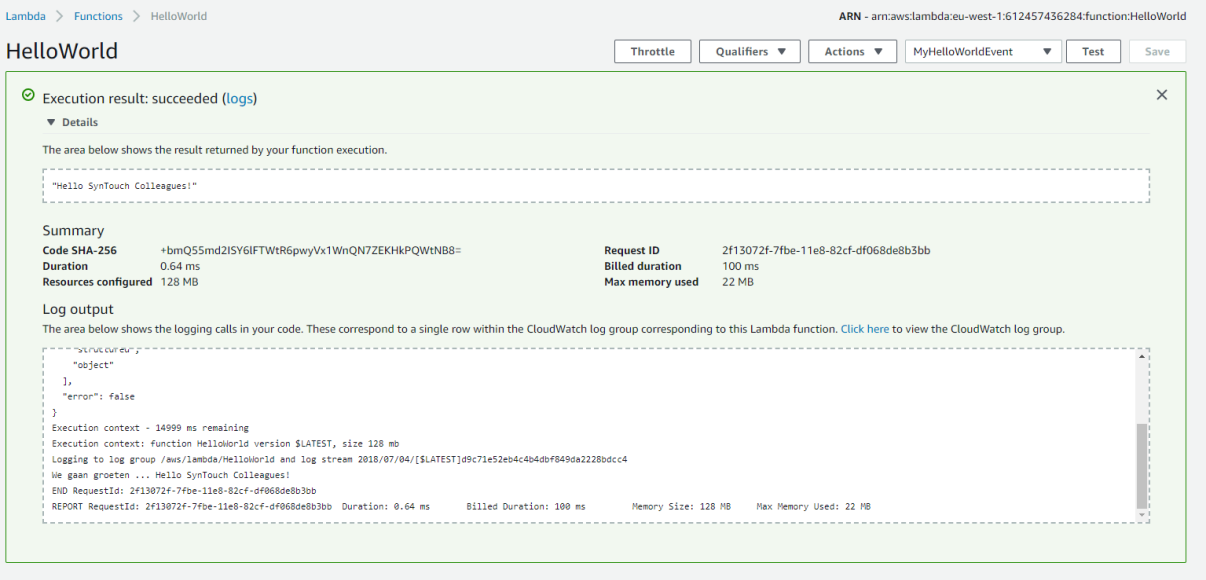
Return to the AWS HelloWorld Lambda function and create a new test event by using the Test button:



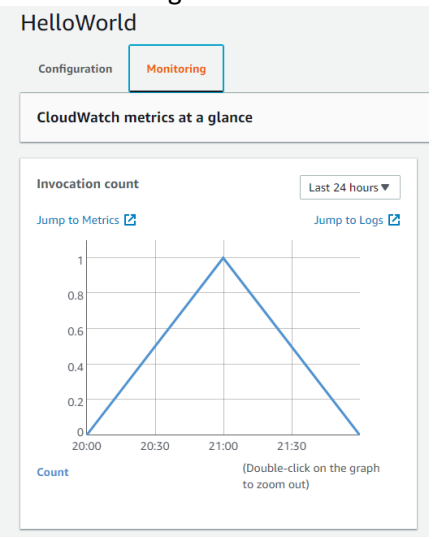
Save the event and test!



The execution logs provide you with the actual output, but also with useful information about the duration, the duration you will be billed for (rounded up to 100 ms) and the actual memory consumed. **Examine the log output for the actual execution in the cloud**.



The monitoring tab returns miscellaneous statistics on your function, but also offers access to the cloudwatch log files where this time period was logged:

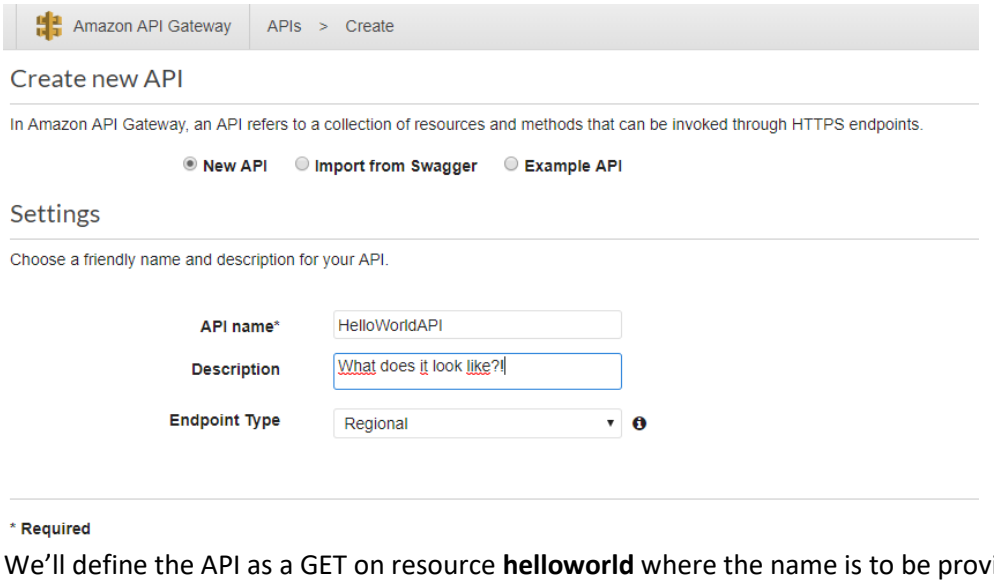


Expose HelloWorld as a WebAPI

Now that we have defined the mandatory HelloWorld artefact as a serverless function in AWS Lambda, let's go the extra mile and expose this to the GBI ("Grote Boze Internet") as an WebAPI to be called over HTTP.

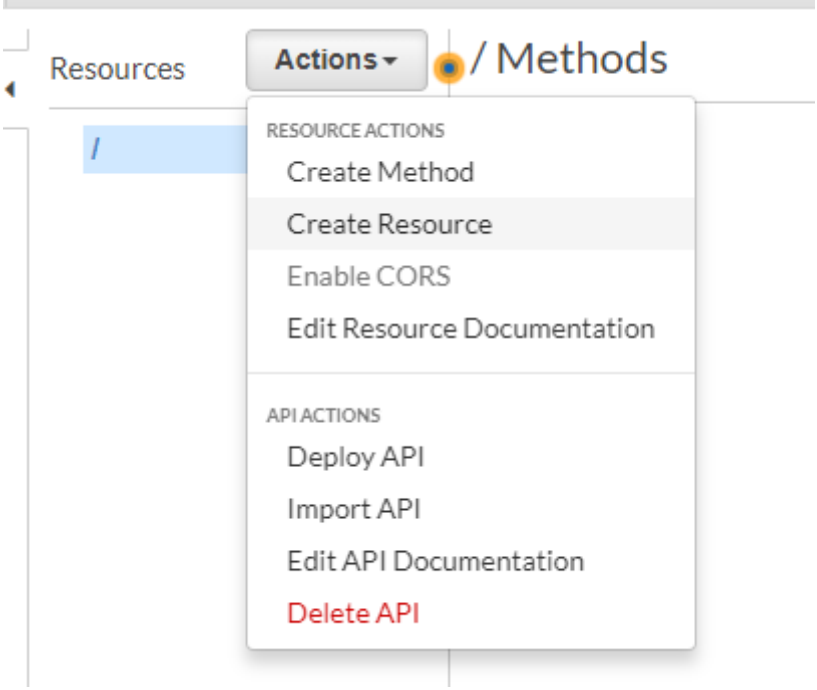
Make sure that you're still in the eu-west-1 (Ireland) region, as this is where your Lambda function lives.

Define a new API in API Gateway:

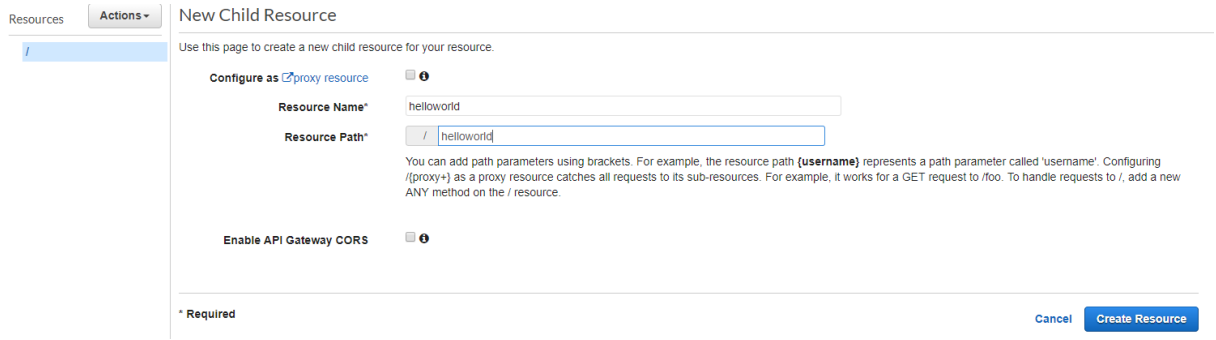


We'll define the API as a GET on resource **helloworld** where the name is to be provided as a query parameter …

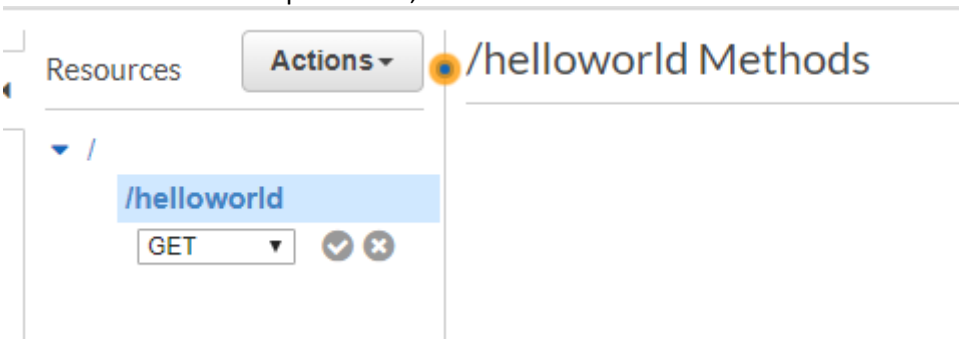
From the drop down, choose "Create Resource":



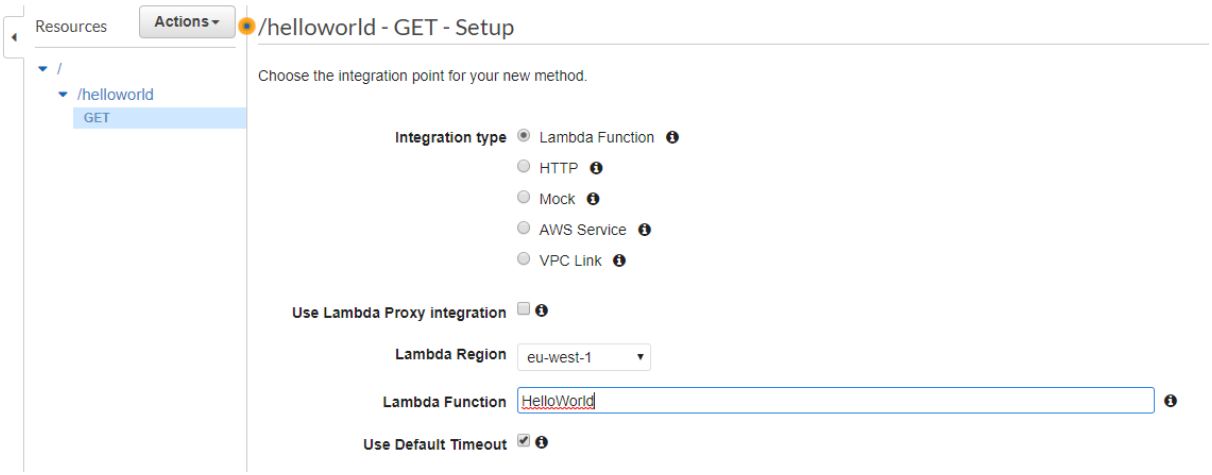
Create the /helloworld resource:



After creation of the resource, **make sure the new resource is selected** and **choose "Create Method"** from the dropdown list; select the **GET** HTTP method:

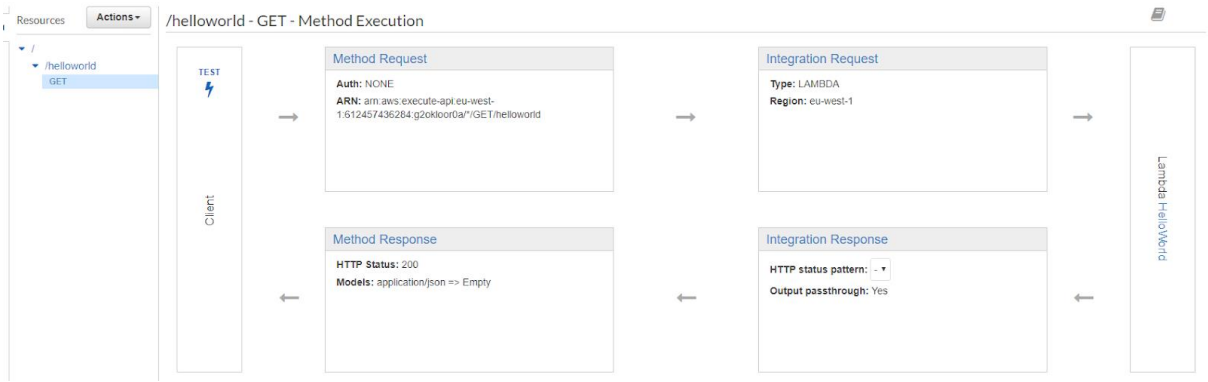


If you **press the "Okay" action** **icon**, you're presented with a configuration form to connect your exposed endpoint. Of course **we'll be connecting this to the HelloWorld lambda function**:



When asked, authorize API Gateway to invoke your Lambda function.

The API Gateway represents the integration with the backend as a number of phases:

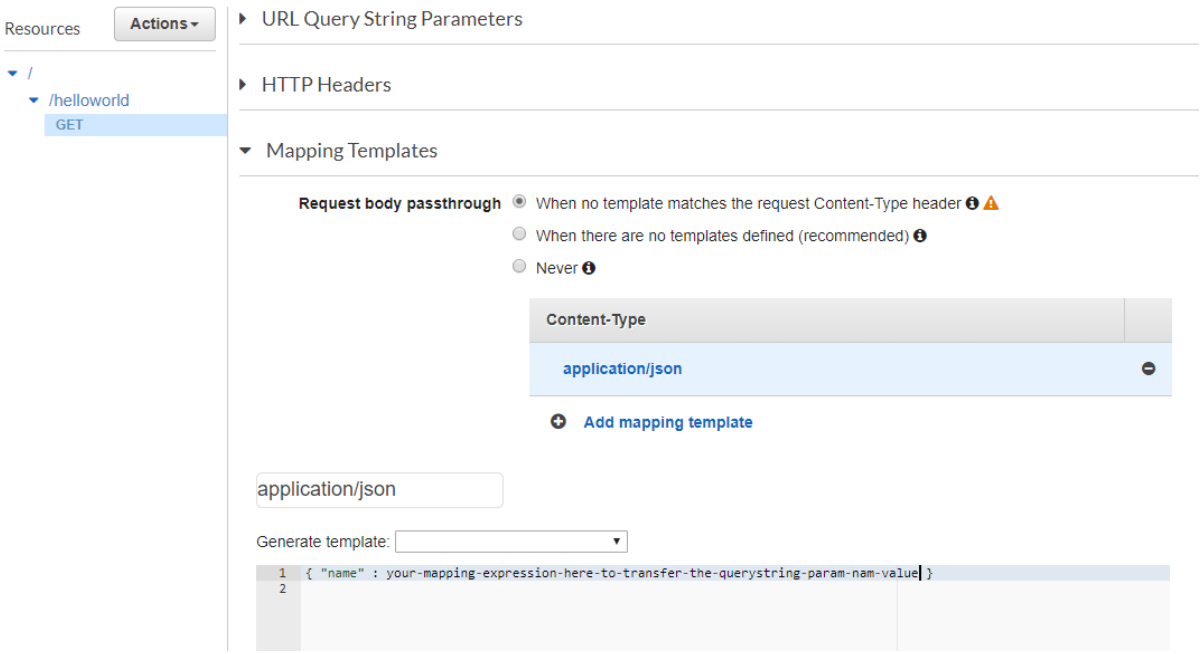


In the first stage (Method Request), the expected payload structure, headers and query string parameters can be defined. The request can be rejected if it does not satisfy the defined requirement. For our current purpose, this is not relevant.

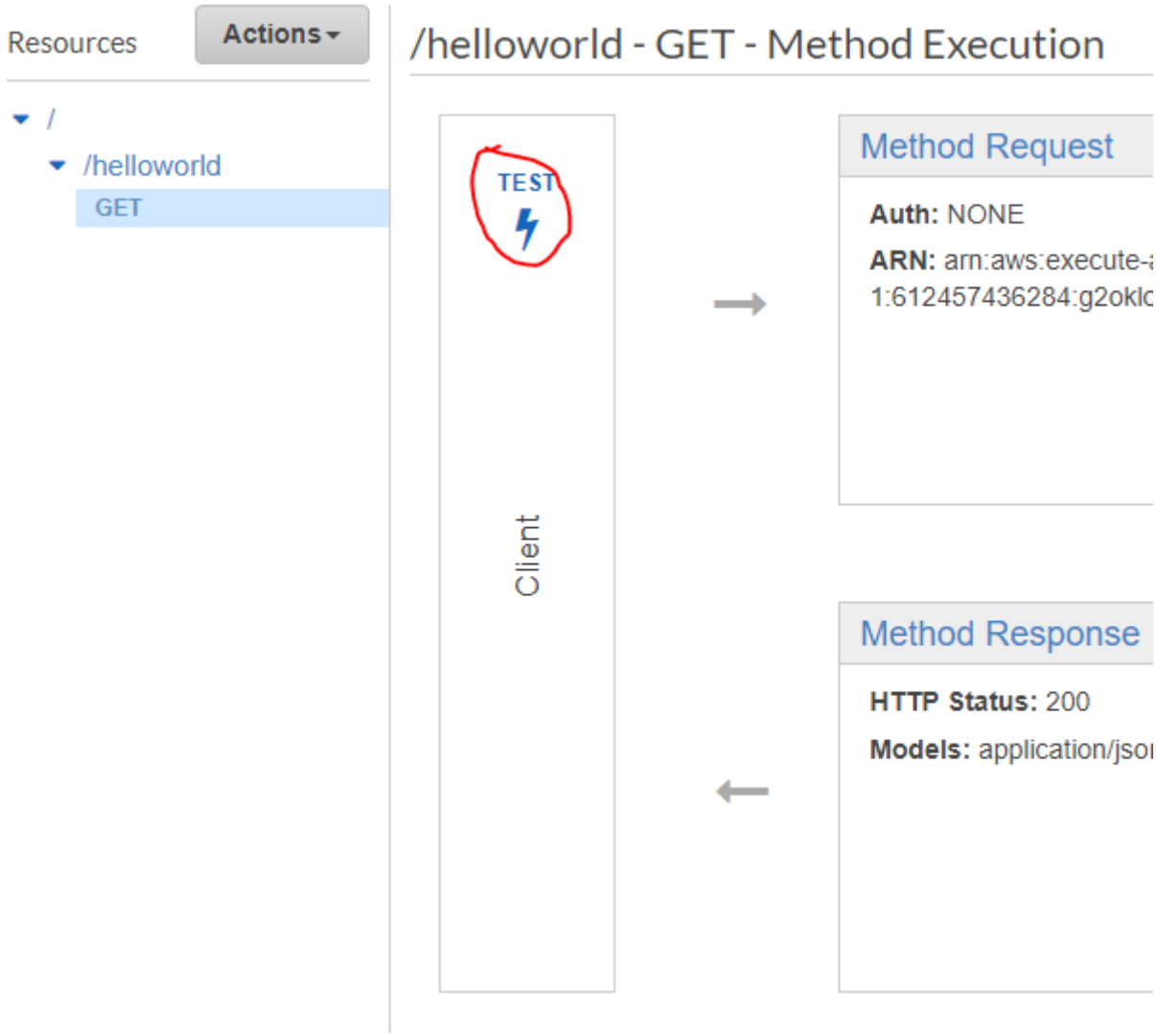
The next step, Integration Request, is used to transform the received request to the request the backend (in this case our HelloWorld lambda function) will receive.

Here, we need to find a way to transfer the contents of the name querystring parameter to a name element in the JSON payload for the backend service. API Gateway supports the Apache Velocity templating engine for performing these tasks. For an overview of the available mapping functions, see https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-mapping-template-reference.html

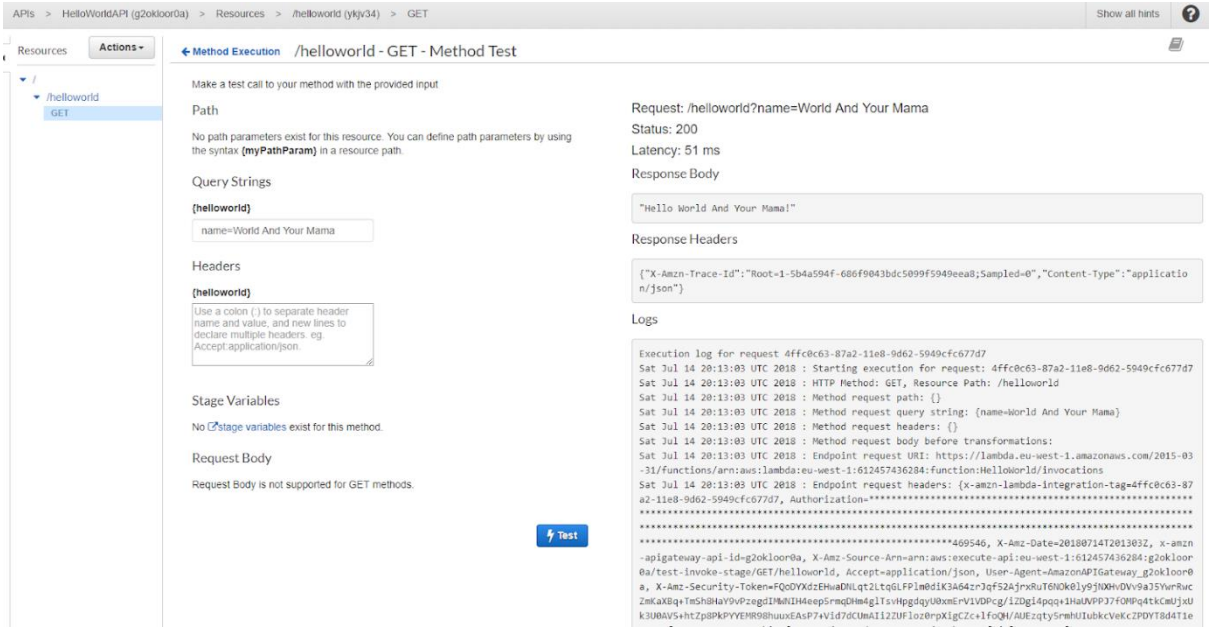
Define a new template to be used for application/json content types; insert your expression replacing your-mapping-expression-here-to-transfer-the-querystring-param-nam-value:



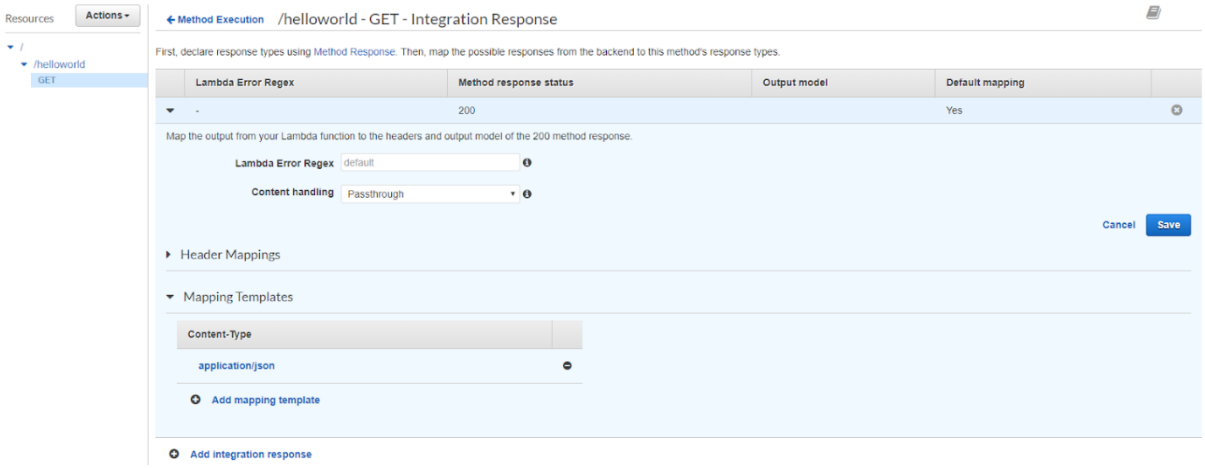
After completing your mapping template, navigate back to your API, select the GET method on helloworld and invoke the TEST button on the client:



Next, provide the query string parameter name and set to a value of your choice. TEST!



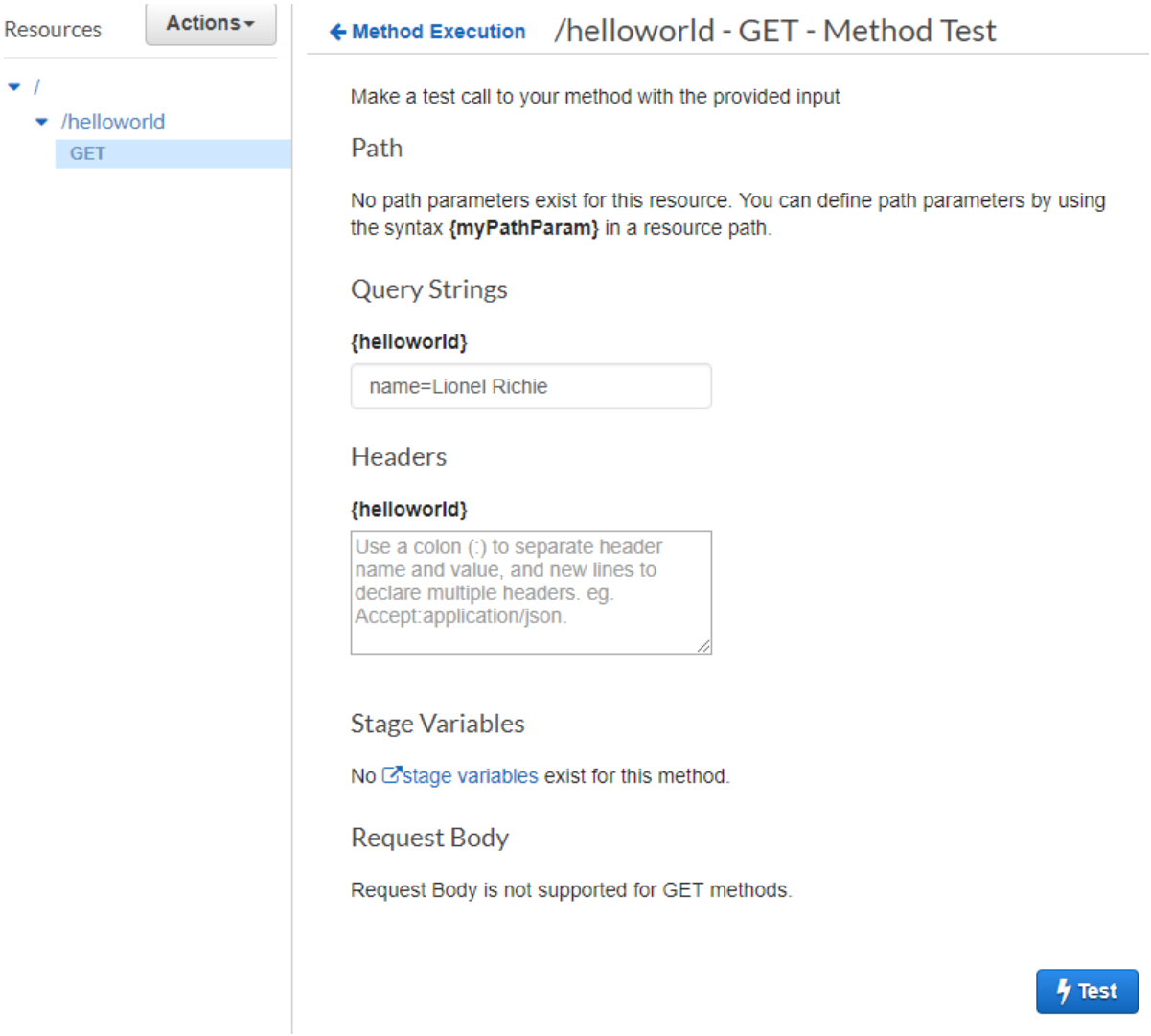
As you can **see from the logs** (and the response), the query string parameter is mapped to the name element in the payload of the event. *However, the response is simply returned as plain old text - since the function simply returns text*.
This can be easily solved using the **Integration Response mapping template**, in a similar fashion we have applied to the request:



Add a mapping template to take the entire response body and assign this to the JSON message element; the resulting response should be like:

{ “message” : “*The-actual-response-from-your-lambda-should-go-here…*”}

Test the GET method again, providing a proper query parameter (and possibly some other query parameters than only name):



Inspect the response message to verify the API is now properly responding JSON (response content + header)!

```
{
  "message": "Hello Lionel Richie!"
}
```

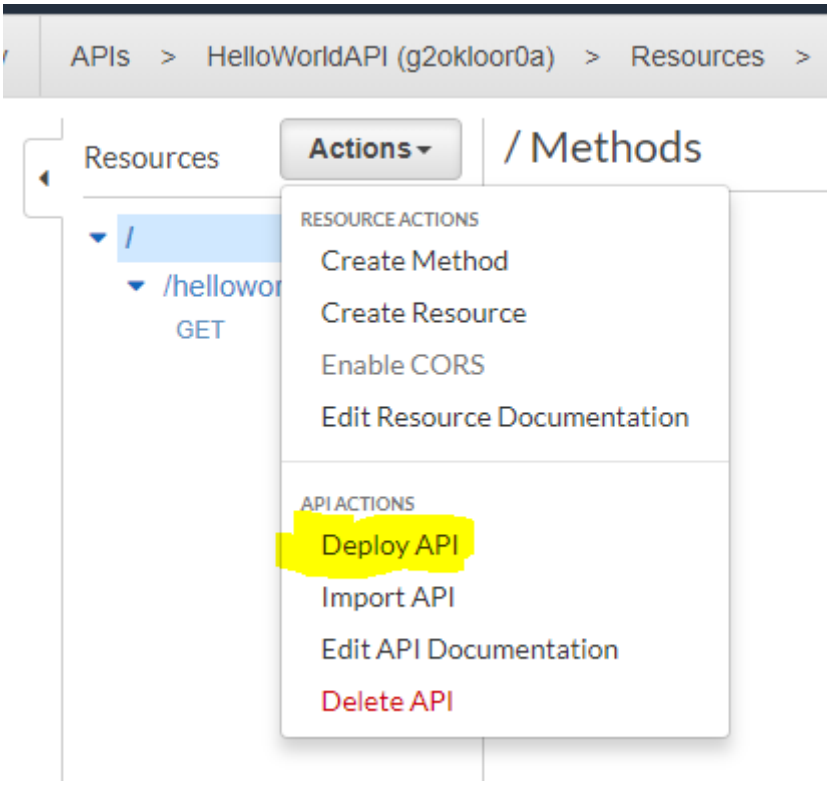
Response Headers

```
{"X-Amzn-Trace-Id":"Root=1-5b4a5ba8-7adcdff64f418131abe48e20;Sampled=0","Content-Type":"applicatio
n/json"}
```

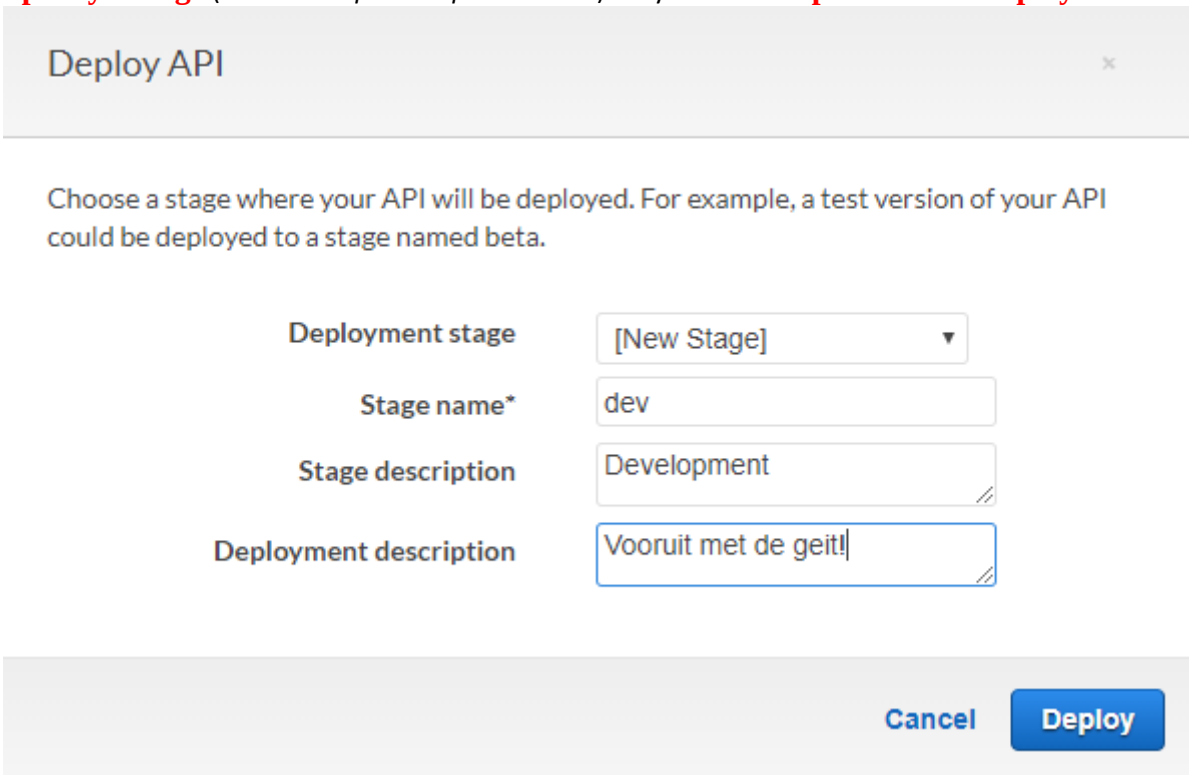
Check the Logs for the steps executed upon receiving the request!

Making the API accessible

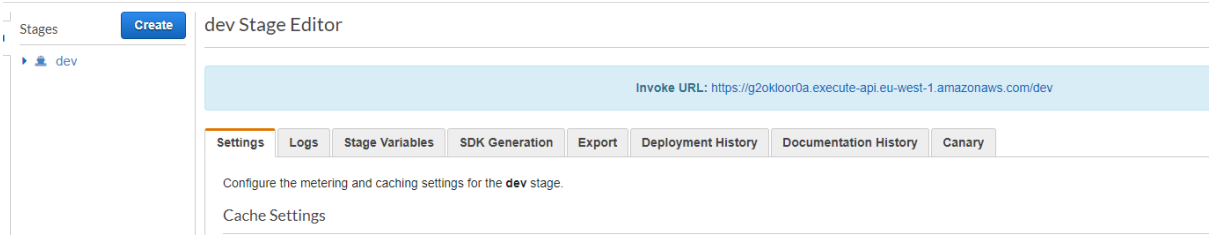
The final step in the process is actually deploying the API to the world:



Specify a stage (becomes a *path step* in the URL) for your API and **perform the deploy**:

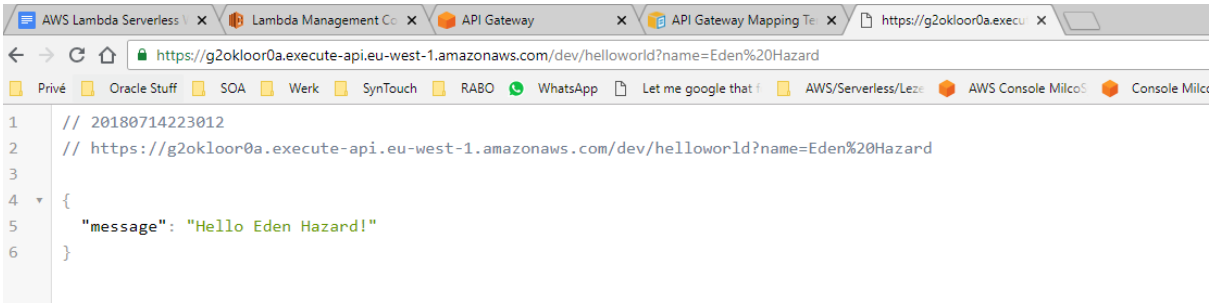


Examine the confirmation:



As you can see, here the stage "**dev**" is exposed at "https://g2okloor0a.execute-api.eu-west-1.amazonaws.com/dev". The **resource** for our API is helloworld and the GET request requires a name parameter, so you can test this api by going to https://g2okloor0a.execute-api.eu-west-1.amazonaws.com/dev/helloworld?name=Eden%20Hazard.

Test your own API using a GET request from a web browser to verify it is reachable and works as expected:



(The rendering in the screenshot above was down by my Chrome browser plugin "JSON Viewer").

Well done!