# CS 470 Module Four AWS Lambda and API Gateway Explained

In traditional software development in many programming languages, there are two primary components of any function in your application: the function signature and the body of code that implements the logic. Building serverless functions for the web is not much different. The REST API is the function signature, and body of code is your Lambda function.

Here’s an example with a main function from Java:

Main function in Java code :
public class Test {
  public static void main(String args[]) {
    System.out.println("Hello World");
  }
}


In the above code, **public static void main(String args[])** is the function signature. It says that this method, called “main”, takes an array of strings called “args”, and returns nothing. The code between the inner two braces ({}) is the function implementation. In this case, it says to print “Hello World” to the console.

Let’s look at a comparable example from the AWS Serverless world.

First, the function signature. Since we are using REST (see this [REST API Tutorial](https://restfulapi.net/) for a reminder on REST), we will need a base URI, a resource name, and a method. Our base URI will be “http://localhost” and our resource name will be “Hello”. But what method do we use? There is no real console to log to in the case of REST API. There is, however, the response. Since we will not be passing any parameters in, we are just getting a response – so we will use the GET method. So, our function signature will look something like **GET http://locathost/API**.

Second, we need a function body. Since we will be using NodeJS in this class, here is a method body in Node.

Method body in Node code:
exports.handler = function (event, context) {
  context.done(null, 'Hello World');
}

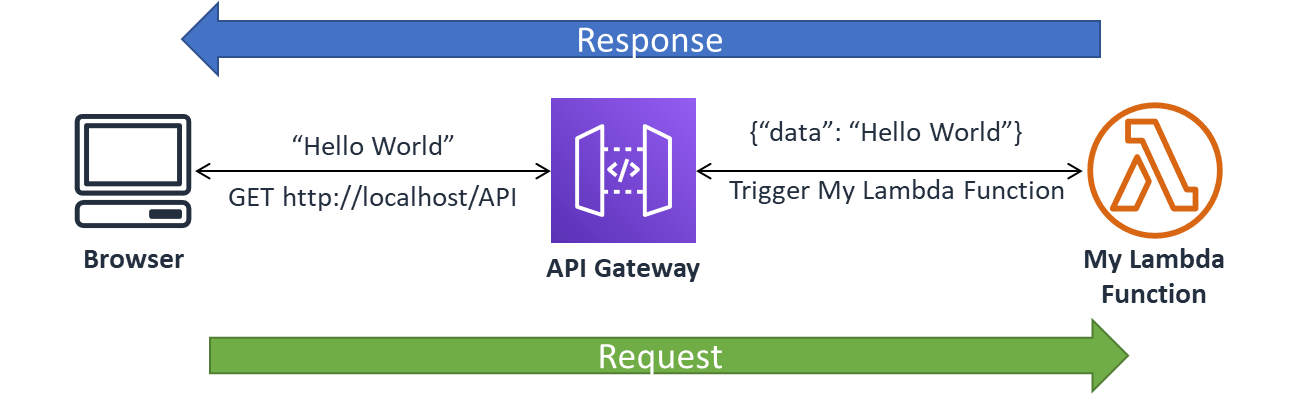

That method body will return “Hello World” to the web browser that made the API call.

You may be wondering how the API got to the Node, how the Node code knew how to return a result, and how the API knew how to handle the result. That is what the two AWS services API Gateway and Lambda handle for us. So, let’s explore a little deeper into how they work together.

## **AWS Lambda**

AWS Lambda allows you to run your code without having to provision any servers to run it on. In order to do that, Lambda needs a couple of things. It needs to know what language runtime you need, when to actually run your code, and what to call in your code when it starts. When you create your Lambdas, you will configure the language runtime you want. The second requirement is more interesting. AWS Lambda is based on the concept of Event Driven Programming. Event Driven Programming as a concept is beyond the scope of this class, but Martin Fowler gave [a good presentation at the GOTO Conference in 2017](https://www.youtube.com/watch?v=STKCRSUsyP0) (50:05) that explains much of it.

What you do need to understand is that the Lambda requires a trigger. Something must tell the Lambda when to start running. In our case, that is the API Gateway. Here is a diagram of the relationship of the systems involved.



The last thing the Lambda needs to know is how to start your code. In the case of NodeJS, that is done by exporting a method called “handler” (see the first line of the NodeJS example code above). If Lambda knows the language runtime, when to start, and how to start, then you have all you need to get started. You can learn more about what Lambda offers at [AWS’s website](https://docs.aws.amazon.com/lambda/latest/dg/welcome.html).

## **AWS API Gateway**

The API Gateway has one primary purpose: to map incoming API calls to a service to handle the API call. It can handle a few different flavors of APIs, such as REST, HTTP, and WebSocket, but we will focus only on REST APIs. It also has many advanced features for handling security, scale, canary testing (see [AWS Canary Deployment](https://wa.aws.amazon.com/wat.concept.canary-deployment.en.html)), Swagger/OpenAPI support, and so much more that you can read about in the [Amazon API Gateway Developer Guide](https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html). However, for this class, our focus is the creating of APIs manually in Module Four, and the use of [Swagger](https://swagger.io/docs/specification/2-0/what-is-swagger/) to automate much of the API work in Module Six.

In our use case, the API Gateway will receive a REST API request, forward that request to the correct Lambda function, and manage returning the result. It will also act as our security barrier between the outside world and our Lambdas. The outside world will only be able to access the APIs we deploy, and the API Gateway will be the only service with permissions to call our Lambdas.