# The Lambda Programming Model

Take a look at some of the JavaScript code which will be run by Lambda!

WE'LL COVER THE FOLLOWING

- `lambdaHandler`
  - Lambda interfaces in strongly typed languages

## `lambdaHandler` #

In the sample template, the combination of `CodeUri`, `Handler`, and `Runtime` means that the Lambda environment will try to execute the code using Node.js version 12 by calling the function called `lambdaHandler` inside `app.js` in the `hello-world` directory.

Next, you'll have a look at that file and inspect it, so you can see how Lambda responds to HTTP requests. That file will look similar to the following code:

```
let response;
exports.lambdaHandler = async(event, context) => {
        try {
                response = {
                        'statusCode': 200,
                        'body': JSON.stringify({
                                message: 'hello world',
                        })
                }
        } catch (err) {
                console.log(err);
                return err;
        }
        return response;
};
```

code/app/hello-world/app.js

> Note that SAM generates a source file with lots of comments and they have been removed from the previous example in order to focus on the

> more important aspects of the function. Also, different versions of SAM
>
> contain different starter templates, so the file might not look exactly like in this course, but the interfaces are the same. You can find the version used in this course in the source code package from https://runningserverless.com.

The Lambda function is *asynchronous* in JavaScript, meaning that it has to either be marked as `async` (line 2 in the previous example) or return a `Promise`. For other runtimes, that is not so important. (Previous versions of Lambda runtimes for Node.js also supported callbacks, but this is now discouraged and you will not learn about that approach in this course.)

All Lambda functions have two arguments:

- `event` represents the data sent by the client invoking the function. In this case, it will contain information about the HTTP request.
- `context` contains useful information about the runtime environment, such as the allowed execution time or logging setup.

> ## Lambda interfaces in strongly typed languages
>
> In strongly typed languages, such as Java, you can set up functions with specific request types and interfaces. In languages without strong typing, such as Python or JavaScript, these objects are mostly native key-value dictionaries or hash maps.

In the case of HTTP requests, the Lambda function needs to respond with an object containing the status code and the body of the HTTP response. API Gateway will assume that the response is a JSON object (hence the `JSON.stringify` call on line 6). You can modify this response to send back text or even HTML, which you will do that in Chapter 6.

Now you'll look at the deployment process in the upcoming lessons!