Setting up Cloud Functions

Launch cloud functions are a mechanism for you to provide backend functionality on your sites, and enable you to write, deploy, and run server-side code on-demand as API endpoints.

These functions are co-located with your frontend code and part of your Git workflow. As traffic increases, they automatically scale up and down to meet your needs.

Example use cases: CRUD operations with database, sending automated email, server-side input validation, etc.

You must code your functions in JavaScript and save them to the /functions directory in your project's root folder.

The function endpoint is determined by its file path relative to /functions.

Each JavaScript file to be deployed as a cloud function must export a handler function with the following general syntax:

```
export default function handler(request, response) {

// your server-side functionality

}
```

Example :

```
// functions/hello.js

export default function handler(request, response) {
response.status(200).json({
body: request.body,
  query: request.query,
  cookies: request.cookies,
});
}
```

The above example returns the request body, path query, and cookies in JSON format.

This function runs when you visit the URL path: /hello

Sometimes you may want to place extra code files, such as utils.js, inside the /functions folder. To skip turning these files into serverless functions, default exporting from such files is not supported. Below is an example of a valid utils.js file that can be placed inside the /functions directory.

Example:

```
//functions/utils.js

export function getShortName(name) {
  return name.slice(0, 3);
}

//functions/user.js

import { getShortName } from "./utils"

export default function handler(request, response) {
  const name = "Hilary";
  const shortName = getShortName(name);
  response.status(200).send({ name, shortName});
}
```

Node.js Request and Response Objects

Each request to a Launch cloud function gets access to Request and Response objects.

These objects are the standard HTTP Request and Response objects from Node.js.

Node.js Helpers

Additional helpers are provided inside the Request and Response objects passed to the function. These are:

Method	Description	Object
req.query	An object containing the request's query string, or {} if the request does not have a query string.	Request
req.cookies	An object containing the cookies sent by the request, or {} if the request contains no cookies.	Request
req.body	An object containing the body sent by the request, or null if no body is sent.	Request
res.status(code)	A function to set the status code sent with the response where the code must be a valid HTTP status code. Returns res for chaining.	Respons e
res.send(body)	A function to set the content of the response where the body can be a string, an object, or a Buffer.	Respons e
res.json(obj)	A function to send a JSON response where the obj is the JSON object to send.	Respons e
res.redirect(url)	A function to redirect to the URL derived from the specified path with status code 307 Temporary Redirect.	Respons e
res.redirect(statusCod e, url)	A function to redirect to the URL derived from the specified path, with the specified HTTP status code.	Respons e

Request Body

The req.body property populates with a parsed version of the content sent with the request when possible, based on the value of the Content-Type header as follows:

Content-Type header	Value of req.body
No header	Undefined
application/json	An object representing the parsed JSON sent by the request.
application/x-www-form-urlencoded	An object representing the form data sent with the request.
text/plain	A string containing the text sent by the request.
application/octet-stream	A Buffer containing the data sent by the request.

When the request body contains malformed JSON, accessing req.body will throw an error. You can catch that error by wrapping req.body with try...catch:

```
try {
const data = req.body;
}
catch (error) {
  res.status(400).json({ error: 'Invalid JSON in request body' });
}
```

Handling different HTTP methods

To handle different HTTP methods in a cloud function, you can use req.method in your handler as below:

```
export default function handler(req, res) {
if (req.method === 'POST') {
  // Process a POST request
```

```
  } else {
  // Handle any other HTTP method
}
}
```

Execution Timeout

Launch cloud functions enforce a maximum execution timeout. This means that the function must respond to an incoming HTTP request before the timeout has been reached. The maximum execution timeout is 60 seconds. If a request times out, the response error code would be 500.

Dynamic API Routes using Path Segments

Deploying cloud functions with Launch also gives users the ability to use path segments as file names.

When using path segments, any dynamic filename can be used; this is indicated by the use of square brackets ([ ]).

Note: The filename for the path segment is used solely for the purpose of providing a key name for accessing the value on the req.query object.

Example: functions/user/[name].js

Here, the value passed for the path segment is made available to the req.query object under the key used for the filename(name).

```
// functions/user/[name].js

export default function handler(request, response) {
  response.end(`Hello ${request.query.name}!`);
}
```

Running the function on the endpoint /user/Jake will return the response as Hello Jake.

Environment Variables

Users can access environment variables inside the cloud functions.

Environment variables can be added by going to the corresponding environment's Settings page on Launch.

Example:

```
export default async function handler(req, res) {
const uri = process.env.DATABASE_URI;
const client = new DBClient(uri); // perform database operations...
res.status(200).send("Success");
}
```

Note: A new deployment must be triggered after adding/modifying environment variables.

Debugging Cloud Functions - Server Logs

The Server logs section displays real-time logs generated by the cloud functions from the latest deployment.

The logs in Server Logs are not persistent and will be cleared if you refresh the page or if you perform a different action on the same page

Running Cloud Functions Locally

You can run your Launch project Cloud Functions locally using the launch:functions command in CLI.

Limitations

- Writing Cloud Functions using TypeScript is currently not supported.

Add API Endpoints to a Website in Launch

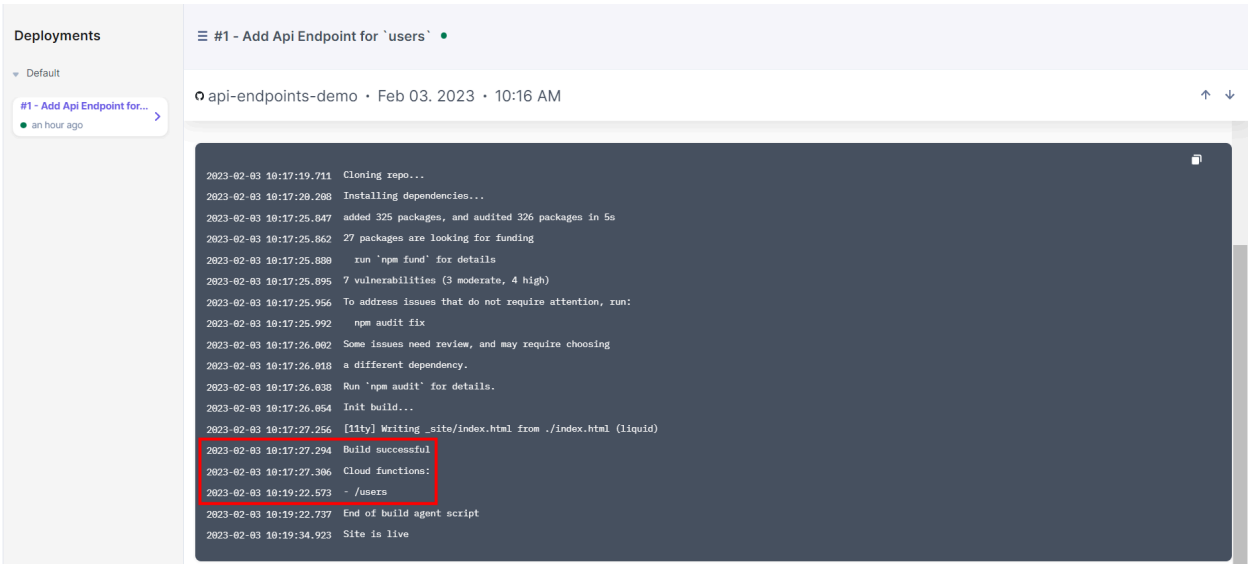
Launch allows you to write Cloud Functions to create API endpoints.

Steps for execution

Follow the steps to write a cloud function that can be used to add API endpoints to a website in Launch.

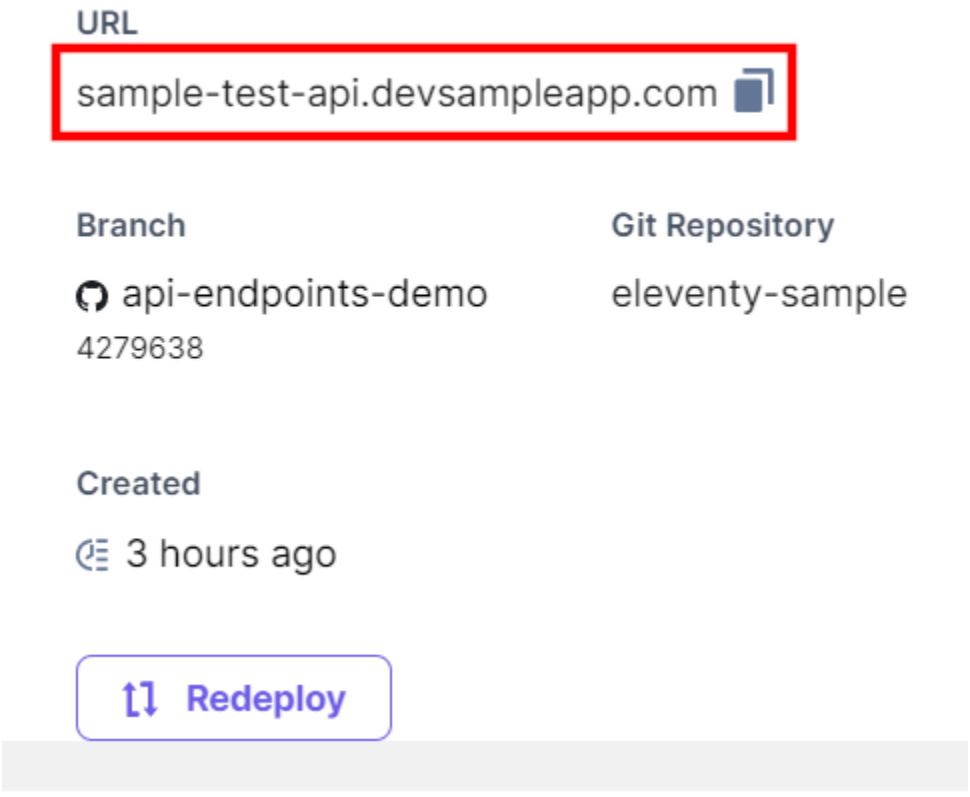
1. Create a folder named /functions in your project's root folder.
2. Create a JavaScript file to code your functions and then save the file to the /functions folder.
   Example:
   // functions/users.js
   export default function handler(request, response) {
   const users = [
   {name: "Jack", age: "25"},
   {name: "Rick", age: "28"},
   {name: "Jane", age: "34"},
   ];
   response.status(200).send(users);
   }
3.

   Deploy your project in Launch using one of the following methods:
   1. Import from a Git Repository
   2. Upload a file
4. After successful deployment, you will see the Cloud Functions displayed in the log:



The function endpoint is determined by its file path relative to /functions. The function runs when you visit the path: /users.

5. Copy the URL from the Deployment Information.



In this example, the URL of the API endpoint will be https://sample-test-api.devsampleapp.com/users.

6. On sending a request to the API Endpoint, you will see the following response:


```
[
{
"name": "Jack",
"age": "25"
},
{
"name": "Rick",
"age": "28"
},
{
"name": "Jane",
"age": "34"
}
]
```