Getting started with custom apps

CHAPTER 1: WHY CUSTOM APPS MATTER

1.1 Course introduction

1.2 APIs and Make

1.3 HTTP calls

1.4 Custom apps

1.5 Custom apps and APIs

1.6 How to build a custom app?

1.7 App visibility

1.8 Use case - HTTP call

CHAPTER 2: BUILDING THE CUSTOM APP

2.1 Custom app structure

2.2. Giving instructions

2.3 Getting started

2.4 Logo

2.5 Build it

2.6 Connections

2.7 Base

2.8 Modules

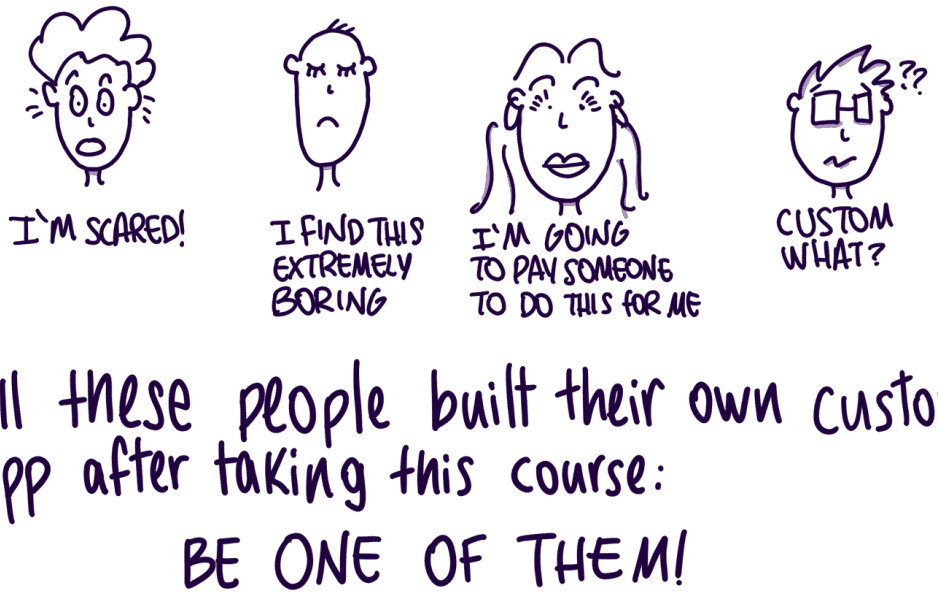
2.9 Search module

2.10 Testing your custom app

SUMMARY

Course complete

Welcome to the Getting started with custom apps course.



If you have never built a custom app before or never thought about it, you're in the right place!

In this course, you will explore and learn:

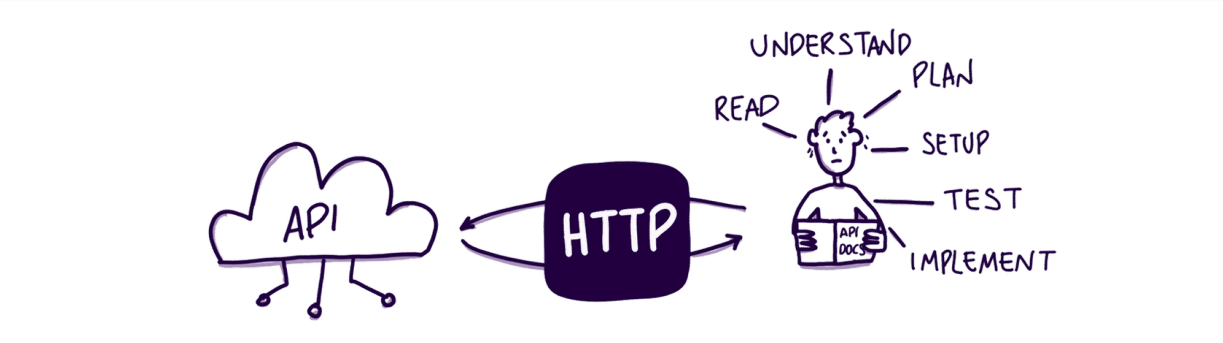
- what custom apps are and the benefits of using them
- an overview of the custom app elements and how they relate to HTTP calls
- how to build your first custom app

Discover this amazing feature that Make offers!

Note that everyone with a Make account can create custom apps. Yes, even with a free plan! Exciting, right?

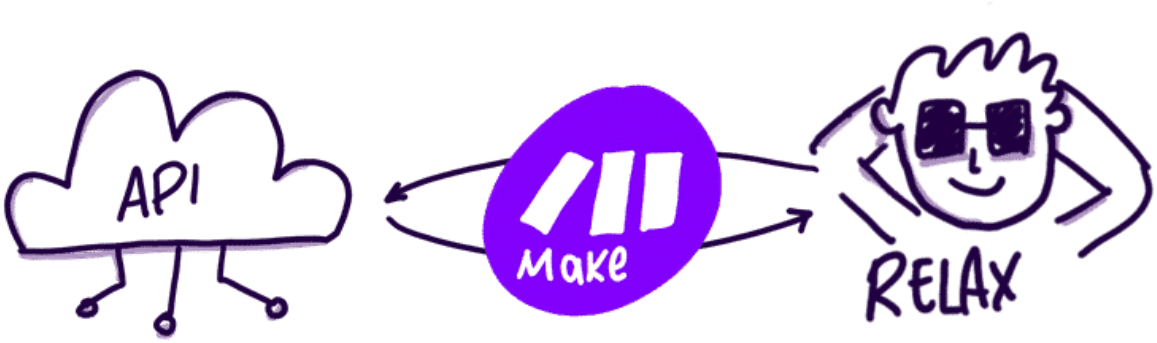
CONTINUE

1.2 APIs and Make



You know that an **API (Application Programming Interface)** is a set of rules and tools that allow different software applications to communicate with each other. **Endpoints** are specific URLs that define where to access a specific API service. Each endpoint corresponds to a particular function or action of the API.

To use an API, you need to read its documentation, which explains **how to use the API**, **how to authenticate**, **what data** is available, **how to request it**, and **what will be the format** of the response. You use HTTP to make the request.



In Make, this is much easier.

Make offers **apps** that integrate with third-party services, using **modules** to call their specific API endpoints.

These modules help you interact with APIs without dealing with the technical details, making everything nice and easy. The apps are ready for you to use them, no need to worry about a thing!

At the time of writing this course, there are nearly 2,000 apps available in Make (and counting).

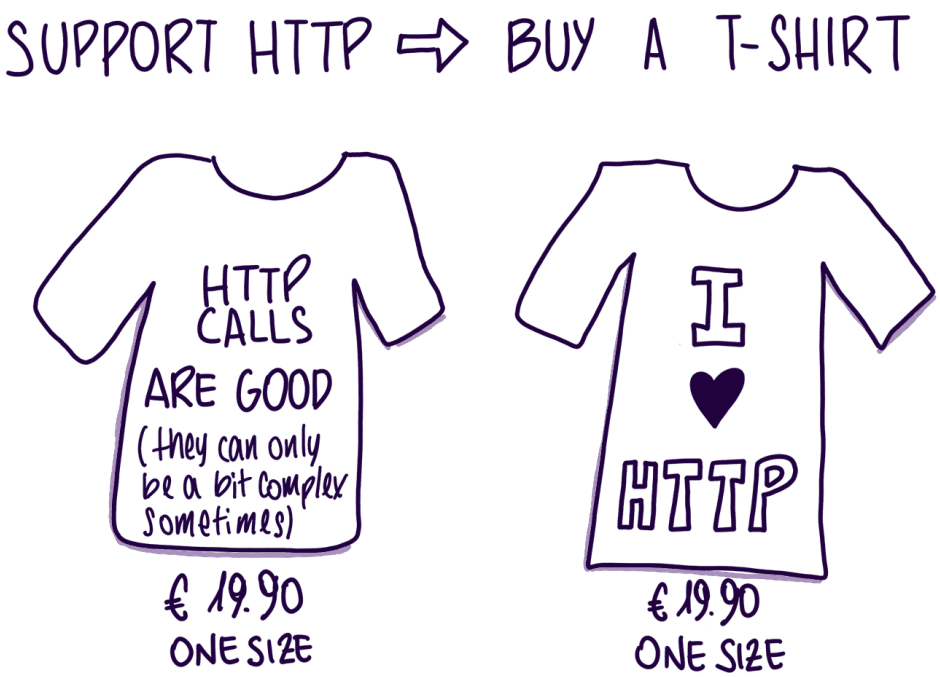
CONTINUE

1.3 HTTP calls

If an app isn’t available in Make, you can still use it in your scenario by using the **HTTP modules** to interact with an API.

You’ll need to **check the documentation**, set up the authentication, and include any required information in the request, like headers, query parameters, or body.

Once you send the request to the API endpoint, the app responds with data that you can use in your automation.



Using HTTP modules gives you the flexibility to include APIs that aren't available in Make in your scenarios. However, it comes with some limitations:

1. **HTTP calls are not easy for beginners.** Setting up HTTP calls in Make can be hard for people without technical knowledge. You need to understand API documentation, set up the authentication, and define all the parameters for the call. This is more complicated and less intuitive than using apps already present in Make, which simplify these steps with pre-built configurations.

2. **You might have to set up the same call in different scenarios.** If you use the same API in multiple scenarios, you will have to set up the HTTP call in each of them. It's not the end of the world and you can do it, but for sure you can think of a better use of your time.

3. **You can't filter or change the data you get.** HTTP calls return all the data the API sends. Even if you only need a small part, they don't allow you to filter or personalize the API response. For example, if you use OpenWeatherMap, you'll get a lot of weather details, even if you just want the temperature. If the API doesn't let you filter, you'll have to go through the data and pick out what you need. You basically have to live with it.

4. **Handling pagination is not automatic.** Some APIs split their data into multiple pages. With HTTP calls, you need to manually add extra steps to get all the data. This involves adding modules to handle pagination parameters and retrieve all data, as brilliantly explained in the course **Pagination - retrieving all data from APIs.** It is doable, but it will require more time to set it up, and you will use more operations when running your scenario, because of the additional modules that handle pagination.

5. **Complex authentication methods are harder to set up.** OAuth2 Authorization Code flow is easier to implement because Make has a built-in module for it. But other types of authentication can be more complex to implement from scratch and you might need to make multiple HTTP calls to get it working. For example, a token-based authentication, where you first get a token by logging in and then use it for other calls, can be done but requires multiple HTTP calls.

6

Credentials and sensitive data might not be protected. For example, if you're setting up a multi- step authentication using the **HTTP> Make a request** module, your credentials will appear in the history logs. At the moment, there's no way to hide it.

CONTINUE

1.4 Custom apps

You can take it a step further and **build your own custom app**. A custom app is an app that you build yourself, similar to the ones already available in Make. It connects to the API you choose and has the specific features or actions you need.

In your custom app, you basically **set up HTTP calls to the API's endpoints**, allowing you to have more control over customizing both the call and the response.

Everything you need to build a custom app is an **API**. You'll need to check the **API documentation** to get all the details to set up the request, like authentication, base URL, endpoints, methods, and parameters. Once you have this information, you can use it to create your custom app. This course will guide you through the process of building your first custom app.

Once created, you can use it in your scenarios and share it with others. Later in this unit, you'll learn more about how to make your app available and who can access it.

Note that to use your custom app you need to be able to install it in your Make organization. To do this, you must have one of the following roles in Make: **Owner**, **Admin**, or **App Developer**.

Click here for more information about these roles.

1.4.1 Why use custom apps?

Here are some advantages of using custom apps:

1. **Non technical users don't need to set up complex HTTP calls.** If you have non-technical team members, they won't need to read API documentation or set up HTTP calls. You can do the setup for them, and they can simply use the app directly in the scenario editor, making their life (and yours) way easier.

2. **You don't need to repeat the HTTP call setup in multiple scenarios.** If you call the same API across multiple scenarios, you can build a custom app and use it whenever you need it. The custom app eliminates the need to redefine the HTTP call parameters and settings in every scenario. It saves time, ensures consistency, and makes a better use of your time.

3. **You can filter or personalize the API response.** With a custom app, you can define which information from the data your API returns to share with the scenarios by filtering, setting a specific output, and organizing

the data. This makes it easier to get just the information you want, and use it in the next steps of your scenario.

4. **You can handle pagination.** Custom apps allow you to define how to retrieve all data in the case pagination is implemented by the API. This ensures that users can easily get complete datasets without manually setting up repeaters and multiple calls in your scenarios. This also saves you credits!

5. **You can implement more complex authentication methods.** With custom apps, you can handle more complex authentication methods (like all the OAuth2 flows, token refresh mechanisms, or multi-step authentication) without needing multiple calls in your scenario. Saving credits, again!

6. **You can implement complex calls**. For tasks that require multiple calls and modules, custom apps let you combine them into a single API call. For example, if you need to retrieve data, create a new document, and add information from the first document, you can build a custom app to do all of this in one module.

7. **You can create and personalize instant triggers.** Custom apps allow you to personalize instant triggers by providing more control over webhooks. With a custom app, you can define the data structure in advance, so you don’t need to make a call to set it up. You can also define specific actions when a webhook connected to a third party API is created or removed. Additionally, custom apps let you create a webhook that can be shared across multiple scenarios, making it easier to reuse and maintain.

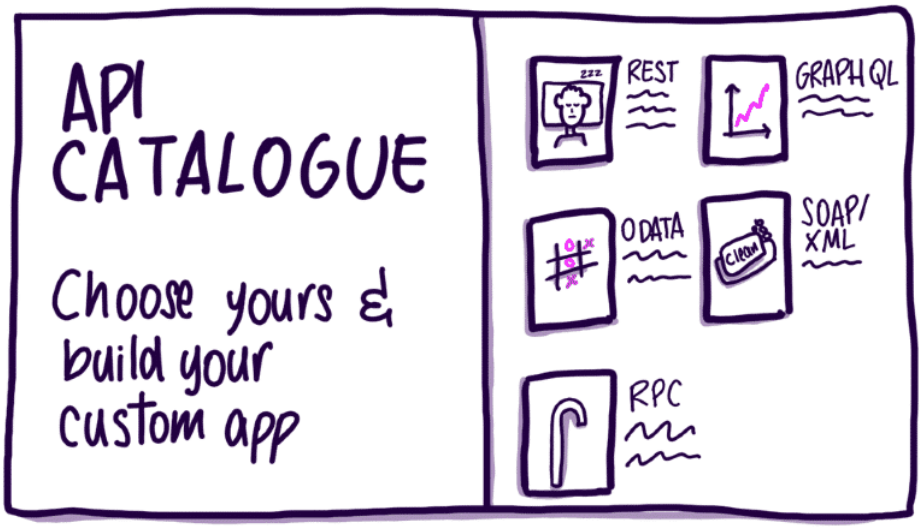
8. **You can share your custom app with other users**. Once you've built your custom app, you can make it available to users outside your organization, so they can use it without having to build it from scratch. Making their lives easier and making you the hero of the day.

CONTINUE

1.5 Custom apps and APIs

To be able to build a custom app you just need an API (with some documentation to know how to use it). That's it.

Different kind of APIs and services can be used when building custom apps:



- **REST API:** The classic one.
- **GraphQL**: Aiming to surpass REST API popularity. It provides a single endpoint and gives users more control over the data they request.
- **OData**: An extension of REST APIs. It uses standardized query options and data handling features, simplifying tasks like filtering, sorting, and interacting with data.

- **SOAP/XML**: Older and more complex method for interacting with third-party services. More verbose and complex than REST APIs. It uses XML documents to carry all the call information.

- **RPC**: The precursor of SOAP/XML. It uses larger XML or JSON documents.

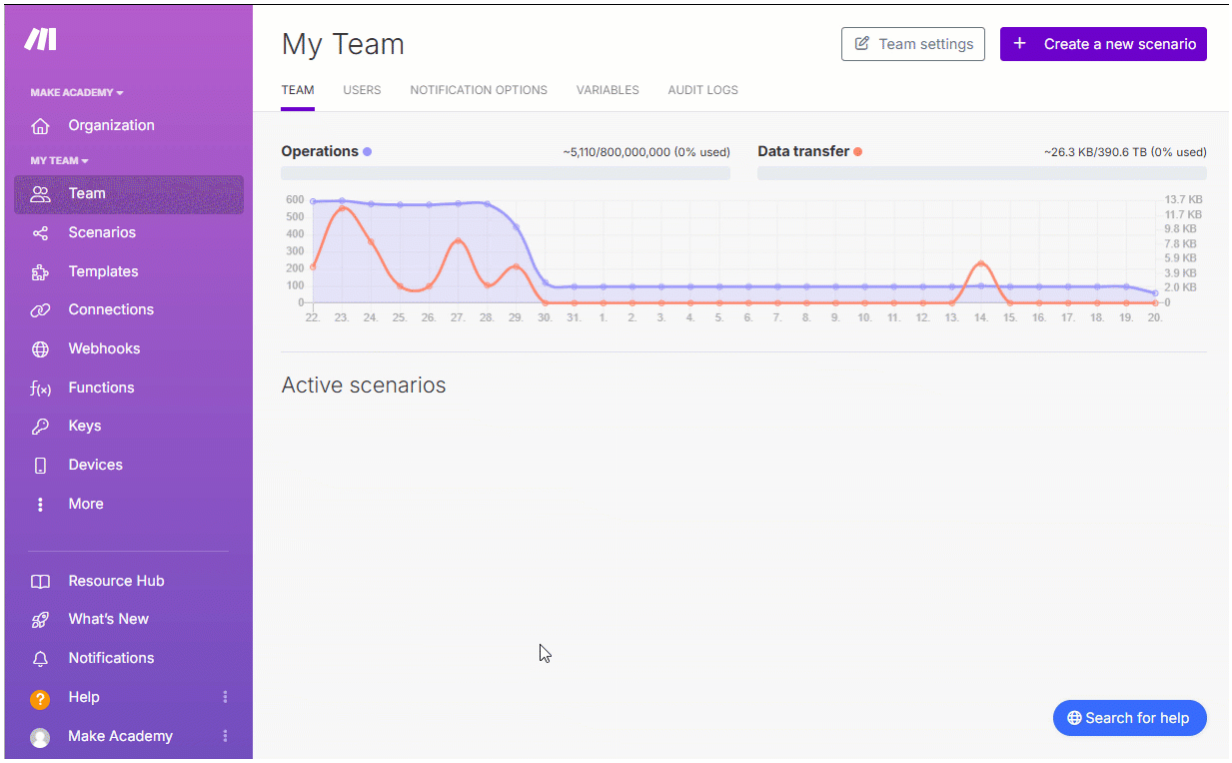
Disclaimer: The list above contains protocols that allow access to third-party resources that utilize HTTP and its mechanisms (methods, headers, query parameters, body, etc), and because of this they are considered as APIs.

CONTINUE

1.6 How to build a custom app?

You can build a custom app in Make in two ways: through the web interface within Make or by using Visual Studio Code.

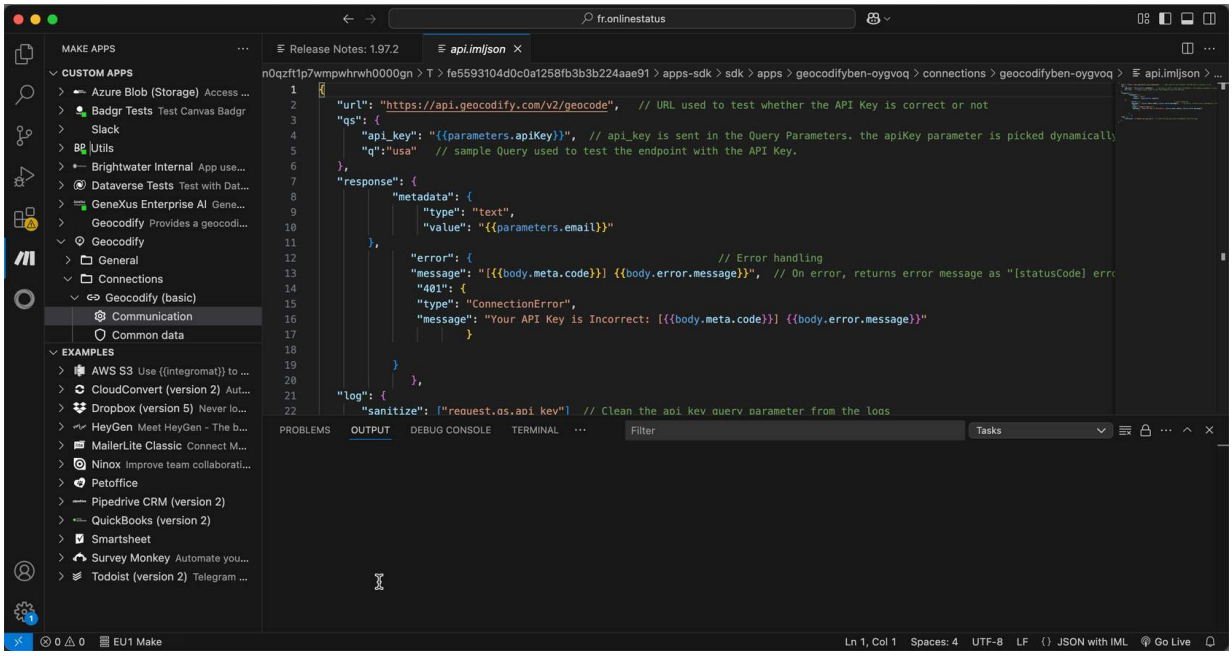
1.6.1 Web interface



This option lets you create and manage the custom app configuration **directly within the Make platform**. It offers a graphical interface, allowing you to easily build the app, by setting up connections, modules and parameters and personalizing the app behavior.

In this course, you will use the web interface to build your custom app.

1.6.2 Visual Studio Code (VS Code) extension



This option lets you set up the custom app using the **VS Code editor**.

The extension is perfect if you prefer working in a developer-friendly environment. You work with local folders and files, using features like syntax highlighting, code formatting, integration with version control systems like Git, and collaborative development.

It connects to the **Make API** to retrieve your custom apps and sync the sources between your computer and Make, enabling offline work and easier file sharing.

However, because it requires access to the **Make API**, it is not available for Make Free accounts.

Check the [documentation] if you want to learn more.

CONTINUE

1.7 App visibility

Your app can have different statuses that reflect its availability and ability to be used in the scenario editor.

Private

When you create a custom app, it is first set as **private**. This means only you, as the author, can see it in the scenario editor. You can use the app in your scenarios if you have the permissions to install apps in the organization (**Owner**, **Admin**, **App Developer** roles). However, other members of the organization won't be able to use the app in their scenarios until you create at least one scenario with it.

Note that you must enable the modules to make them visible so that others in your organization can see and use them in their scenarios.

Published (approval not requested)

To share your app with people outside of your organization, you need to publish it by clicking the **Publish** button.
Once published, you can share it with users from any organization using an **invitation link**. To install and use the app in the scenario builder, these users must have admin rights or permission to install apps (**Owner**, **Admin**, **App Developer**).

After publishing an app, it won't be possible to:

- Change the app back to private
- Delete the app
- Remove any component of the app (e.g. a module)

Approved app (globally available)

To make your app available to all Make users in their scenario editor, you must request a review of the app. The Make team will verify that it meets Make's **conditions** and **best practices**.

You can find more details about app visibility and the publishing process in the documentation.

APP VISIBILITY

CONTINUE

1.8 Use case - HTTP call

In this course you will build a custom app to call the Geocodify API.

Geocodify converts addresses into geographic coordinates (latitude and longitude), making it easier to locate places on a map.

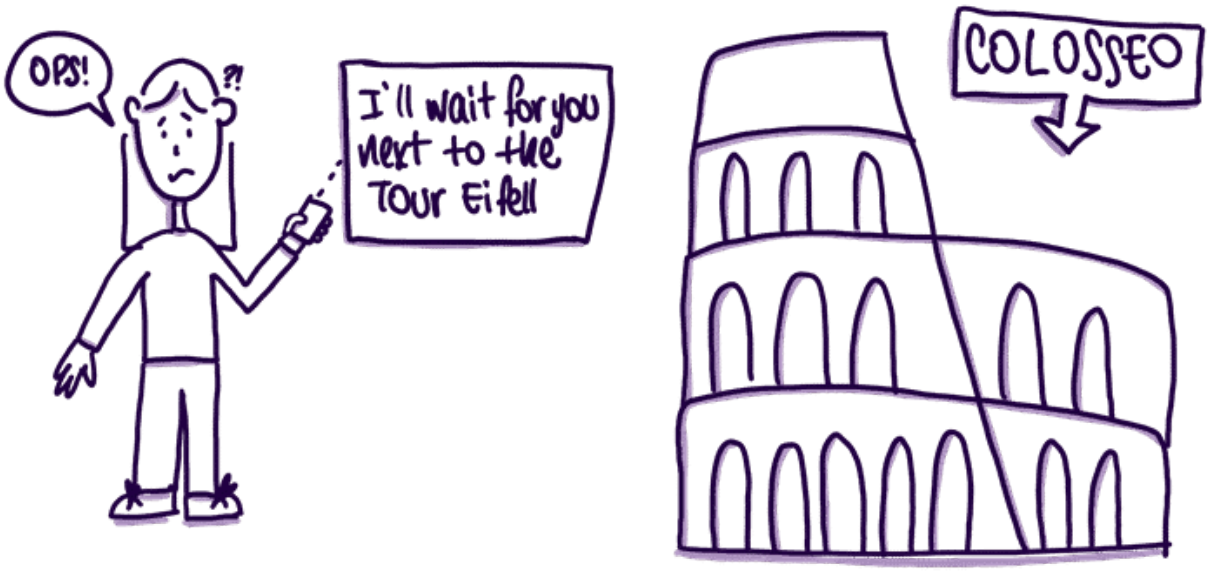
If you need to retrieve latitude and longitude for a specific address, Make doesn't provide a built-in app for that. You have two options: either make an HTTP call using the HTTP app or build a custom app.

In this unit you will start by setting up the HTTP call focusing on the different elements that you will use when building the custom app, which you'll explore in the next unit.

The main elements to consider are:

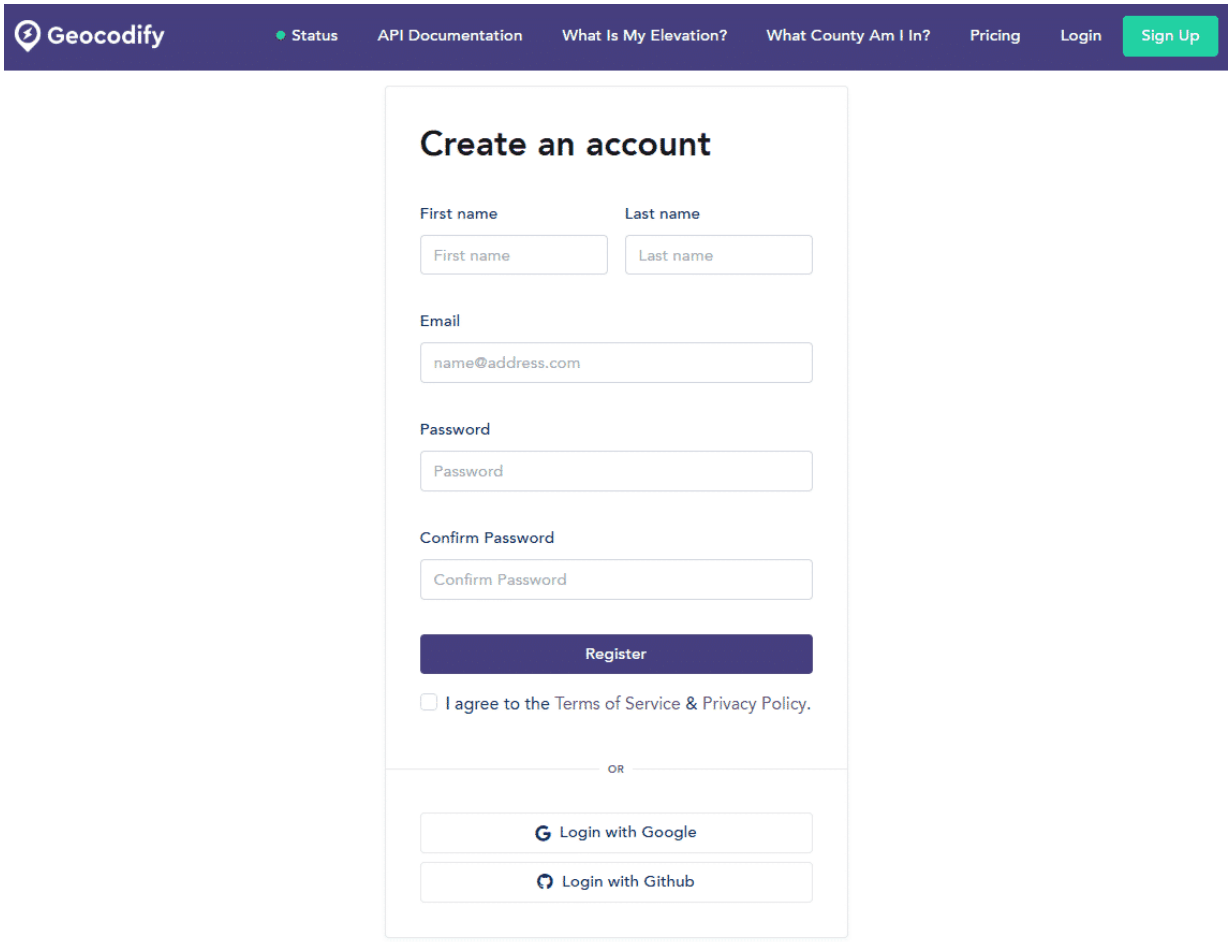
- **Authentication**
- **URL** (base + endpoint)
- **HTTP method**
- **Query parameters** (if present)
- **Body** (if present)
- **Headers**

This is a simple and straightforward app that doesn't require lots of work. However, it will help you understand the different elements and the advantages of setting up the call once, rather than having to repeat it every time you want to call the API.



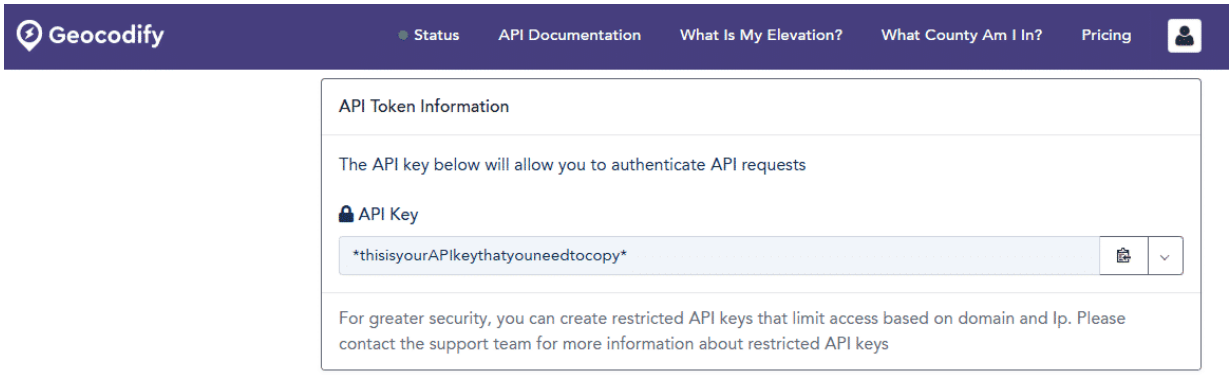
1.8.1 Preparation - API token

Before you begin, you need to get the API token from the **Geocodify API**.



On the Geocodify website, click **Sign Up** to create an account. Enter your details and click **Register** or **Login with Google**.

After logging in, you need to retrieve the API key that you will use when setting up the HTTP call.



Scroll down to find the **API Token Information** and copy your **API Key**.

1.8.2 API docs

To plan the HTTP call, start by studying the [API documentation], where you will find the information on the different elements of the API call.

Click each hotspot to learn more about the different elements.

Authentication

An API key is required for every request to the API. Your API key is used to authenticate you with our API, and it should be provided as a `api_key` URL parameter.

The API key can be retrieved from the account pages. If you do not know your API key please signup for a free account key.

```
?api_key=baa9dc110aa712sd3a9fa2a3dwb6c01d4c875950dc32vs
```

The `api_key` should be appended to the url request like in the example below

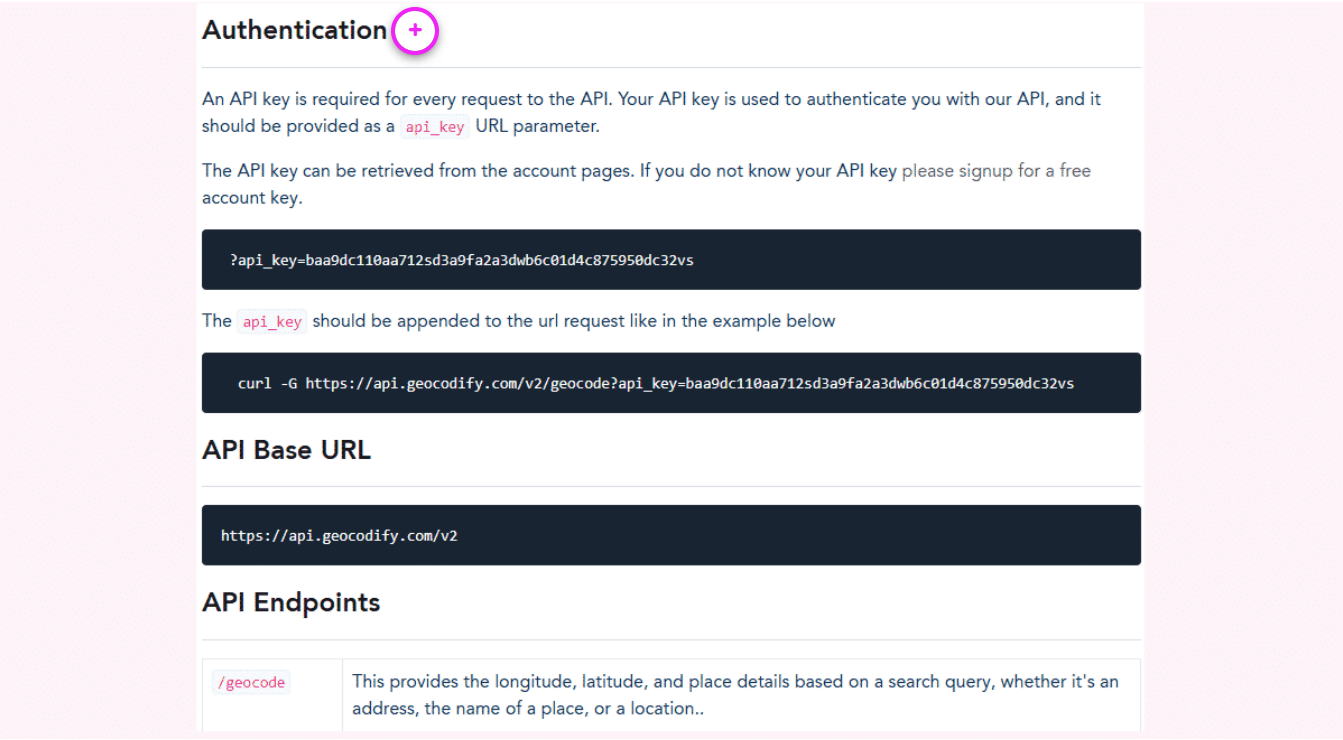
```
curl -G https://api.geocodify.com/v2/geocode?api_key=baa9dc110aa712sd3a9fa2a3dwb6c01d4c875950dc32vs
```

API Base URL

```
https://api.geocodify.com/v2
```

API Endpoints

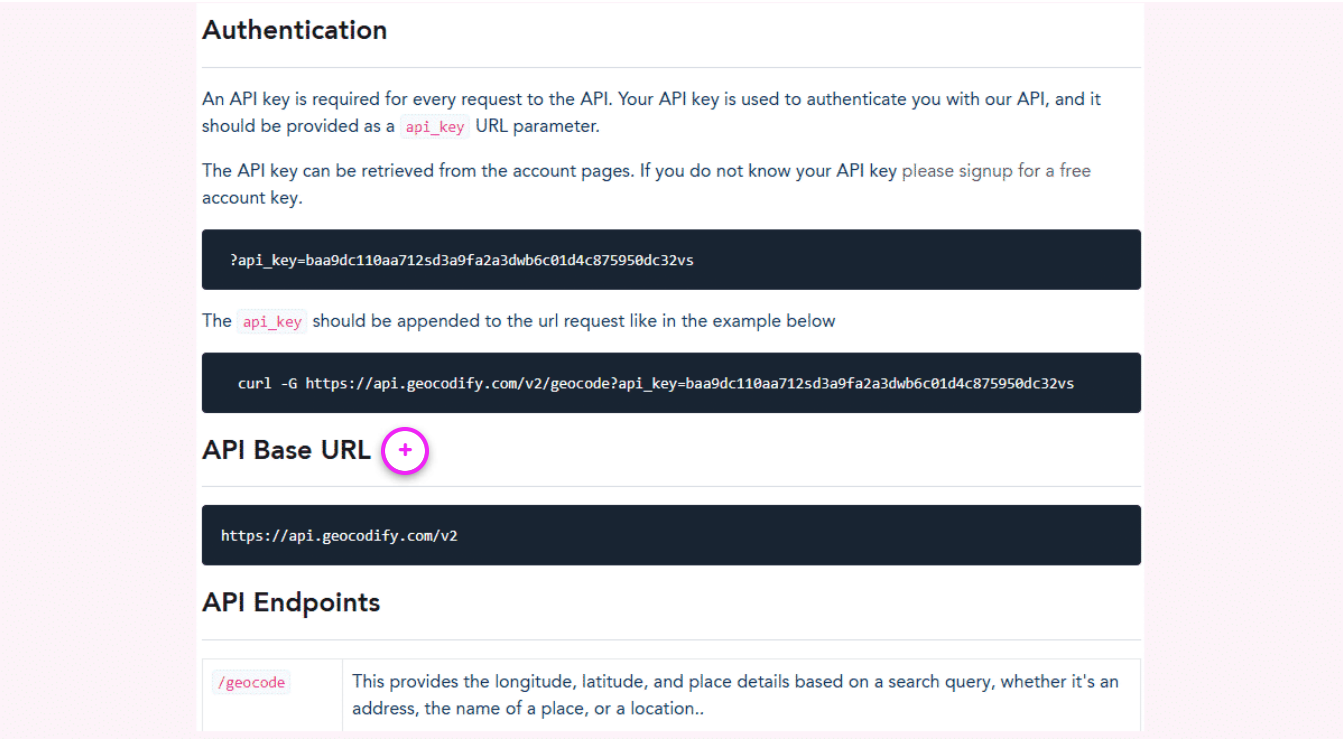
`/geocode`	This provides the longitude, latitude, and place details based on a search query, whether it's an address, the name of a place, or a location..



Authentication (1/3)

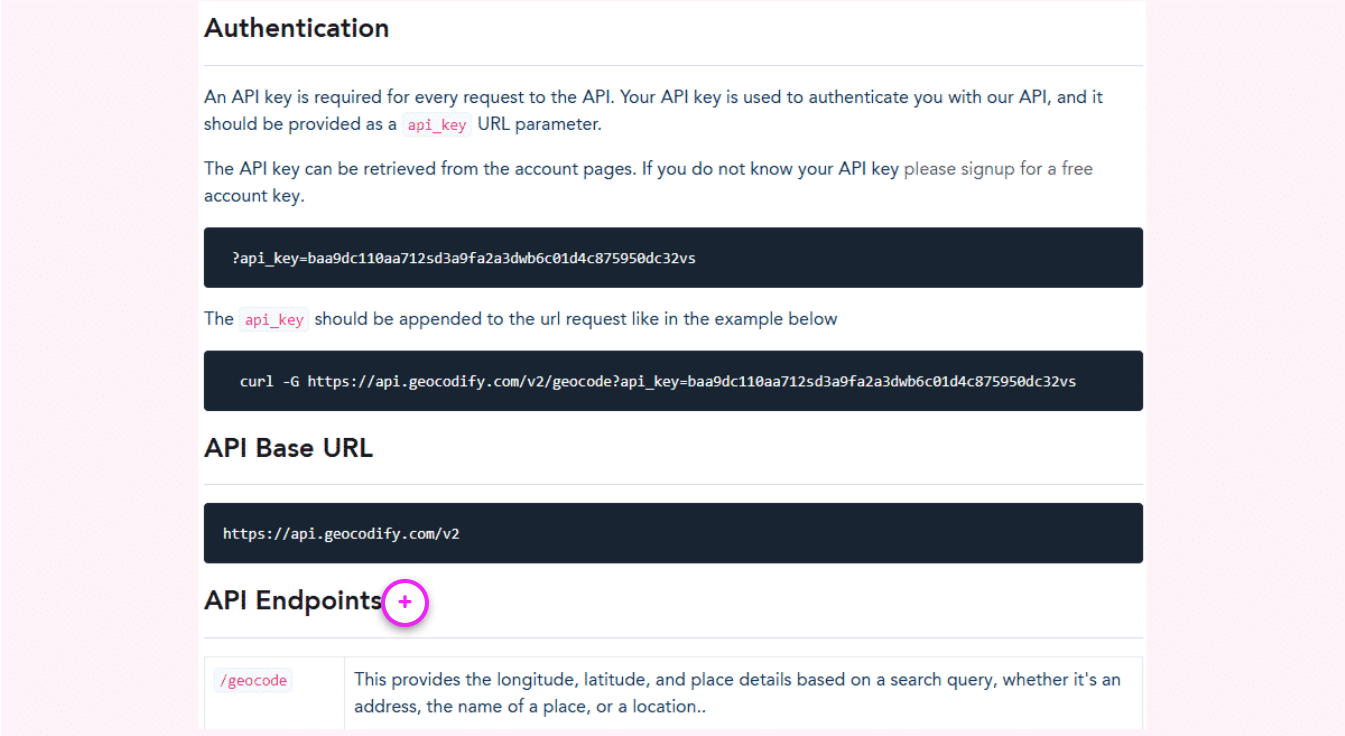
In this section you can find all the authentication details.

You can see that for this API you need to use an **API key**, which should be included in your request **query string** using the **api_key** parameter along with the value that you obtained from the Geocodify website.



API Base URL (2/3)

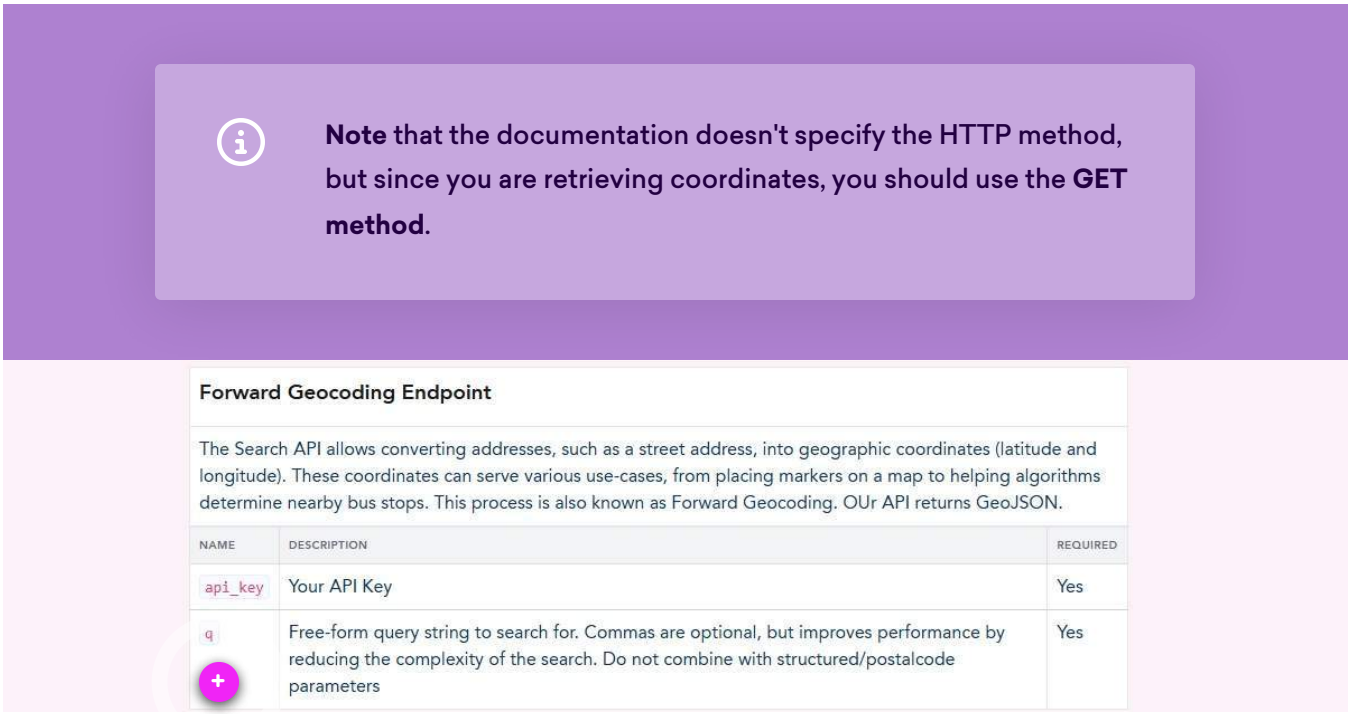
This section provides the **base URL** that you will use to access all the API endpoints.

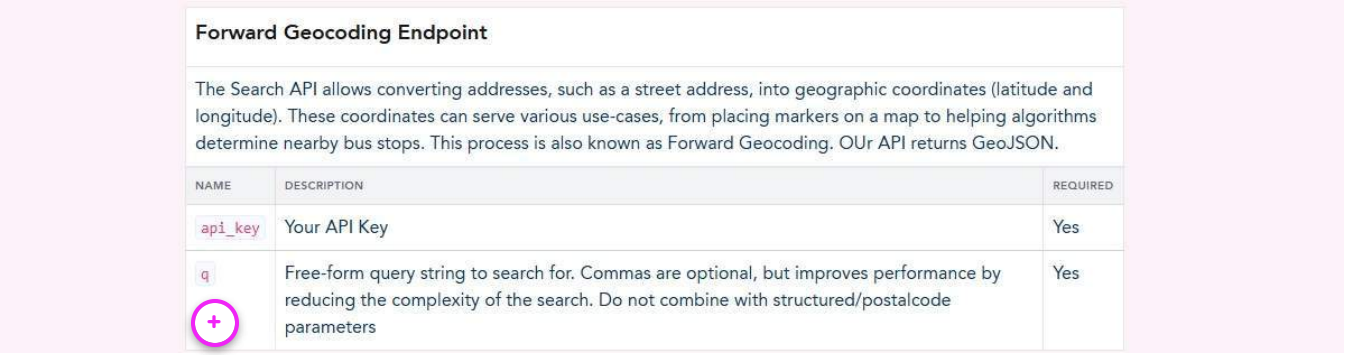


API Endpoints (3/3)

This section lists all the available endpoints.

For this use case, you will use the **/geocode** endpoint to get the coordinates of a specific address.





Forward Geocoding Endpoint

The Search API allows converting addresses, such as a street address, into geographic coordinates (latitude and longitude). These coordinates can serve various use-cases, from placing markers on a map to helping algorithms determine nearby bus stops. This process is also known as Forward Geocoding. OUr API returns GeoJSON.

NAME	DESCRIPTION	REQUIRED
`api_key`	Your API Key	Yes
`q`	Free-form query string to search for. Commas are optional, but improves performance by reducing the complexity of the search. Do not combine with structured/postalcode parameters	Yes

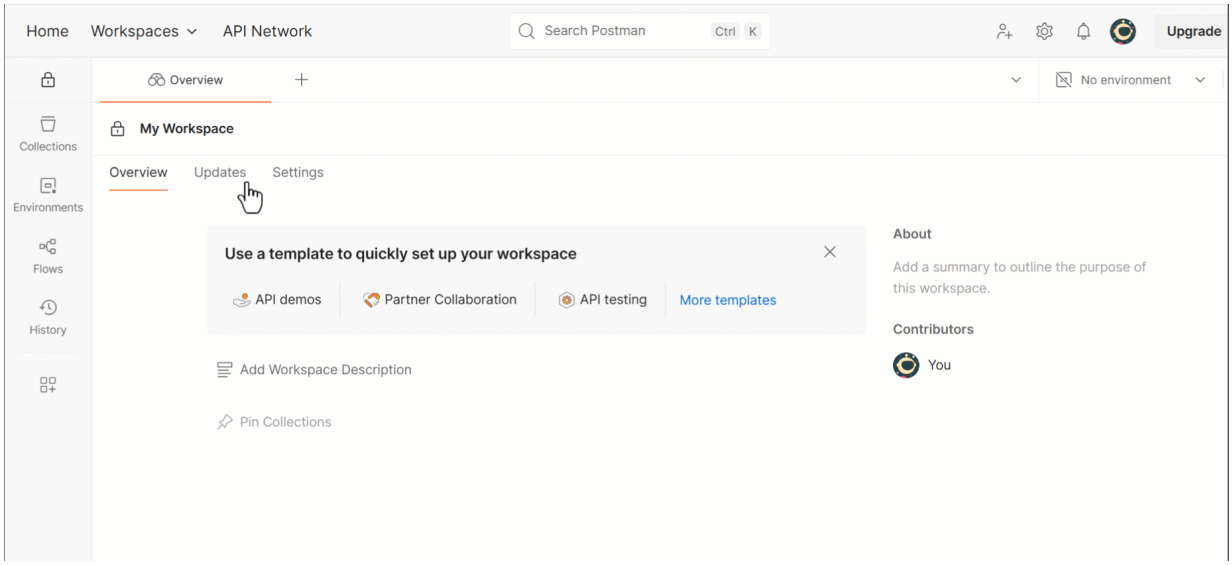
Query parameters

- **api_key:** The API key used for authentication.
- **q:** The address for which you want to retrieve the coordinates.

1.8.3 Postman test

It is good practice to test the HTTP call using Postman before building it in Make. This ensures that everything is working properly and helps you plan the HTTP call setup in Make.

In Postman, create a new request and add the necessary elements to make the HTTP call:



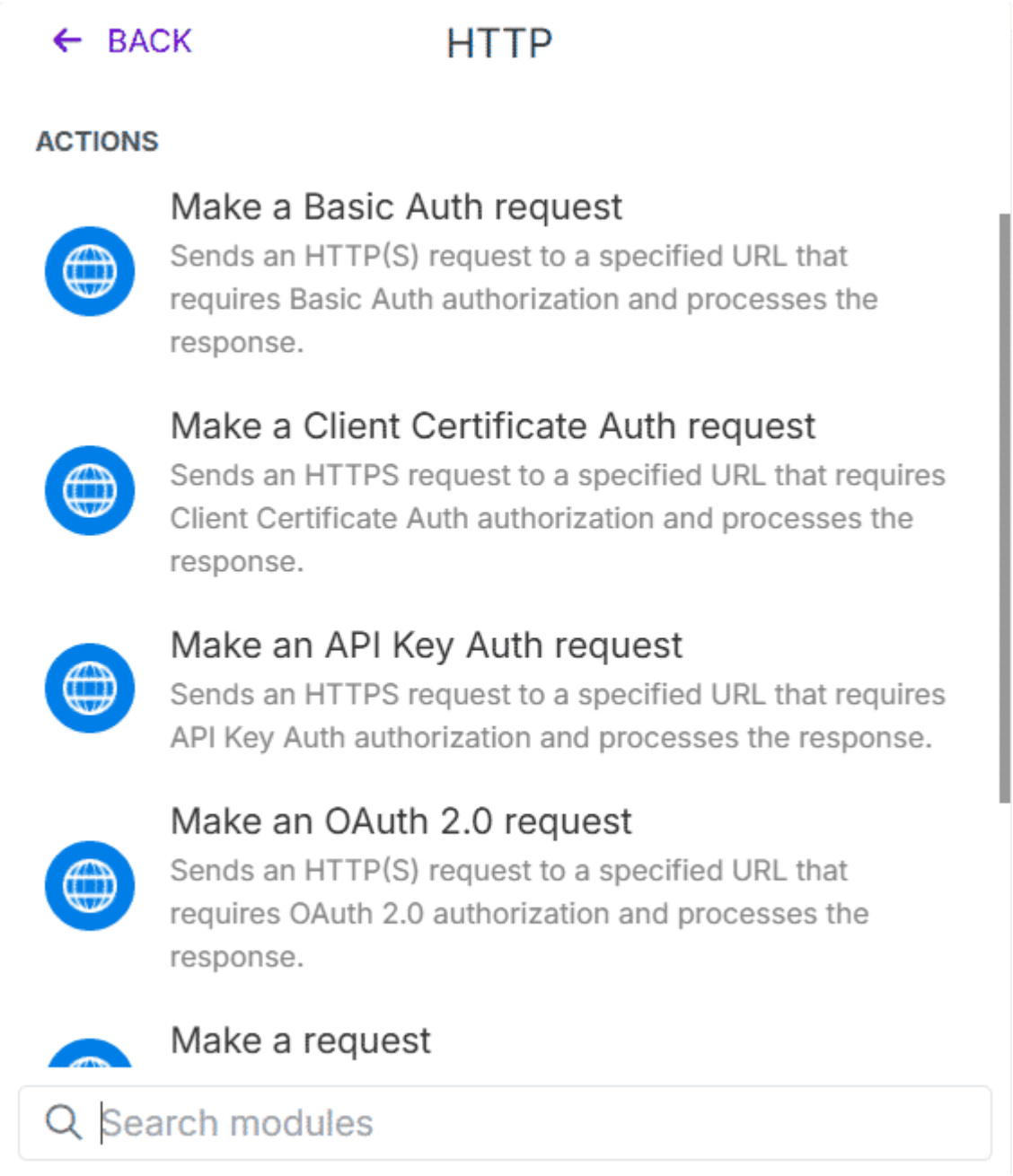
- **Method**: GET
- **URL** (base + endpoint): https://api.geocodify.com/v2/geocode
- **Query parameters**:
  - **api_key**: your API key
  - **q**: address you want to search, for example Champ de Mars, 5 Avenue Anatole France, 75007 Paris, France

Click **Send**. The API responds with a JSON file containing the address information and the API key.

If something is not working properly, use the error that the API returns to troubleshoot any issues.

1.8.4 HTTP call

If everything works fine in Postman, you can move on and build your call using one of the HTTP modules in Make. Use the information from the API documentation and the settings that you have used in Postman to set it up.



The choice of which HTTP module to use depends on the authentication method required by the API.

For Geocodify, you need to provide an API key as a query parameter.

While you could make a generic request, the best option here is to use the **HTTP> Make an API Key Auth request** module. This module lets you hide the API key, so it doesn’t appear in the scenario log and others can't see it when editing your scenario.

Now, let's set it up.

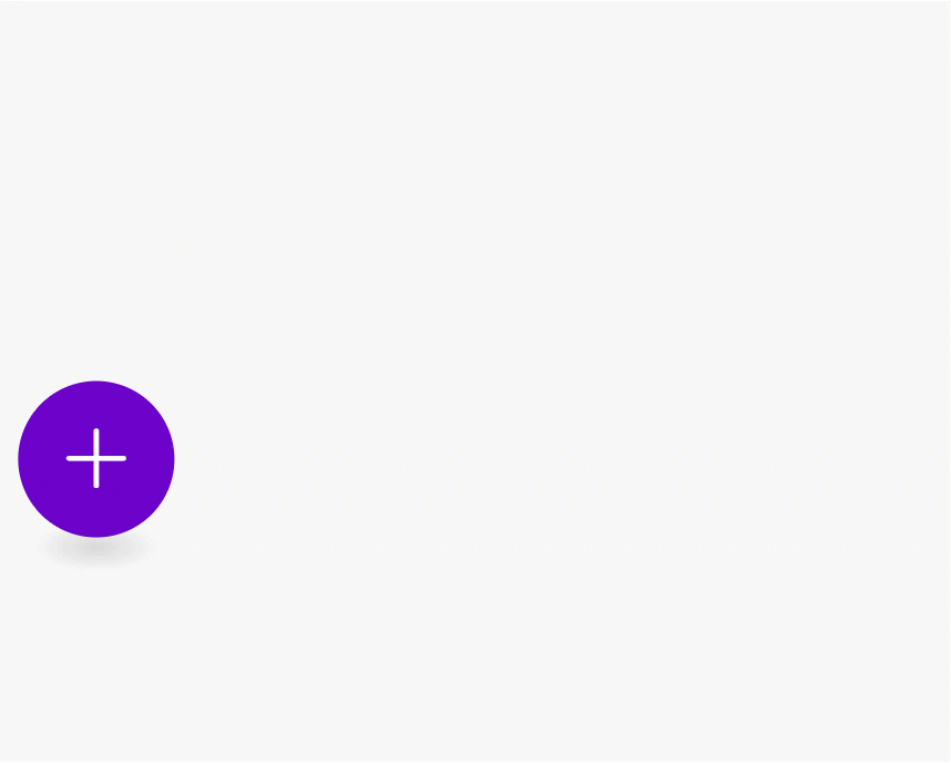
Note that the **Make an API Key Auth request** module allows you to create a new connection that you can use when making other calls to the same API.

Click Start and work through each step.

Set up the HTTP call

You can either follow along and build the HTTP call yourself, or simply have a look at all the steps needed to set up the call.

Step 1 - Authentication



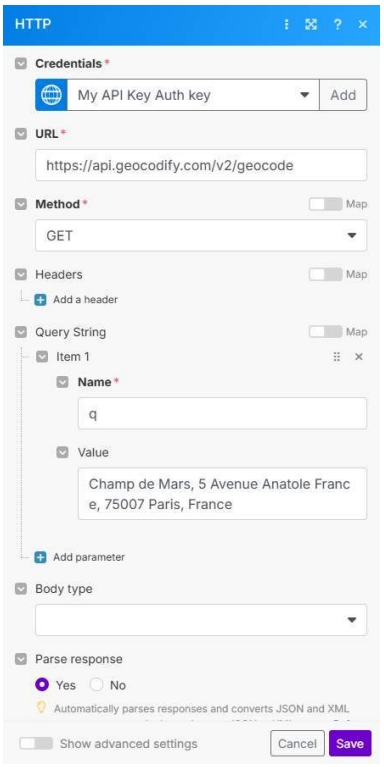
In Make create a new scenario and add an HTTP> **Make an API Key Auth request** module. Click **Create a keychain** to insert your credentials. Click **Add** to insert your API Key.

You can rename the connection if you wish, then paste the **API key** you obtained from the **Geocodify** app in the **Key** field.

For **API key placement** select **In the query string** and type **api_key** as the parameter name, as specified by the API documentation.

Click **Create** to save your settings.

Step 2 - Module set up



Now, you need to add all the necessary information for the API call, including the address for which you want to retrieve the coordinates:

- **URL** (base + endpoint): **https://api.geocodify.com/v2/geocode**
- **Method**: **GET**
- **Query string**: **q=Champ de Mars, 5 Avenue Anatole France, 75007 Paris, France**
- **Parse response** = **Yes** (this allows you to map the response items if needed)

Click **Save** to save the settings.

Step 3 - Run

Save your scenario and click **Run once**.

After your scenario runs, open the output bundles and you will see that the API has returned quite a lot of information.

You will have to dig a bit and find the coordinates under **Output> Data > Response > Features> 1> Geometry> Coordinates**.

With custom apps, you can personalize the output bundle to retrieve this information more easily.

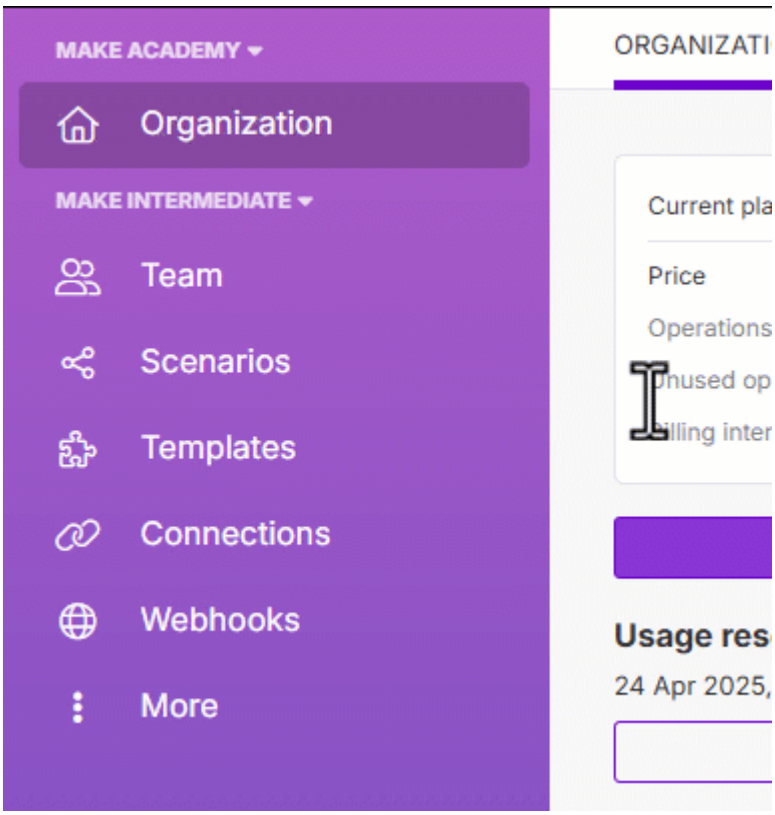
Now that you've set up the HTTP call to call the Geocodify app, it's time to create a custom app that has the same functionality. Move on to the next unit to begin exploring how to build a custom app.

CONTINUE

2.1 Custom app structure

Now that you know how your custom app should work, have reviewed the API documentation, and tested your HTTP call in Postman, it's time to start building it.

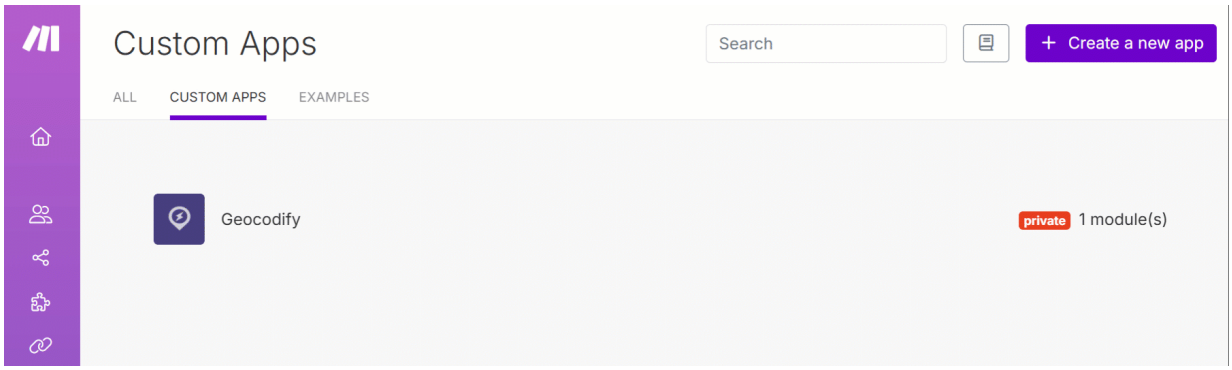
You'll use the web interface for this, and the first step is understanding the app's structure, including its components and tabs.



You can find the **Custom Apps** menu on the left side, where all your apps are listed, and you can create new ones.

For now, let's just explore the different **components** that make a custom app, you'll build the app later in the course.

The app is made of different **components** that define its behavior.



To set up your first custom app, you will focus on the following components, which are the minimum required to create a working app with one module:

- **Base**: define the settings that are common and inherited by all the modules
- **Connections**: configure the authentication settings to access the API and test that they are working properly
- **Modules**: set up the individual modules of your app

To define the settings of your components, you use the **tabs** nested within them. You will explore these tabs in more detail when setting up the components.

Note that the same structure applies when using VS Code.

CONTINUE

2.2 Giving instructions

Make has an **engine** that runs the custom apps (and all apps in Make).

To build your custom app, you need to interact with this engine, giving it instructions on which actions to take to make your app work.

To do this, you use **JSONC** (**JSON with Comments**) and **IML** (**Integromat Markup Language**) in the app builder. **JSONC** is **a version of JSON that allows comments**, unlike standard JSON. Comments can be **single-line** (**//**) or **multi-line** (**/* */**), just like in JavaScript. **IML** is a **language developed by Make** to enable the **use of functions within components**.

You use this to instruct the apps engine on the actions you want it to perform and to specify the parameters that the user will need to input.

Work through the images below to learn more.

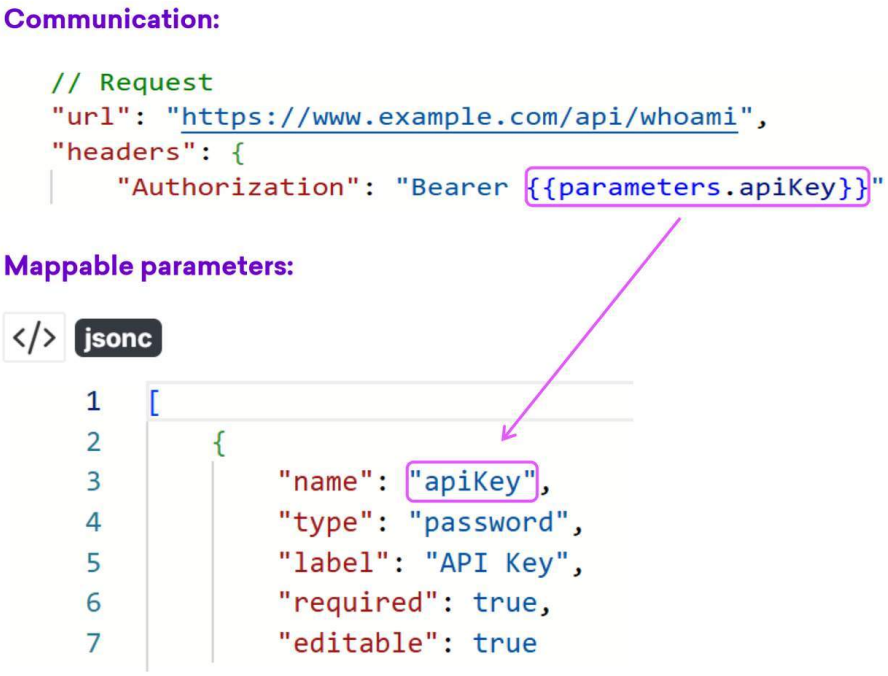
jsonc

```jsonc
{
    // Request
    "url": "https://www.example.com/api/whoami",          // Absolute URL to the API endpoint which validates credentials
    "headers": {                                          // Additional HTTP headers
        "Authorization": "Bearer {{parameters.apiKey}}"   // Authorizes user by API key, provided by user during the connection creatio
    },

    // Response handling
    "response": {
        "metadata": {                                     // Adds authorized user details to the connection label.
            "type": "email",                              // Type of the parameter. Can be "text" or "email".
            "value": "{{body.email}}"                     // The value in "email" will be displayed in connection's label.
        },
        "error": {                                        // Error handling
            "message": "[{{statusCode}}] {{body.error}}"  // On error, returns error message as "[statusCode] error text".
        }
    },

    "log": {
        "sanitize": [                                     // Excludes sensitive parameters from logs.
            "request.headers.authorization"               // Omit HTTP header "Authorization".
        ]
```

You use specific keywords as **directives** to instruct the apps engine to perform specific actions. For example, if you want the apps engine to access a specific URL, you would use: "url": "https://gotothispage.com".

You can find more information about the available directives at **this link**.

To make a request, you must specify at least a URL. The other directives are optional.

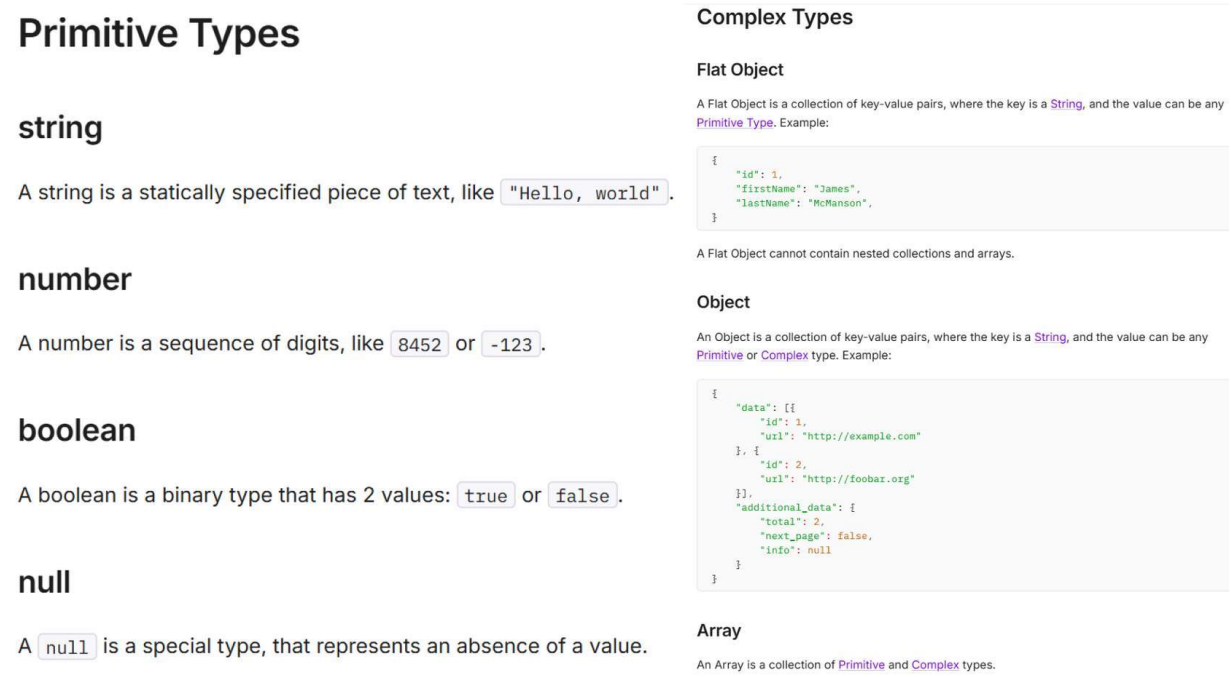


You can use JSONC to define **parameters**, which are values provided by the user of the app and used by the directives. Each parameter is defined by the settings that describe its properties. You can find the common parameter settings at **this link**.

Within the same component, you can access any parameter using the notation **{{parameters.name}}**, like for example **{{parameters.apiKey}}**.

When writing JSONC, you can use various data types that are derived from standard JSON data types. You can find more information about the data types at **this link**.

These are the ones you'll need for this course:



Primitive Types

string

A string is a statically specified piece of text, like `"Hello, world"`.

number

A number is a sequence of digits, like `8452` or `-123`.

boolean

A boolean is a binary type that has 2 values: `true` or `false`.

null

A `null` is a special type, that represents an absence of a value.

Complex Types

Flat Object

A Flat Object is a collection of key-value pairs, where the key is a String, and the value can be any Primitive Type. Example:

```
{
    "id": 1,
    "firstName": "James",
    "lastName": "McManson",
}
```

A Flat Object cannot contain nested collections and arrays.

Object

An Object is a collection of key-value pairs, where the key is a String, and the value can be any Primitive or Complex type. Example:

```
{
    "data": [{
        "id": 1,
        "url": "http://example.com"
    }, {
        "id": 2,
        "url": "http://foobar.org"
    }],
    "additional_data": {
        "total": 2,
        "next_page": false,
        "info": null
    }
}
```

Array

An Array is a collection of Primitive and Complex types.

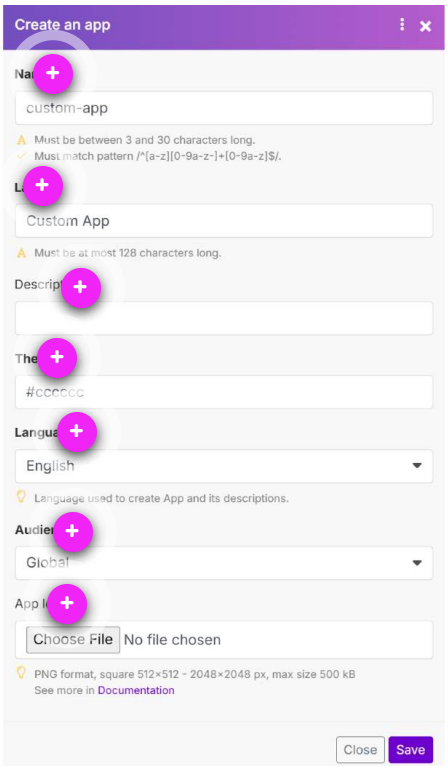
CONTINUE

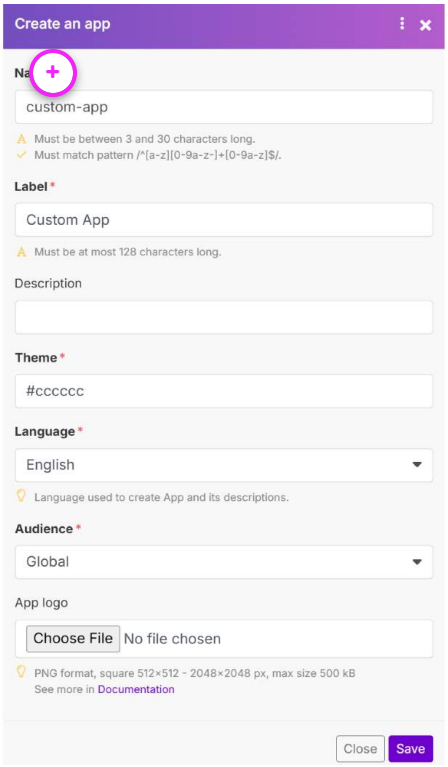
2.3 Getting started

Time to get your hands dirty and start building the Geocodify app.

Go to the **Custom Apps** menu and select **+ Create a new app**. In the pop-up window you will need to fill in the app details. Let's start by having a look at the information you will need to provide.

Click each + to learn more.

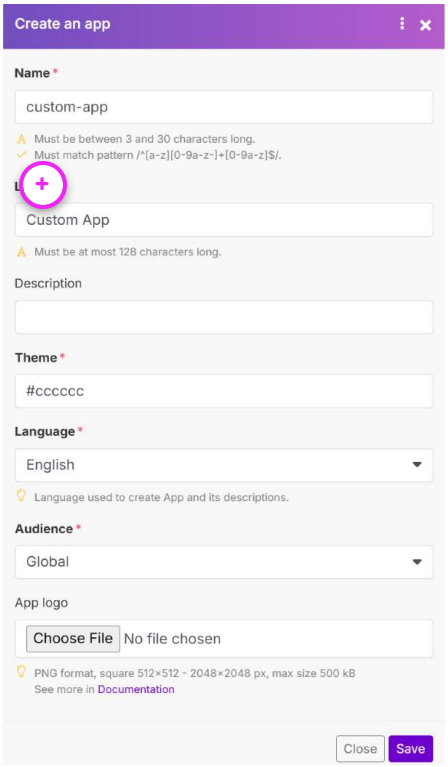




1/7 Name

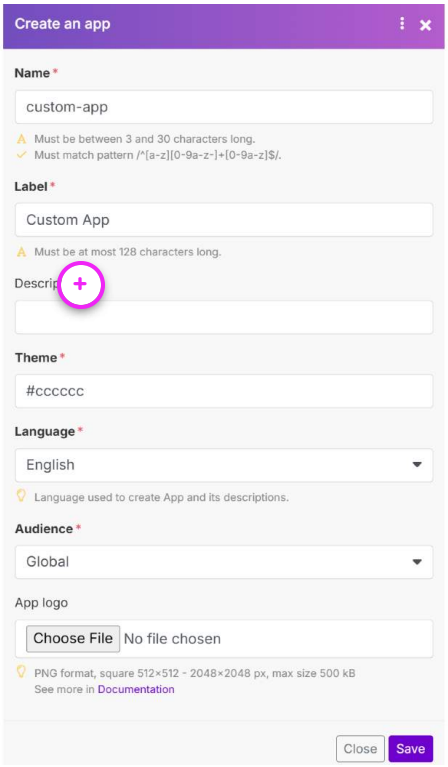
Unique identifier for your custom app. It is an **internal** name and is not visible in the scenario builder.

Check the Regex below the box to see the character requirements.



2/7 Label

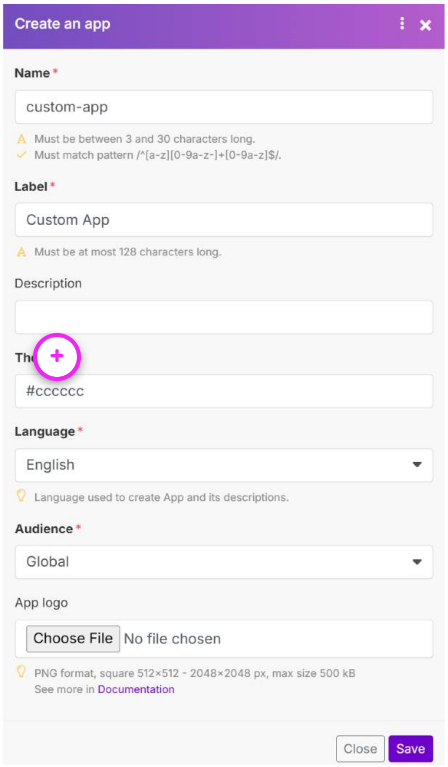
Name of the custom app in the scenario editor. It doesn't have any character restriction, except for the maximum length of 128 characters.



3/7 Description

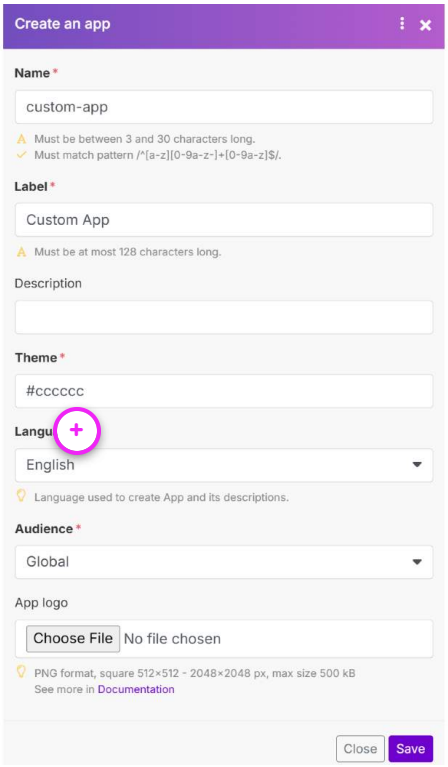
[Optional]

Description of the custom app.



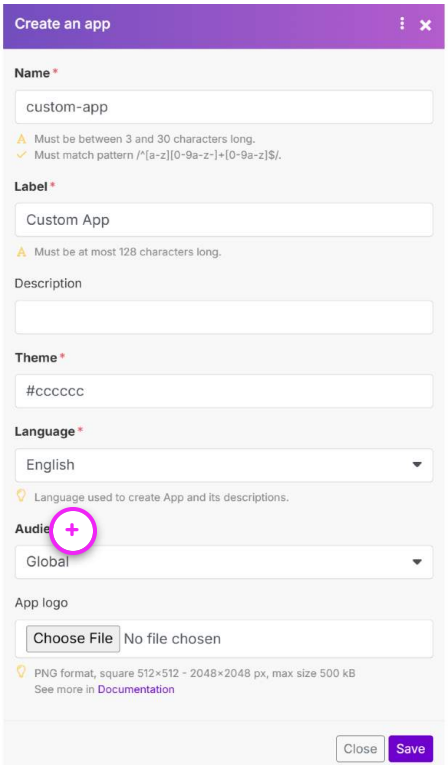
4/7 Theme

Color of the app in the scenario editor. It is specified using a hex code.



5/7 Language

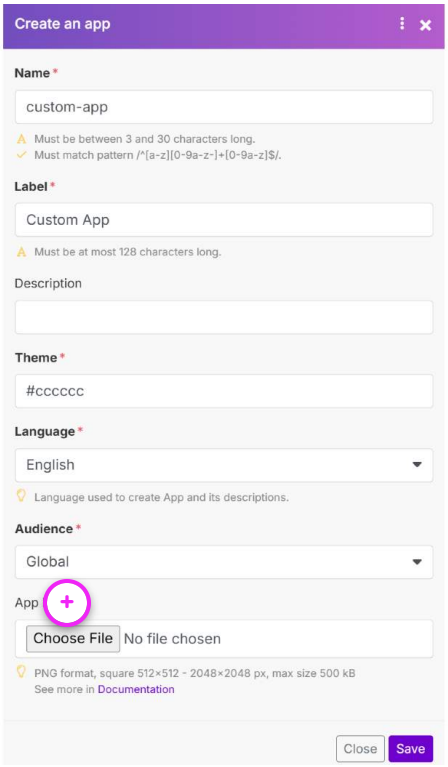
Language of your app. This value is just for your information, Make doesn't automatically translate the app.



6/7 Audience

Where the app is available.

Note that at the moment this parameter doesn't have any effect.



7/7 Logo

[Optional]

Logo of the app used in the scenario editor.

CONTINUE

2.4 Logo

Before starting to fill in all the fields, let's look at the logo requirements and make sure you have a logo for your app.

These are the logo requirements:

- an **image** file in **.png** format
- square dimensions: minimum **512 × 512 px** and maximum **2048 × 2048 px**
- a maximum file size of **512 kB**

Make processes the icon file by adjusting its colors.

- **White** or **transparent areas** in the logo will appear in the color set in the **Theme field**.
- **Black areas** will be converted to **full opacity** and shown as **white**.
- **Colored** or **semi-transparent areas** will be displayed in a **shade between white** and the **Theme field** color.

For this exercise, **download the logo below**.

If you want to create your own, you can use [LunaPic] to edit your photos for free.

FILE

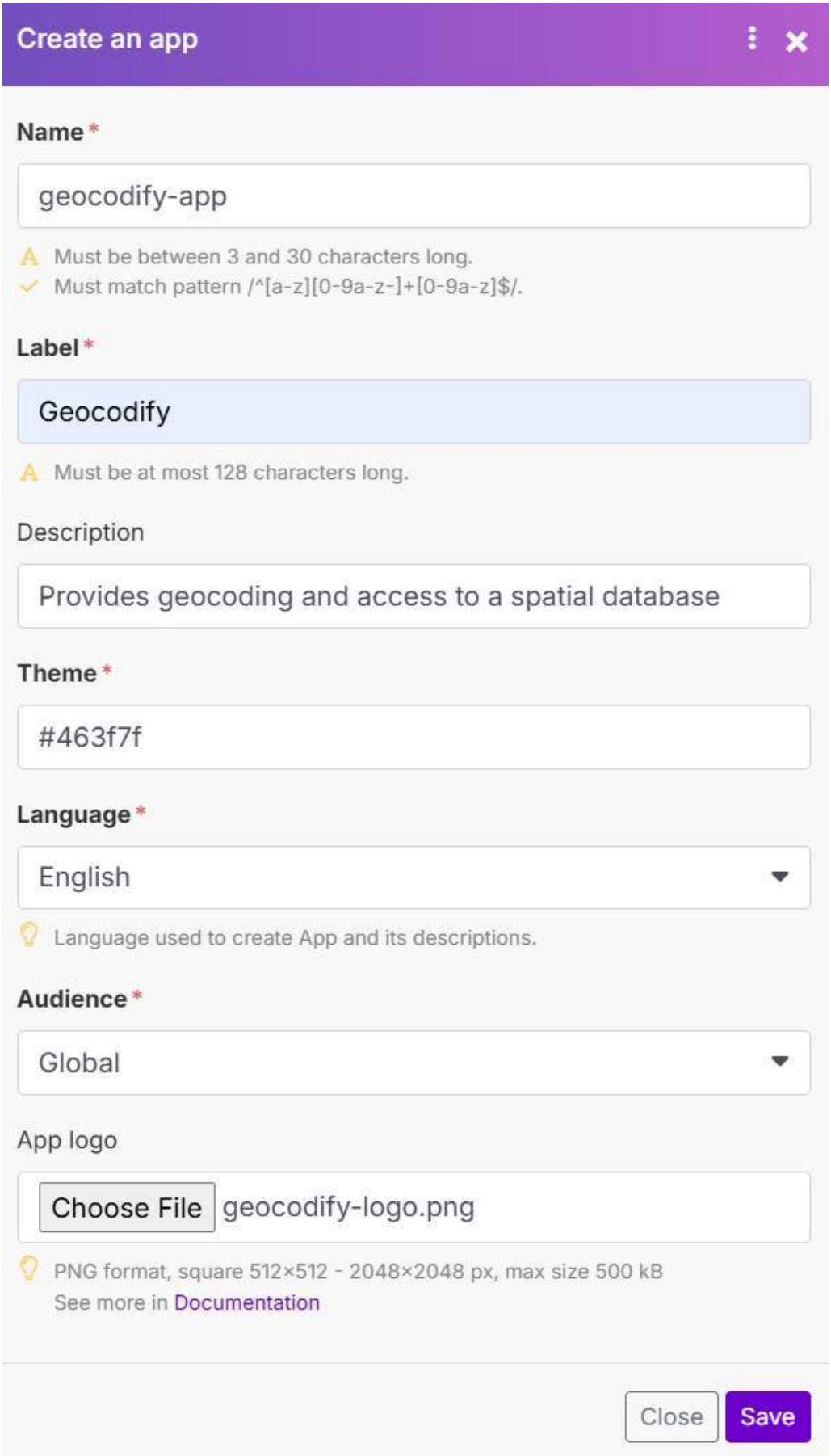
geocodify-logo.png

82.7 KB

CONTINUE

2.5 Build it

If you haven't done it yet, select **+ Create a new app** in the **Custom Apps** menu.


Create an app
Name *
geocodify-app
Must be between 3 and 30 characters long.
Must match pattern /^[a-z][0-9a-z-]+[0-9a-z]$/.
Label *
Geocodify
Must be at most 128 characters long.
Description
Provides geocoding and access to a spatial database
Theme *
#463f7f
Language *
English
Language used to create App and its descriptions.
Audience *
Global
App logo
Choose File
geocodify-logo.png
PNG format, square 512×512 - 2048×2048 px, max size 500 kB
See more in Documentation
Close
Save

Complete the form with the relevant information as shown below:

- **Name**: geocodify-app
- **Label**: Geocodify
- **Description**: Provides geocoding and access to a spatial database
- **Theme**: #463f7f
- **Language**: English
- **Audience**: Global
- **App logo**: Upload the image you downloaded before

Note that you can find the theme color by inspecting the **Geocodify website** and selecting the hex code for the purple background on the landing page.

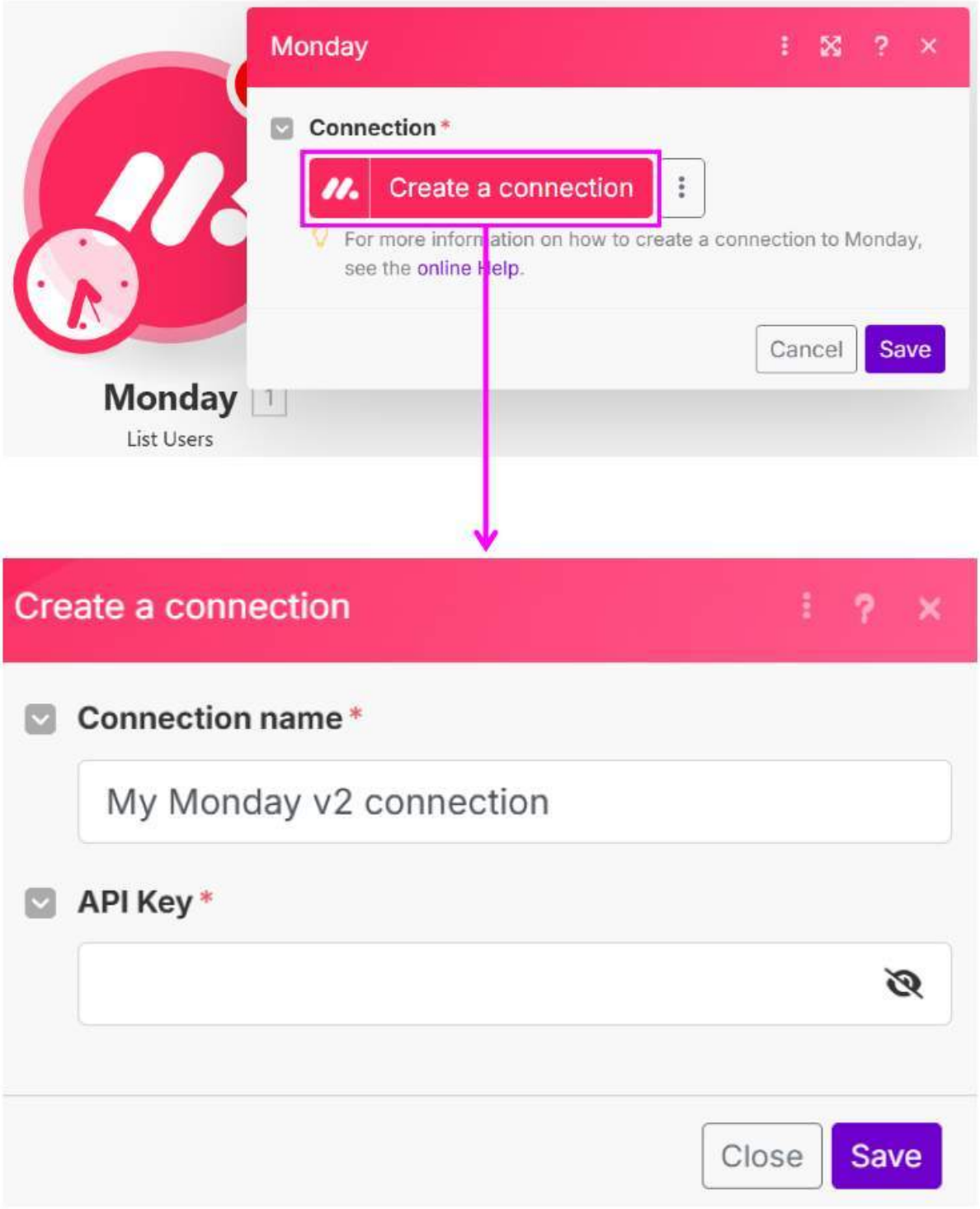
Click **Save** to create the app.

CONTINUE

2.6 Connections

2.6.1 Learn it

To begin setting up the components, start with **Connections**, as the other components will use the **Connections'** parameters.



When building the **Connections** component, you will set up the form that appears when the user clicks **Create a connection**.

You will define the parameters the user has to input and how these are handled by the apps engine to verify the authentication parameters.

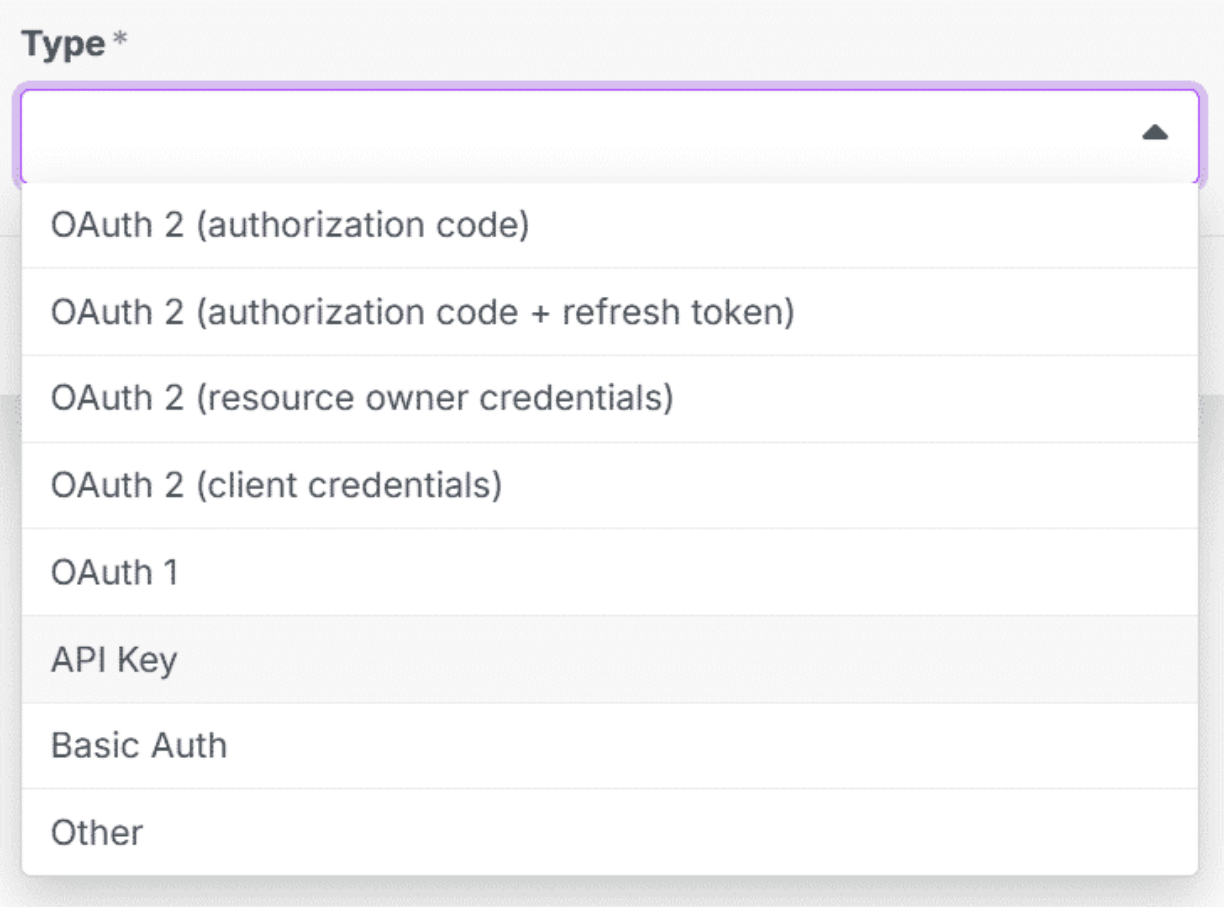
The **Connections** component sets up the details needed to authenticate with the API and ensures everything works when the user enters their credentials in the scenario editor.

It has three main jobs:

1. Getting any relevant information from the user that adds the connection (credentials, API key, etc)
2. Processing the authentication
3. Checking if the authentication works by making a test API call

You set this up in the **Communication** tab, while the **Parameters** tab holds the information the user needs to provide.

Let's have a look at the **Connections** component before setting it up. You don't have to do anything at this point, but you can create a new connection if you want to follow along.



When you click **Create a new connection**, Make presents you with a list of the most common connection types with already filled in code that you can modify according to your needs.

> For this exercise you will select **API Key**, but before doing so let's explore the pre-filled in code.
>
> **Work through the images below to learn more.**

```jsonc
{
  // Request
  "url": "https://www.example.com/api/whoami",          // Absolute URL to the API endpoint which validates credentials
  "headers": {                                          // Additional HTTP headers
      "Authorization": "Bearer {{parameters.apiKey}}"   // Authorizes user by API key, provided by user during the connection creation.
  },

  // Response handling
  "response": {
      "metadata": {                                     // Adds authorized user details to the connection label.
          "type": "email",                              // Type of the parameter. Can be "text" or "email".
          "value": "{{body.email}}"                     // The value in "email" will be displayed in connection's label.
      },
      "error": {                                        // Error handling
          "message": "[{{statusCode}}] {{body.error}}"  // On error, returns error message as "[statusCode] error text".
      }
  },

  "log": {
      "sanitize": [                                     // Excludes sensitive parameters from logs.
          "request.headers.authorization"               // Omit HTTP header "Authorization".
      ]
  }
}
```

request

response

log

The code in the **Communication** tab is divided into three sections:

1. **Request**
2. **Response**
3. **Log**

```
{
    // Request
    "url": "https://www.example.com/api/whoami",          // Absolute URL to the API endpoint which validates credentials
    "headers": {                                          // Additional HTTP headers
        "Authorization": "Bearer {{parameters.apiKey}}"   // Authorizes user by API key, provided by user during the connection creation.
    },
```

The first part defines the **HTTP request** the app uses to validate the user's credentials.
This call happens when the user enters their credentials and clicks **Create a Connection**.

The usual way to validate credentials is to call an API endpoint that returns user details. If the API doesn't provide this, you can use a generic endpoint to validate the credentials by making a GET request to ensure that the authentication is correct. The default HTTP method is **GET**, so you don’t need to specify it.

Adding this validation call is a good practice to make sure the credentials are correct.

```
// Response handling
"response": {
    "metadata": {                                   // Adds authorized user details to the connection label.
        "type": "email",                            // Type of the parameter. Can be "text" or "email".
        "value": "{{body.email}}"                   // The value in "email" will be displayed in connection's label.
    },
    "error": {                                      // Error handling
        "message": "[{{statusCode}}] {{body.error}}"  // On error, returns error message as "[statusCode] error text".
    }
},
```

The second part contains directive on how to handle the **response**:

- **metadata**: Stores the user details that are returned by the API call and displays them next to the connection in the **Connections** page, for ease of identification.
- **error**: Contains instructions on the information that is displayed when an error occurs to help the user understand the issue. Lets you set the type of error and the message that will appear if the API request fails.

```
"log": {
    "sanitize": [                                   // Excludes sensitive parameters from logs.
        "request.headers.authorization"             // Omit HTTP header "Authorization".
    ]
```

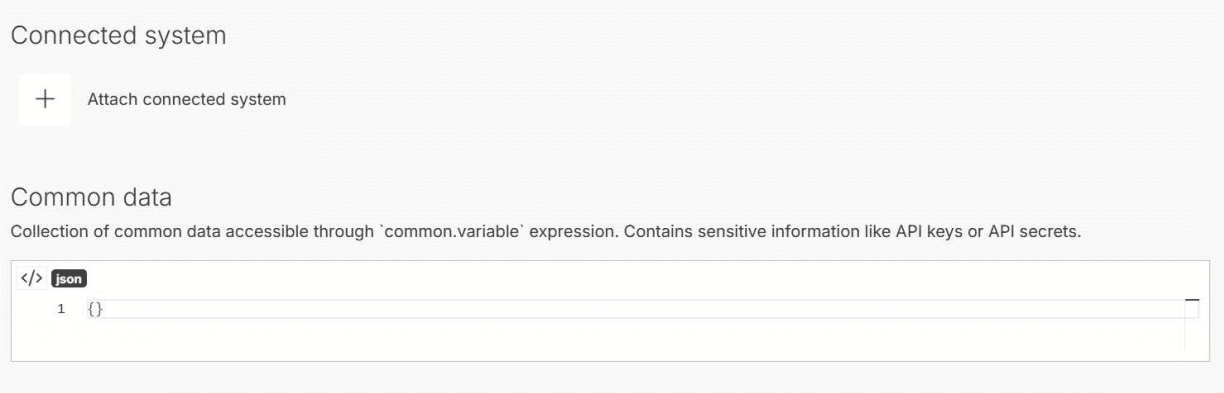
The third part contains instructions regarding the **log** and the information recorded in it:

- **sanitize**: specifies which information shouldn't be recorded in the logs for security reasons.

```jsonc
[
    {
        "name": "apiKey",
        "type": "password",
        "label": "API Key",
        "required": true,
        "editable": true
    }
]
```

The **Parameters** tab contains one parameter: **apiKey**. This is the information the user has to provide in the scenario.

Note that the parameter is used in the **Communication** tab using the notation **{{parameters.name}}**.



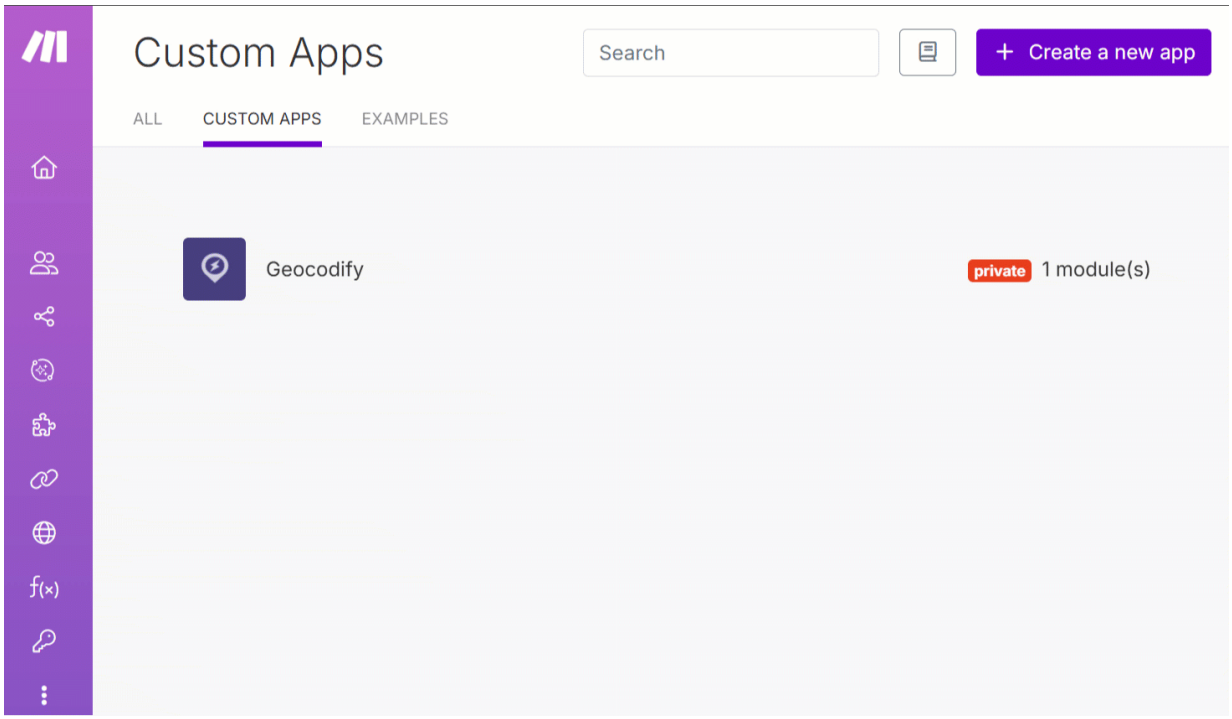
In the **Communication** tab you can also find:

- **Connected system**: To define on-prem agents for Enterprise customers (nothing for you to do here).

- **Common data**: That can be used to store general sensitive data, common to all users. Common data is always stored encrypted in Make, such as Client ID and Secret for apps using OAuth Authorization Code, if you want to allow users to connect easily. Note that the **Common data** is only accessible by the **Connections Communication** tab.

2.6.2 Build it

Now that you know how the **Connections** component works, you can set it up for the Geocodify app. To access the **Connections**, click your **Geocodify** app from the **Custom Apps** menu.



In the **Connections** component, click **Create a new Connection**.

Fill in the details:

- **Label**: Geocodify API Key. (The best practice is to use: App Name + Auth Type).

- **Type**: API Key (Choose this type because the API documentation indicates that you need an API key for the authentication).

Click **Save**.

Select the **Parameters** tab, and then remove the default parameter that is present.

Select, copy, and paste the following code.

```
[
    {
        "name": "apiKey",
        "label": "API Key",
        "type": "password",
        "help": "Enter the API Key provided by Geocodify. For details, see
        "required": true,
        "editable": true
    }
]
```

Save the changes, by clicking the button at the top right.

In the **Communication** tab, remove the code present.

Select, copy, and paste the following code.

```
{
        // Request
        "url": "https://api.geocodify.com/v2/geocode", // Absolute URL to t
        "qs": { // Query parameters
                "api_key": "{{parameters.apiKey}}" // Authorizes user by ap
        },
        "response": {

                "error": { // Error handling
                        "message": "[{body.meta.code}}] {body.meta.error_de

                }
        },
        "log": {
                "sanitize": [ // Excludes sensitive parameters from logs.
                        "request.qs.api_key" ] // Omit query string apy_key
        }
}
```

Save the changes.

Let's look at them in detail, starting from the **Parameters** tab.

jsonc

```jsonc
[
    {
        "name": "apiKey",
        "label": "API Key",
        "type": "password",
        "help": "Enter the API Key provided by Geocodify. For details, see the  [Geocodify account documentation](https:
        "required": true,
        "editable": true
    }
```

You specify one parameter for the API key. This is the only parameter that the user will have to input in the scenario.

- **name**: [Required] Internal name of the parameter. Use it when you want to retrieve the parameter.
- **label**: Parameter name displayed in the module setup.
- **type**: [Required] Data type of the parameter. Click **here** to learn more.
- **help**: Instructions for the user displayed in the module setup. It supports Markdown for text formatting.
- **required**: Specifies if the parameter is required.
- **editable**: (Only for the connection) It specifies whether the user can edit and modify the connection from the Connections page in Make.

Best practices

Help under parameters

Using a help directive, you may **specify a hint of what is expected for the parameter** when it is not that obvious from the label or the expected value is more complicated. The text should start with a capital letter and end with a period.

Editable connection

We recommend **allowing users to edit their connections after they create them**. Updating a connection simplifies scenario and user credential maintenance when there's a change in the user's organization.

You can find the best practices here.

Let's have a look at the updated **Communication** tab.

jsonc

```jsonc
{
    // Request
    "url": "https://api.geocodify.com/v2/geocode", // Absolute URL to the API endpoint which validates credentials
    "qs": { // Query parameters
        "api_key": "{{parameters.apiKey}}" // Authorizes user by api key, provided by user during the connection creat
    },
    "response": {

        "error": { // Error handling
            "message": "[{{statusCode}}] Error code: {{body.meta.code}}\n{{body.meta.error_detail}}" // On error, retu

        }
    },
    "log": {
        "sanitize": [ // Excludes sensitive parameters from logs.
            "request.qs.api_key" ] // Omit query string apy_key
    }
}

```

1/3 Request

Request that the apps engine will make to validate the credentials.

- **url**: Absolute URL of the endpoint that is used for validation.
- **qs**: Query string.
- **api_key**: Key of the **qs** parameter as specified by the API docs. This means that it will use the apiKey that the user provides.

jsonc

```jsonc
{
    // Request
    "url": "https://api.geocodify.com/v2/geocode", // Absolute URL to the API endpoint which validates credentials
    "qs": { // Query parameters
        "api_key": "{{parameters.apiKey}}" // Authorizes user by api key, provided by user during the connection creat
    },
    "response": {

        "error": { // Error handling
            "message": "[{{statusCode}}] Error code: {{body.meta.code}}\n{{body.meta.error_detail}}" // On error, retu

        }
    },
    "log": {
        "sanitize": [ // Excludes sensitive parameters from logs.
            "request.qs.api_key" ] // Omit query string apy_key
    }
}

```

2/3 Response

Instructions on how to display any error: **[error code] error message**.

This information is typically present in the API docs. Since it isn't available in this case, you need to retrieve it manually.

Send a request with incorrect credentials in Postman, then check the response body to identify where the error code and message appear.

If available, it's good practice to include the status code in the error response.

</> jsonc

```jsonc
{
    // Request
    "url": "https://api.geocodify.com/v2/geocode", // Absolute URL to the API endpoint which validates credentials
    "qs": { // Query parameters
        "api_key": "{{parameters.apiKey}}" // Authorizes user by api key, provided by user during the connection creat
    },
    "response": {

        "error": { // Error handling
            "message": "[{{statusCode}}] Error code: {{body.meta.code}}\n{{body.meta.error_detail}}" // On error, retu

        }
    },
    "log": {
        "sanitize": [ // Excludes sensitive parameters from logs.
            "request.qs.api_key" ] // Omit query string apy_key
    }
}

```

3/3 Log

Indicates to omit the **api_key** parameter present in the query string of the request from the log.

Best practices

Connections

Every connection should have a way how to check if the used API Key/Token is valid (Validation Endpoint): the info block in your **Connection** component. That means **each connection should have a part that uses the used API Key/Token against an endpoint that requires only the API Key/Token to run**.

Error handling

The error handling part should contain the **HTTP status code** and **the error type**.

The error handling code should correspond to the structure of the server response. The error object in our example contains the code and message fields. It is also important to show the status code of the error, this can be accessed using the statusCode keyword.

Sanitization

You should **always sanitize the log, so no personal tokens and/or keys can leak**.

You can find the best practices here.

CONTINUE

2.7 Base

2.7.1 Learn it

The Base component contains the **common settings of the app that are inherited by all modules**. The Base component doesn’t directly correspond to a visible part of the app, it simply holds **general settings**.

The common settings include:

- **Base URL**: The API base URL used as the main address for all requests to the API.
- **Authorization**: Authentication information and credentials.
- **Error Handling and Sanitization**: Same as the Connections component.

If a module doesn’t define a setting, it follows what’s in the Base. However, if a module specifies a different value for the same setting, for example a different error handling directive, this overrides what's present in the Base.

The **Base** component doesn't contain any tabs, but just the code where you specify the common directives and the common data if needed. Let's have a look at the default code.

```jsonc
{
    // Default request configuration
    "baseUrl": "https://www.example.com/api/v2",          // Default base URL for all modules and RPCs.
    "headers": {                                          // Default HTTP headers for all modules and RPCs.
        "Authorization": "Bearer {{connection.apiKey}}"   // Authorization by API key, which user will provide in the connection as parameter
    },

    // Default response handling
    "response": {
        "error": {                                        // Error handling
            "message": "[{{statusCode}}] {{body.error}}"  // On error, returns error message as "[statusCode] error text".
        }
    },

    "log": {
        "sanitize": [                                     // Excludes sensitive parameters from logs.
            "request.headers.authorization"               // Omit HTTP header "Authorization".
        ]
    }
```

The structure of the JSONC code of the **Base** component is very similar to the **Communication** tab that you have set up in the **Connections** component.

Note that in this case you want to use the authentication parameters from the **Connections** component, which is why you set that up first. To do this, use the component name followed by the parameter's name: **connection.apiKey**.

2.7.2 Build it

To build the **Base** component, remove the code that is present.

Then **select, copy, and paste the following code**.

```
{
        // Default request configuration
        "baseUrl": "https://api.geocodify.com/v2", // Default base URL for
        "qs": { // Default query parameters for all modules.
                "api_key": "{{connection.apiKey}}" // API key, which user
```

```
        },

        // Default response handling
        "response": {
                "error": { // Error handling
                        "message": "[{{body.meta.code}}] {{body.meta.error_
                }
        },

        "log": {
                "sanitize": [ // Excludes sensitive parameters from logs.
                        "request.qs.api_key" // remove api_key query param
                ]
        }
}
```

Save the changes.

Let's look at it in detail.

jsonc

```jsonc
{
    // Default request configuration
    "baseUrl": "https://api.geocodify.com/v2", // Default base URL for all modules and RPCs.
    "qs": { // Default query parameters for all modules.
        "api_key": "{{connection.apiKey}}" // API key, which user will provide in the connection as parameter.
    },

    // Default response handling
    "response": {
        "error": { // Error handling
            "message": "[{{body.meta.code}}] {{body.meta.error_detail}}" // On error, returns error detail
        }
    },

    "log": {
        "sanitize": [ // Excludes sensitive parameters from logs.
            "request.qs.api_key" // remove api_key query param from logs.
        ]
    }
```

</> jsonc

```jsonc
{
    // Default request configuration
    "baseUrl": "https://api.geocodify.com/v2", // Default base URL for all modules and RPCs.
    "qs": { // Default query parameters for all modules.
        "api_key": "{{connection.apiKey}}" // API key, which user will provide in the connection as parameter.
    },

    // Default response handling
    "response": {
        "error": { // Error handling
            "message": "[{{body.meta.code}}] {{body.meta.error_detail}}" // On error, returns error detail
        }
    },

    "log": {
        "sanitize": [ // Excludes sensitive parameters from logs.
            "request.qs.api_key" // remove api_key query param from logs.
        ]
    }
```

1/4 baseURL

Contains the **base URL** of the API. Note that it contains the API version.

You will specify the different endpoints in the modules.

jsonc

```jsonc
{
    // Default request configuration
    "baseUrl": "https://api.geocodify.com/v2", // Default base URL for all modules and RPCs.
    "qs": { // Default query parameters for all modules.
        "api_key": "{{connection.apiKey}}" // API key, which user will provide in the connection as parameter.
    },

    // Default response handling
    "response": {
        "error": { // Error handling
            "message": "[{{body.meta.code}}] {{body.meta.error_detail}}" // On error, returns error detail
        }
    },

    "log": {
        "sanitize": [ // Excludes sensitive parameters from logs.
            "request.qs.api_key" // remove api_key query param from logs.
        ]
    }
```

2/4 qs

Contains the **query parameter** for authentication.

Notice that the **apiKey** parameter is taken from the **Connections** component and accessed using **connection.apiKey**

jsonc

```jsonc
{
    // Default request configuration
    "baseUrl": "https://api.geocodify.com/v2", // Default base URL for all modules and RPCs.
    "qs": { // Default query parameters for all modules.
        "api_key": "{{connection.apiKey}}" // API key, which user will provide in the connection as parameter.
    },

    // Default response handling
    "response": {
        "error": { // Error handling
            "message": "[{{body.meta.code}}] {{body.meta.error_detail}}" // On error, returns error detail
        }
    },

    "log": {
        "sanitize": [ // Excludes sensitive parameters from logs.
            "request.qs.api_key" // remove api_key query param from logs.
        ]
    }
```

3/4 response

Instructions on how to handle errors.

Notice that in this case the code is the same as in the **Connections** component.

jsonc

```jsonc
{
    // Default request configuration
    "baseUrl": "https://api.geocodify.com/v2", // Default base URL for all modules and RPCs.
    "qs": { // Default query parameters for all modules.
        "api_key": "{{connection.apiKey}}" // API key, which user will provide in the connection as parameter.
    },

    // Default response handling
    "response": {
        "error": { // Error handling
            "message": "[{{body.meta.code}}] {{body.meta.error_detail}}" // On error, returns error detail
        }
    },

    "log": {
        "sanitize": + Excludes sensitive parameters from logs.
            "request.qs.api_key" // remove api_key query param from logs.
        ]
    }
```

4/4 log

Instructions on the information that is saved in the logs.

sanitize specifies what needs to be omitted.

Notice that in this case the code is the same as in the **Connections** component.

Best practices

Base

The Base section should contain **data that is common to all (or most) requests**. At the very least, this should include the **root URL**, the **authorization headers**, the **error-handling section**, and the **sanitization**.

Base URL

Make sure that the Base URL **uses the URL of the API**, which is shared among all modules or their majority.

Authorization and sanitization

The Base section should also have authorization, which is common for all modules. This **authorization should use the API Key**, **Access Token**, **or Username and Password entered in the connection**. **The sanitization should hide all these sensitive parameters.**

Error handling

Each service sends an error message when an error occurs. Most of the time this happens when the request has wrong parameters or values or when the service has an outage. That's why error handling is required. The **error handling code should correspond to the structure of the server response**.

You can find the best practices here.

CONTINUE

2.8 Modules

In Make, you can set up 6 different types of modules, each with a specific function.

- **Action**: Triggers **a single response from the API**, such as adding a new location, deleting a location, or retrieving information about a place.
- **Search**: Returns **multiple results from the API**, like searching for places or locations based on specific criteria. If the API returns an array of items, you can use the iterate directive to loop through the array and generate a bundle for each item.
- **Trigger (polling)**: **Monitors changes in the application or service**, such as triggering an action when a new location is added or updated.
- **Instant trigger (webhook)**: **Listens to incoming requests from the third-party application when an event happens**, such as reacting to a location update or a new event related to a place.
- **Universal**: Allows making **any arbitrary API calls to the service**, such as querying location data or executing a custom geocoding request. It is the Make an API call module that you can see in most Make apps.
- **Responder**: Sends processed data back to a webhook, such as the location information.

In this course you will set up a **Search module** to **retrieve the coordinates of an address**.

CONTINUE

2.9 Search module

2.9.1 Learn it

The **Search module sends a request and returns multiple results**. Use this module when you want to let users search for records.

Inside the **Search module** component there are 5 tabs:

- **Communication**: Information on **what the engine needs to do to manage the API call** (call the endpoint, process the response, pagination, etc). Remember that the following elements are **inherited from the Base**: **base URL**, **error handling**, **log sanitize**. If something needs to be changed, you can write it here, and it will override the settings of the Base component.
- **Static parameters**: Input **parameters that the user cannot map from other modules**. They are only used for polling triggers, which don't have Mappable Parameters.
- **Mappable parameters**: Input parameters in the interface that the user can either **enter manually or map from the output of other modules**.
- **Interface**: **Labels of the module's output** added to **make the output more straightforward and easy to interpret**. By specifying it, there's no need to first run

the module in your scenario to get the output structure for mapping the elements.

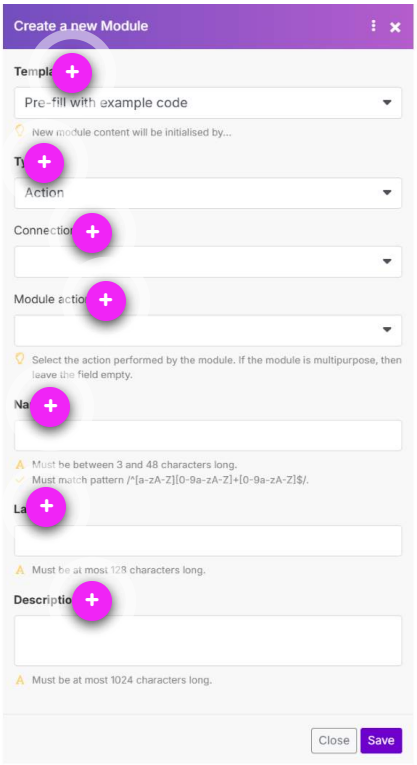
- **Samples**: Examples of data to help the users set up the module.

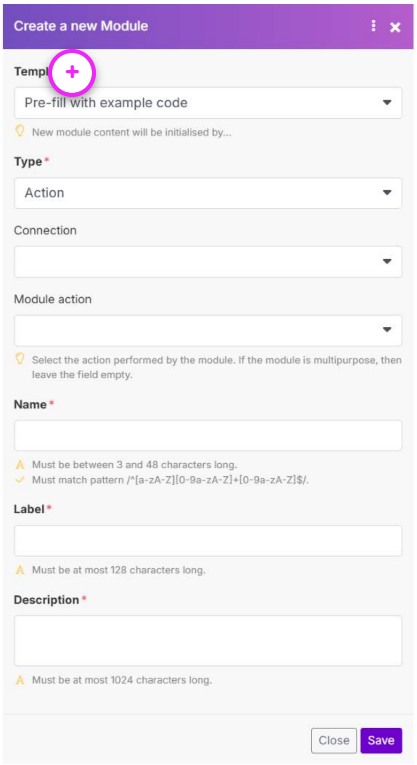
You're going to set up the **Search module** from scratch, so you won't be exploring any default code for this component.

2.9.2 Build it

The **Search module** sends one or more requests and returns multiple results. Use this module when you want to let users search for items or simply return multiple items.

From the **Custom Apps** menu> **Geocodify**, go to the **Modules** component and select **+ Create a new module**. In the pop-up window you will need to fill in the module details. Let's start by having a look at the different items before filling them in.

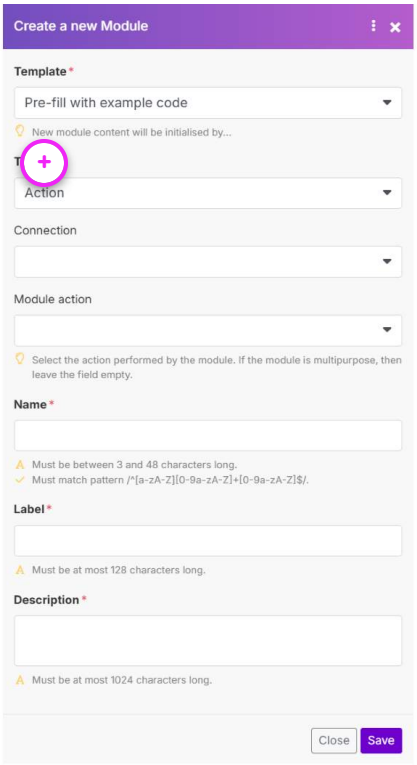

Create a new Module
Pre-fill with example code
New module content will be initialised by...
Action
Select the action performed by the module. If the module is multipurpose, then leave the field empty.
Must be between 3 and 48 characters long.
Must match pattern /^[a-zA-Z][0-9a-zA-Z]+[0-9a-zA-Z]$/.
Must be at most 128 characters long.
Must be at most 1024 characters long.
Close
Save



1/7 Template

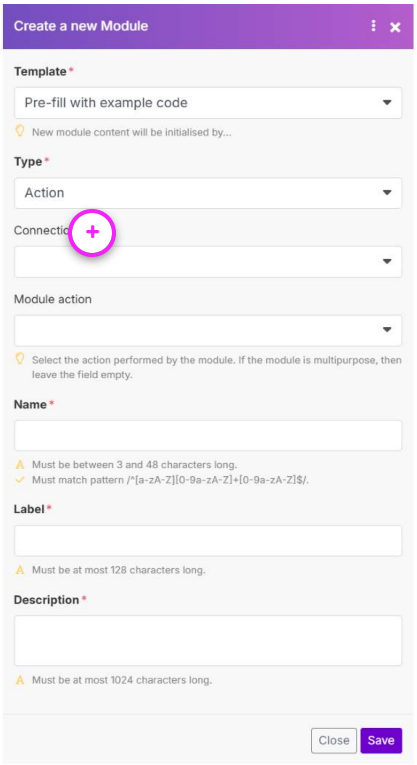
Select how you want the module to be pre-filled with code. The options are:

- **Blank module**
- **Pre-fill with example code** [selected by default]
- **Copy code from existing module**



2/7 Type

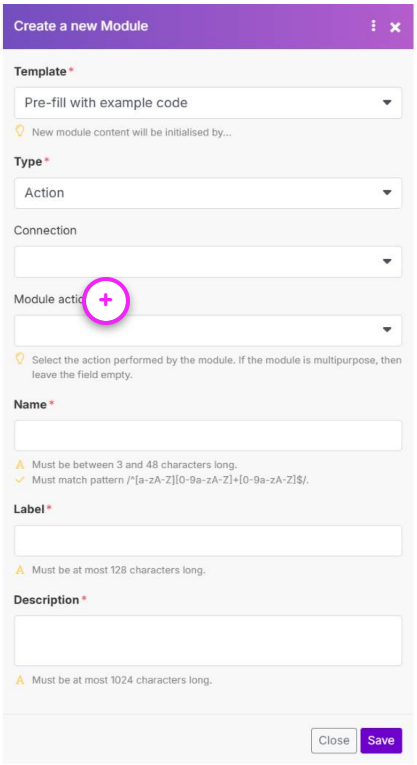
Choose one of the six module types described above.



3/7 Connection

[Optional]

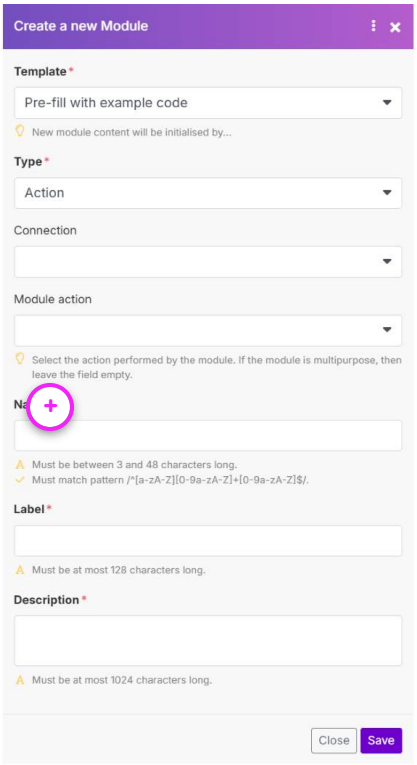
Link a connection to authenticate the API calls.



4/7 Module action

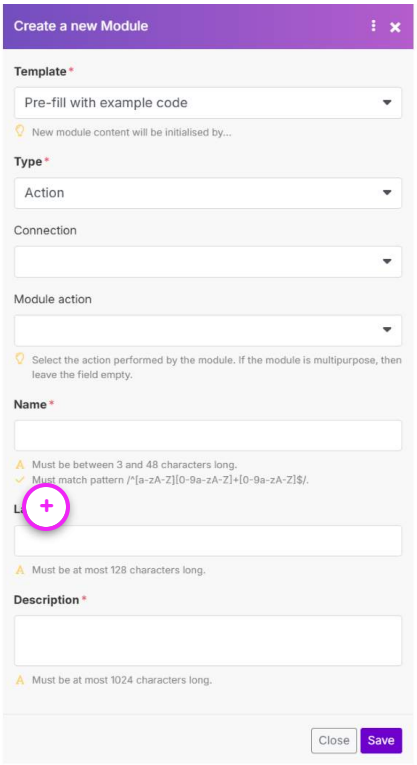
[Optional]

Only for Action modules.



5/7 Name

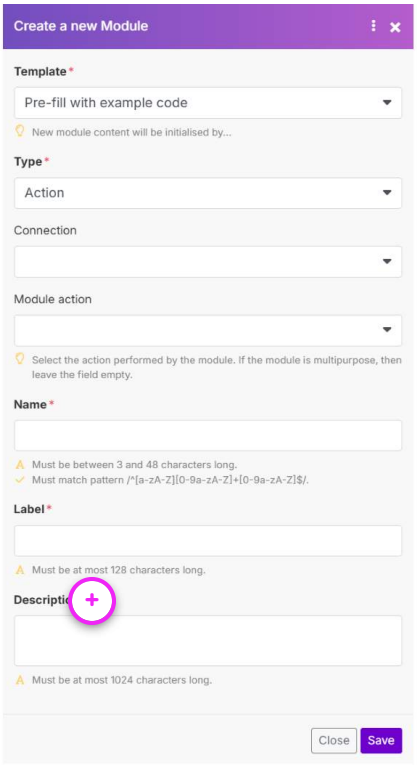
Module internal name. See instruction below the box for character restrictions.



6/7 Label

Name of the module as shown in the scenario.

See **Best Practices** below for naming recommendations.



7/7 Description

Description of the module.

Fill in the module setup as below:

Create a new Module

Template *

Blank Module

New module content will be initialised by...

Type *

Search

Connection

Geocodify

Name *

geocode

Must be between 3 and 48 characters long.
Must match pattern /^[a-zA-Z][0-9a-zA-Z]+[0-9a-zA-Z]$/.

Label *

Search Geolocation

Must be at most 128 characters long.

Description *

Provides longitude, latitude, and place details of a location (address, name of a place, or location).

Must be at most 1024 characters long.

Close Save

Template: Blank module (you will set it up from scratch)

Type: Search (to retrieve geolocalization details)

Connection: Geocodify (the connection you have created earlier)

Name: geocode

Label: Search Geolocation

Description: Provides longitude, latitude, and place details of a location (address, name of a place, or location).

Click **Save**.

Best practices

Type

Modules should be associated with the **correct type**, depending on their functionality.

Name

A name of a module [...] **should not match with any reserved word in JavaScript**.

Label

Every module should have a label that precisely describes the module's use. For each type of module, there is a **standard naming convention**. But it may change depending on the functionality of the module. The label should be composed of the **verb expressing the intended action** (Create, Update, Watch, etc.) and the **name of the entity being processed** (Customer, Invoice, Table, etc).

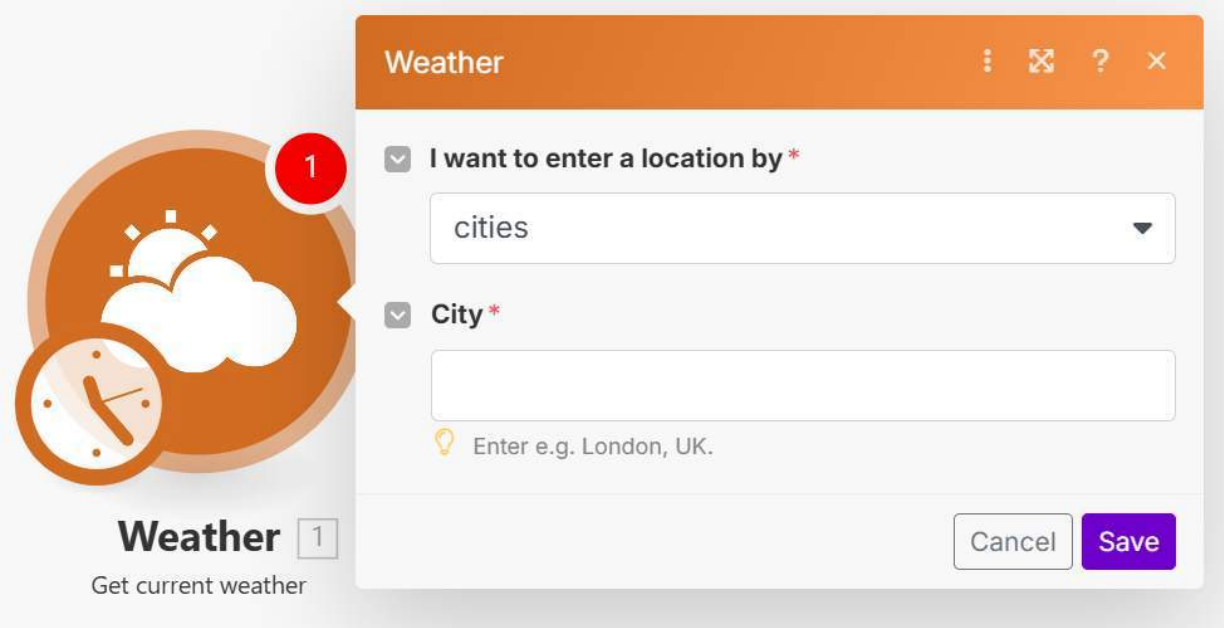
Names of modules follow the **English format** and the **sentence case capitalization**. Basically, only the first letter of the first word in a sentence and proper nouns are capitalized, with all other letters and words in lowercase. Note that you might see **some old apps still using Title Case** (every word is capitalized except for articles, prepositions, and conjunctions), but new apps are using sentence case for module names.

These modules retrieve data from the service and allow retrieving one or more results. Compose the label using simple verbs like **Search** or **List**. Use the naming convention of the service you are implementing.

Description

In a few words, **describe the functionality of the module**. Write the description in the **third person** and capitalize only the first letter of the first word in the description (like a **normal sentence structure**).

You can find the best practices **here**.



To set up the tabs, start with the **Mappable parameters**.

As mentioned earlier, this is the information the user needs to provide, which will appear in the scenario as shown on the image.

In this case, the user must enter a location to geolocate, so you need to add a mappable parameter for the **location**.

Select, copy, and paste the following code in the **Mappable parameters** tab.

```
// Defines "location" as module input parameters.
[
    {
        "name": "location_info", // Makes value accesible via "{{parameters
        "label": "Location", // Sets the user friendly label visible in the
        "type": "text", // Sets the type to text.
        "help": "Type the location you want to geolocate", // Sets the help
        "required": true // Indicates this parameter is mandatory.
```

```
    }
]
```

Save the changes.

Then **select, copy, and paste the following code** in the **Communication** tab.

```
{
        // Request to API endpoint.
        "url": "/geocode", // Endpoint relative to base URL
        "method": "GET",
        "qs": {
                "q": "{{parameters.location_info}}" // Required query parar
        },

        // Response handling
        "response": {
                "output": "{{body}}" // Return the body of the response
        }
}
```

Save the changes.

Let's have a look at them in detail.

Work through the images below to learn more.

```jsonc
// Defines "location" as module input parameters.
[
    {
        "name": "location_info", // Makes value accesible via "{{parameters.location_info}}".
        "label": "Location", // Sets the user friendly label visible in the module
        "type": "text", // Sets the type to text.
        "help": "Type the location you want to geolocate", // Sets the help text with an example under input field in the module.
        "required": true // Indicates this parameter is mandatory.
    }
]
```

Mappable parameters

As you've seen before, you specify one parameter that the user has to input.

- **name**: [Required] Internal name of the parameter. Use it when you want to access the parameter using **{{parameters.name}}**.
- **label**: Parameter name displayed in the module setup.
- **type**: [Required] Type of the parameter.
- **help**: Instructions for the user displayed in the module setup. It supports Markdown for text formatting.
- **required**: Specifies if the parameter is required.

```jsonc
{
    // Request to API endpoint.
    "url": "/geocode", // Endpoint relative to base URL
    "method": "GET",
    "qs": {
        "q": "{{parameters.location_info}}" // Required query parameter. Parameter "location_info" is defined in Mappable parameters.
    },

    // Response handling
    "response": {
        "output": "{{body}}" // Return the body of the response
    }
}
```

Communication

This section contains all the details specific to this module. The **base URL**, **authentication**, **error handling**, and **sanitization** are inherited from the **Base** and do not need to be specified.

- **url**: The API endpoint. Since it starts with **/**, it is joined to the **base URL**. Note that if the URL starts with **https://**, it will override the base URL.
- **method**: The default method is **GET**, so it could have been omitted in this case.
- **qs**: Query parameter containing the location to geolocate. Note that you access it by using **parameters.location_info**.
- **response**: Defines the output returned, which in this case is the whole body of the response.

Leave all the other tabs as they are, including the **Interface**. This means that you won't apply any filtering or customization to the output, but all the information from the API will be returned as is. This is a topic for a more advanced course, stay tuned!

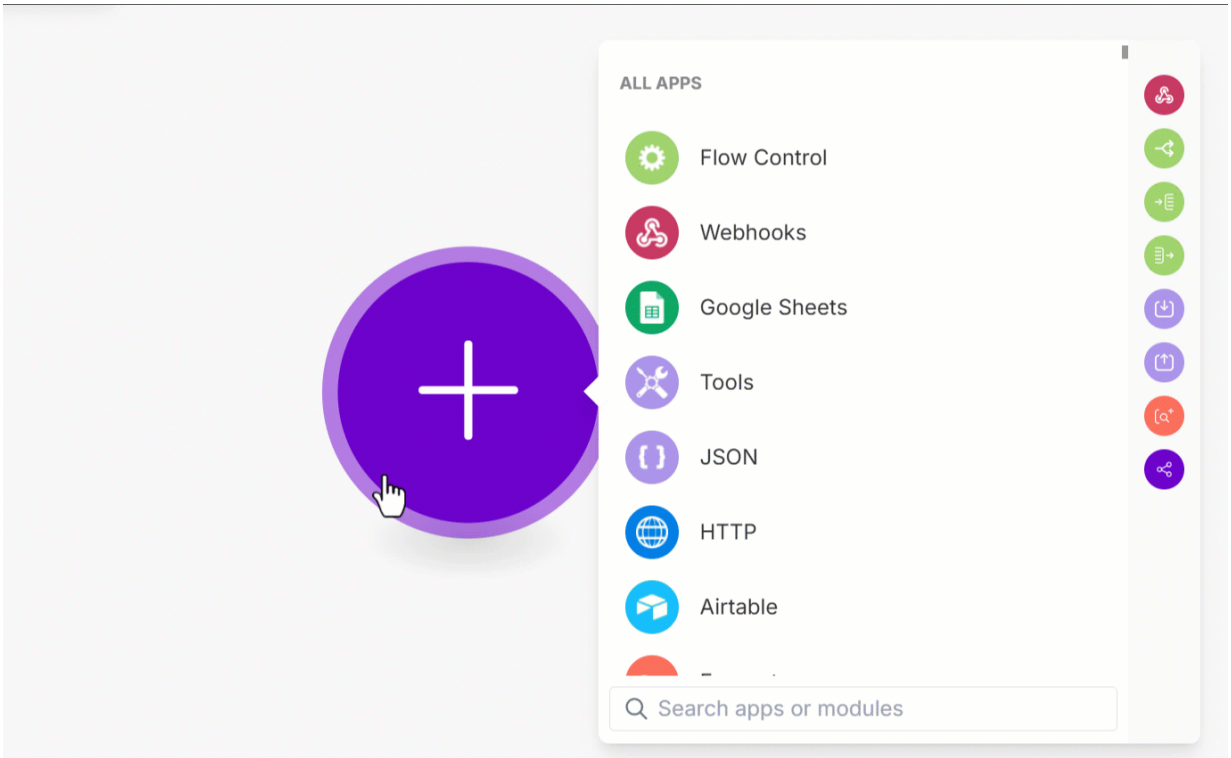
CONTINUE

2.10 Testing your custom app

2.10.1 Test it!

You're all set and it's time to test your app in a scenario.

Work through the images below to learn more.

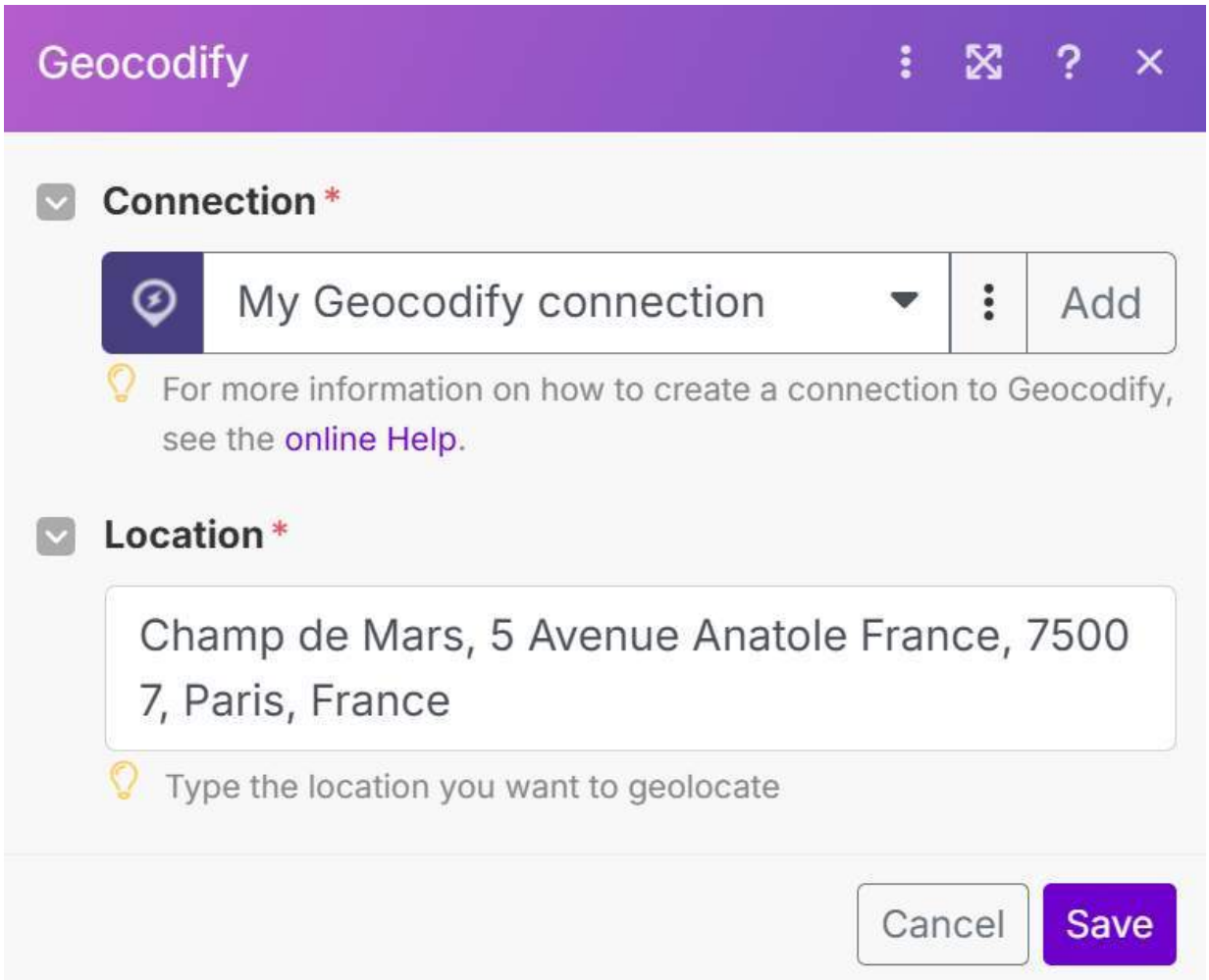


In Make create a new scenario and add the **Geocodify> Search Geolocation** module that you have just created. Notice that it has the **Private** tag.

- Create **Create a connection** to insert your authentication details.

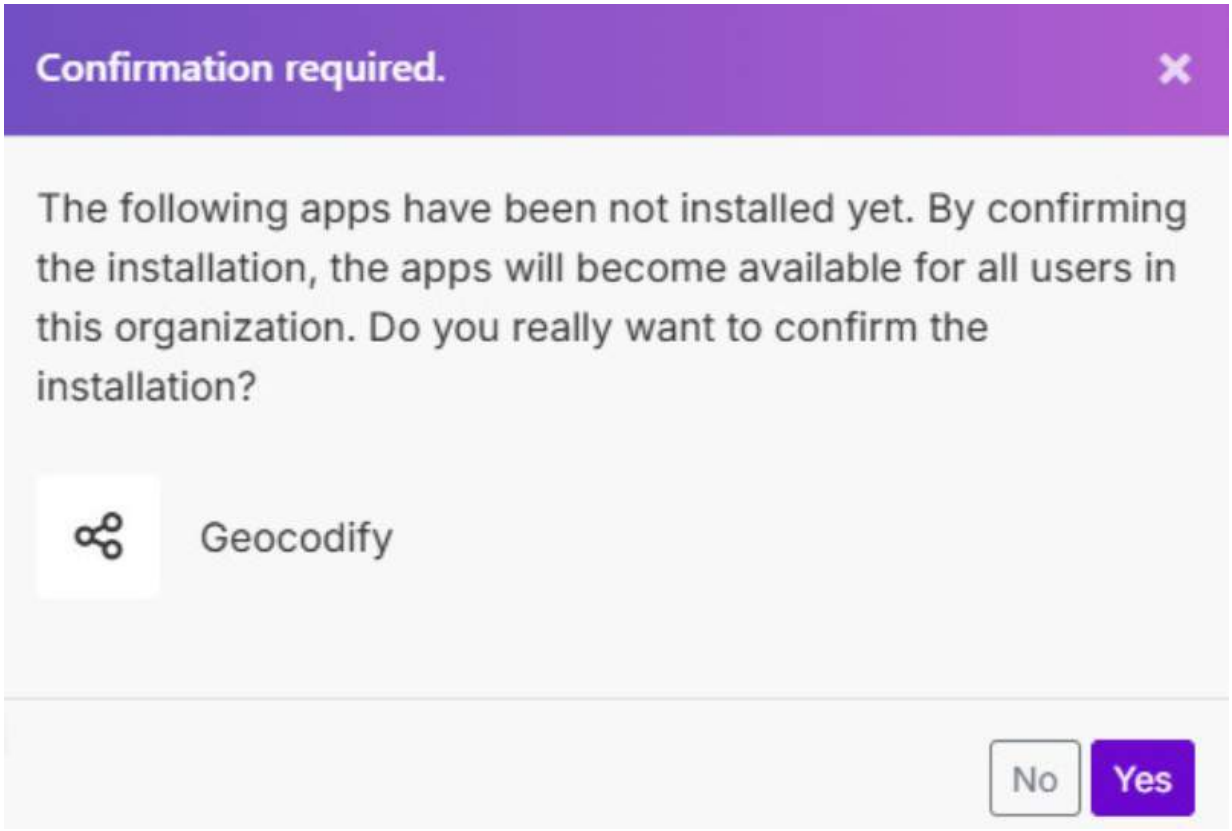
- Paste the **API key** from the **Geocodify API** in the **API Key** field and you can rename the connection if you want.
- Click **Save** to save the connection.

The app engine will make the call to the API specified in the **Connections** element to validate the credentials. If everything is working fine, no error message is displayed.

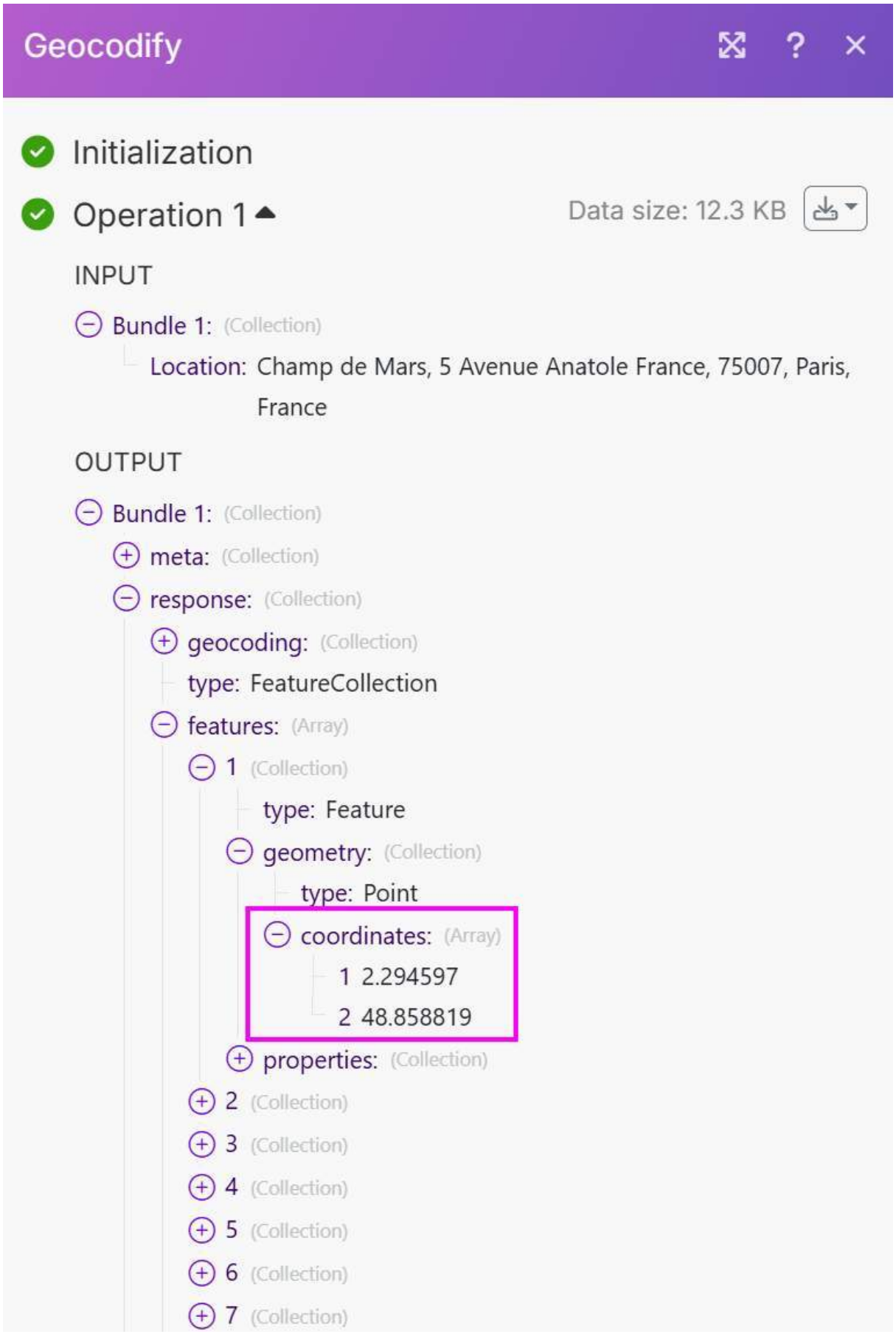


- Insert the address you want to geolocate, for example Champ de Mars, 5 Avenue Anatole France, 75007 Paris, France. This is the **Location** parameter you set up in the **Mappable parameters** tab.

- Click **Save** to save the module.



- **Save the scenario**. A pop-up appears appears asking for confirmation to install the app in your organisation.
- Click **Yes** on the pop-up to install the app in your organisation.
- Click **Run once** to run the scenario.

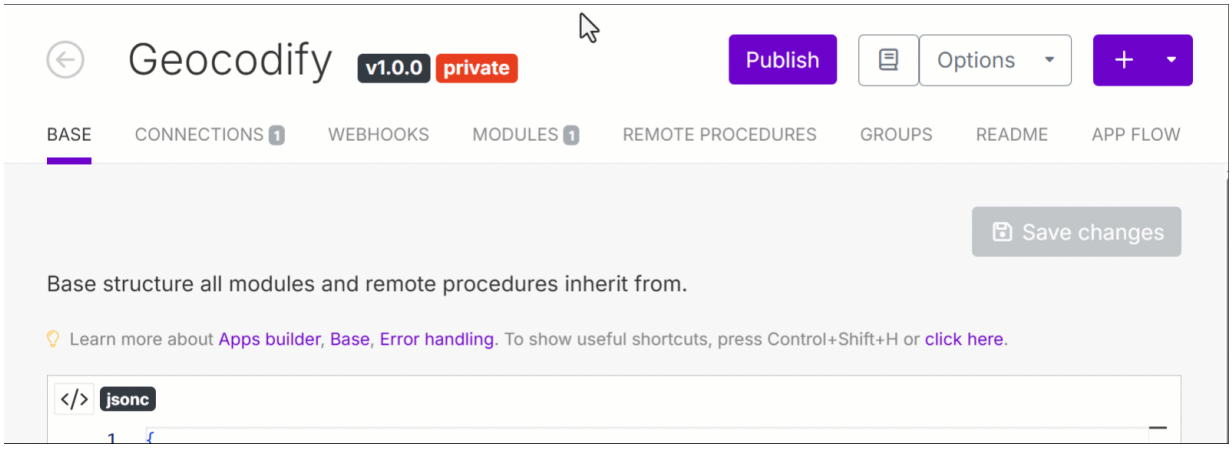

Geocodify
Initialization
Operation 1
Data size: 12.3 KB
INPUT
Bundle 1: (Collection)
Location: Champ de Mars, 5 Avenue Anatole France, 75007, Paris, France
OUTPUT
Bundle 1: (Collection)
meta: (Collection)
response: (Collection)
geocoding: (Collection)
type: FeatureCollection
features: (Array)
1 (Collection)
type: Feature
geometry: (Collection)
type: Point
coordinates: (Array)
1 2.294597
2 48.858819
properties: (Collection)
2 (Collection)
3 (Collection)
4 (Collection)
5 (Collection)
6 (Collection)
7 (Collection)

Open the **Output** to retrieve the coordinates. They are under **Output> Bundle 1> response> features> 1> geometry> coordinates**.

This happens because you chose to return the response as is, without filtering or customization. In future courses, you'll learn how to use the interface to tailor the output and return only the necessary information in a structured format.

2.10.2 Make the module visible

There's one final step to allow members of the organization to use this new module: you need to make it visible to everyone. When you create the module, it's **hidden by default**. Even if the app is installed, the module won't appear in the scenario editor for other users unless you make it visible.



To do so, go to the specific module in the app setup and turn it **ON**. It will now have the **visible** tag.

And that's it! This marks the end of your first introduction to the world of custom apps!

You now understand why a custom app is necessary, its structure and components, and how to use API docs to set up a simple custom app. But there's so much more you can do to build an app that works the way you want. That's a story for another time and other courses.

CONTINUE

Course complete

Congratulations on completing the **Getting started with custom apps** course.

We'd love to hear from you! Rate the course and share your thoughts and impressions with us.

RATE THIS COURSE

When you are done, click **Go to dashboard** to return to the Academy.