

Learning Objectives

- Explain the purpose of Postman
- Install Postman on your computer
- Create a Postman variable that has the base URL for testing
- Perform GET, PUT, POST, and DELETE requests with Postman
- Identify the limitations of this primitive API

Clone Blango Repo

Clone Blango Repo

Before we continue, you need to clone the blango repo so you have all of your code. You will need the SSH information for your repo.

In the Terminal

- Clone the repo. Your command should look something like this:

```
git clone git@github.com:<your_github_username>/blango.git
```

- You should see a blango directory appear in the file tree.

You are now ready for the next assignment.

Postman Part 1

Intro

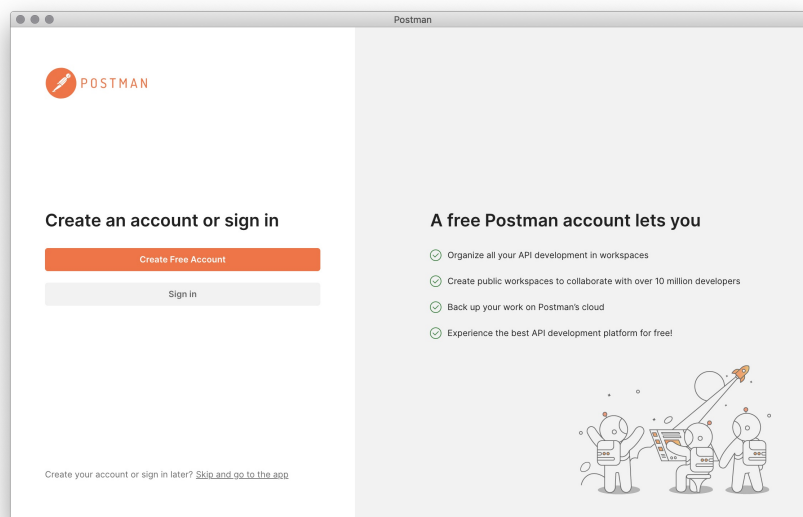
Postman is a cross-platform GUI tool to make HTTP requests. It has many features like support for variables, multiple environments, saved requests, grouped requests, JSON validators, scripting, and more.

We'll walk through how to set up Postman to test our basic API.

Try It Out

Start by downloading Postman; do this on your local computer and not in the Codio environment.

After you've installed it, the first time you launch you'll be prompted to sign in or create an account. While having a Postman account has some benefits, it's not necessary to have one, and you can skip this step by clicking the **Skip and go to the app** link in the bottom left of the page.

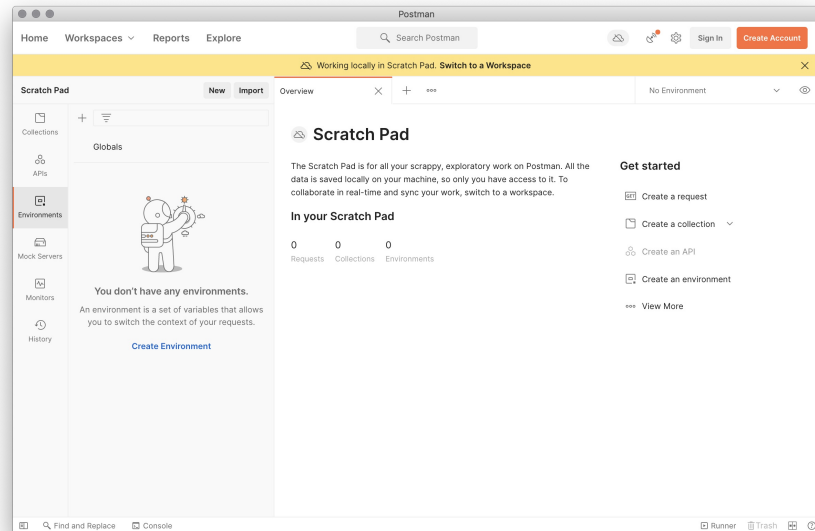


[Click here to see a larger version of the image](#)

Once you're in the app, the first thing we'll do is set up an *Environment*. This is a collection of variables that you can swap between when making requests. For example, in our setup we'll create a `base_url` variable that

will be used to construct URLs. You can then run the same request against a different host by switching environments and using a different `base_url` variable.

Click on **Environments** in the left sidebar, and you'll see a message that says *You don't have any environments*.

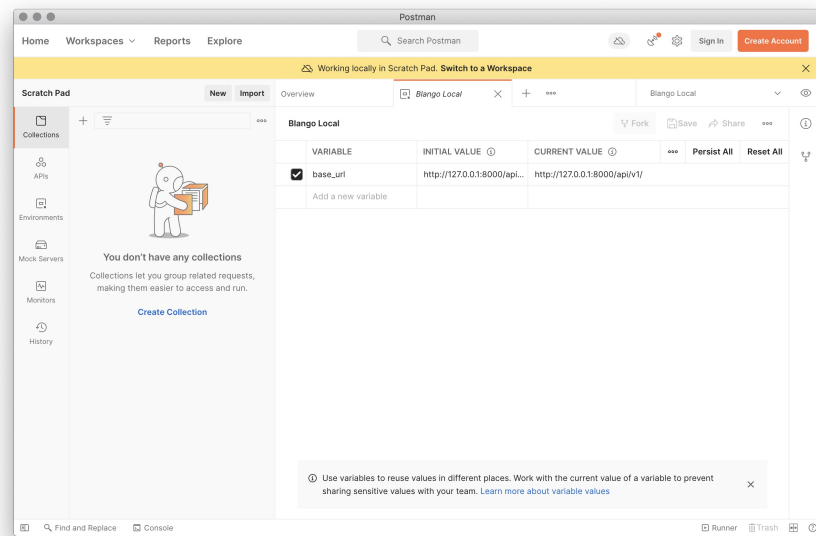


[Click here to see a larger version of the image](#)

Click on **Create Environment**. Enter a name, in the screenshot you'll see we've used *Blango Local*, but *Blango Codio* might make more sense for your case. Then, create a variable called `base_url` with the *INITIAL VALUE* set to your Django V1 API root URL. For example `https://[host]-8000.codio.io/api/v1/` or `http://127.0.0.1:8000/api/v1/`. After entering it you should see the *CURRENT VALUE* is updated to match. Click the *Save* icon to save your environment. You will get errors if it is not saved. Be sure to mark the environment as *active*. Click on the check mark next to the environment name. The check mark should become black.

▼ Finding your Codio host

In the Codio menu bar, click "Project" => "Box Info". This will open a tab with your Codio host name in it.

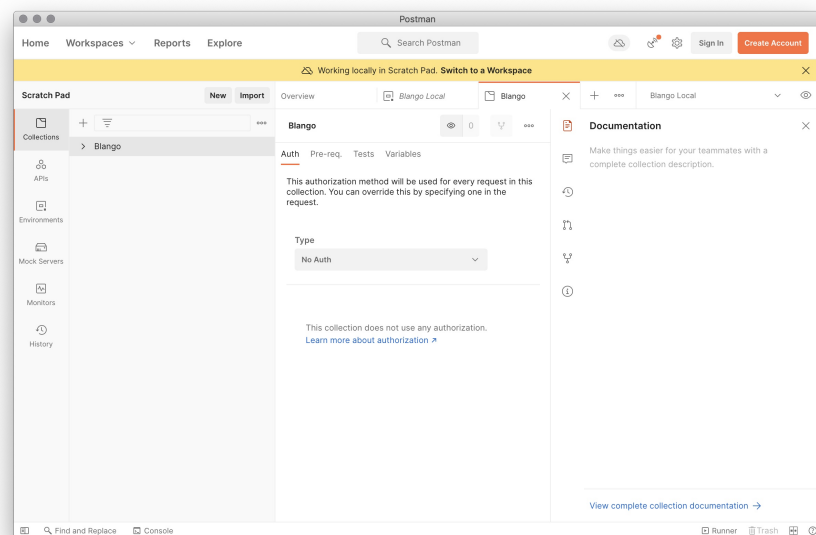


[Click here to see a larger version of the image](#)

The reason that there are two value columns is because a script could change the *CURRENT VALUE* of the variable, so if they differ, that's the one that will be used.

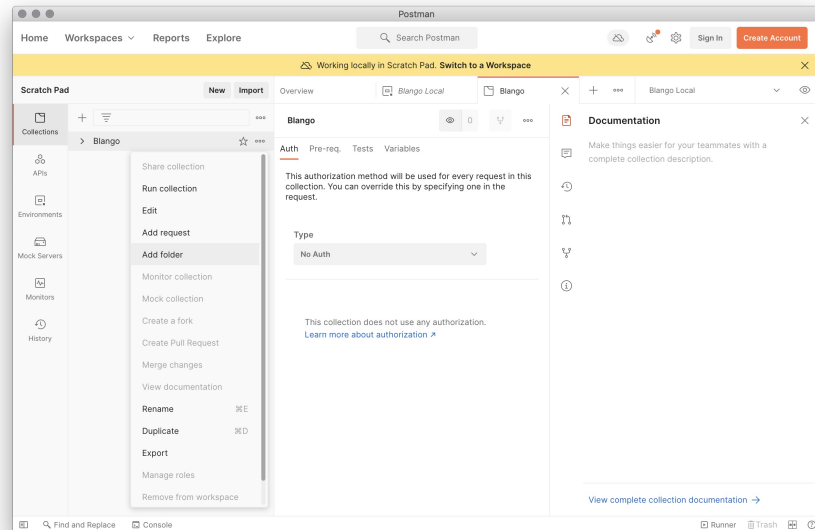
This is the only variable that we'll be using, so we can now create a *Collection*. This is a collection of API requests and is used for organization. You could just create all your API requests in a single collection, if you wanted.

Click on **Collections** in the left sidebar and then click **Create Collection**. Give the new collection the name *Blango*.



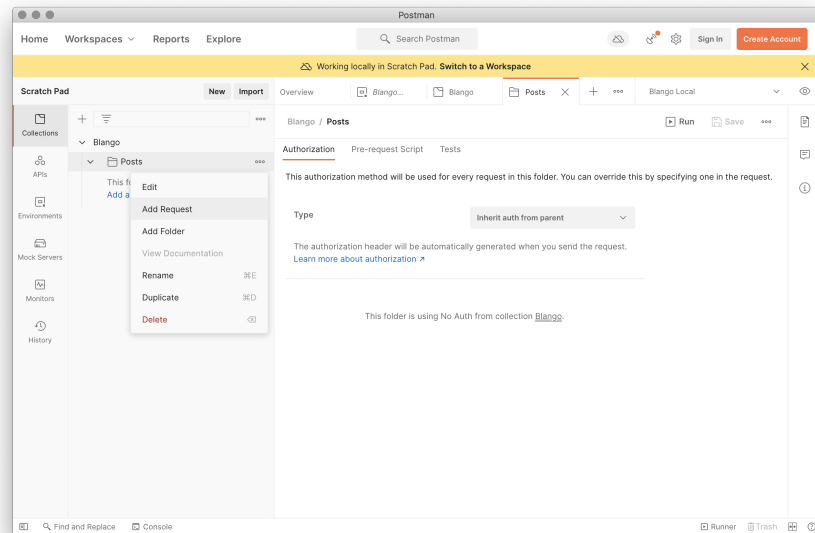
[Click here to see a larger version of the image](#)

After it's created, click the ellipsis menu (the three dots ...) next to the collection name, and choose **Add folder**. Collections can have folders and sub-folders to help organize all your requests. You don't have to create a folder, but it will make it easier to manage. Name your new folder *Posts*, as it will contain all the API requests related to managing Post objects.



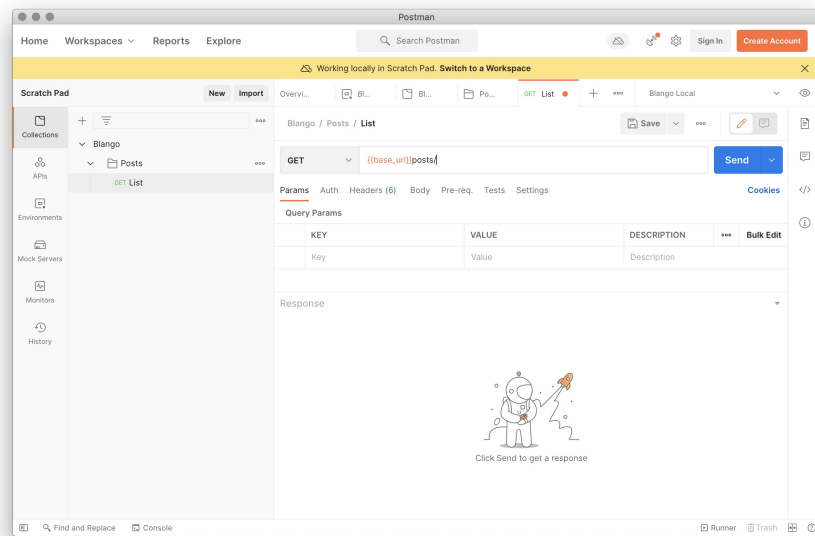
[Click here to see a larger version of the image](#)

Finally we'll add a request. Click on the ... by the folder and choose **Add request**.



[Click here to see a larger version of the image](#)

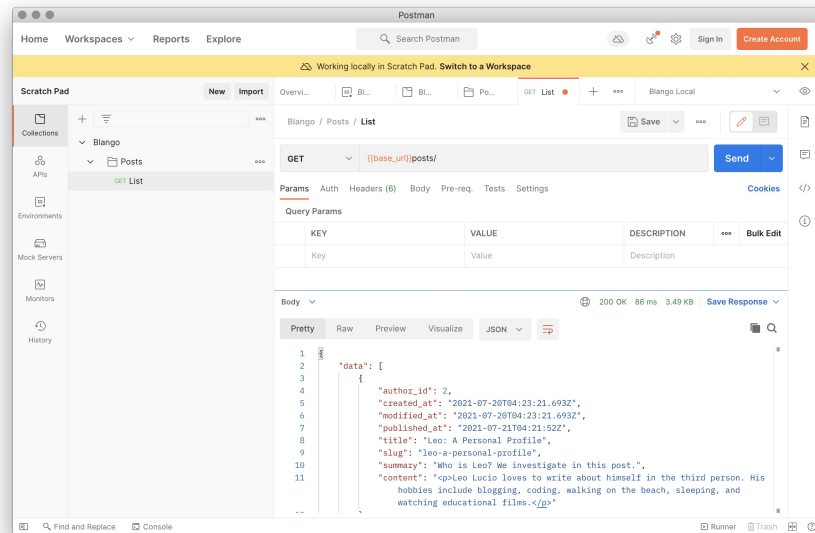
Call the request *List*, since it will be for listing posts. Set the method to GET and for the URL enter. `{{base_url}}posts/`.



[Click here to see a larger version of the image](#)

Like Django, Postman uses the double curly-braces to render a variable. For it to be able to find this variable though, we need to select the environment from which it was added. The environment selector menu is above the **Send** button, and says *No Environment*. Select **Blango Local** or **Blango Codio** from the menu, as appropriate. Be sure to start the dev server.

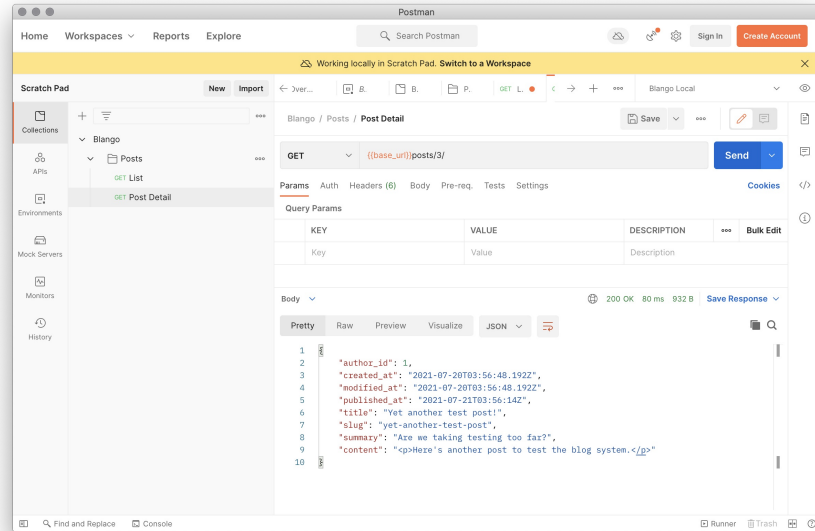
Finally, click the **Send** button. Postman will make the request and show you the response at the bottom of the window.



[Click here to see a larger version of the image](#)

Click the **Save** button, and then we'll create a request to call the detail endpoint. Choose **Add request** again, under the **Posts** folder menu. Call the new request *Post Detail* and set the URL to the detail URL for a post that

you know exists. For example `{{base_url}}posts/3/`. Click on **Send** to try it out and you'll see the detail response down the bottom of the window. Save this request by clicking **Save**.

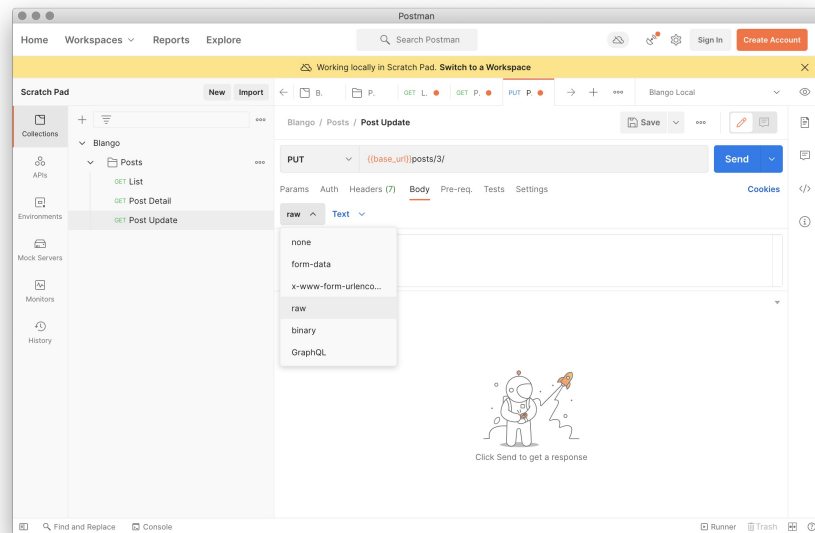


[Click here to see a larger version of the image](#)

Postman Part 2

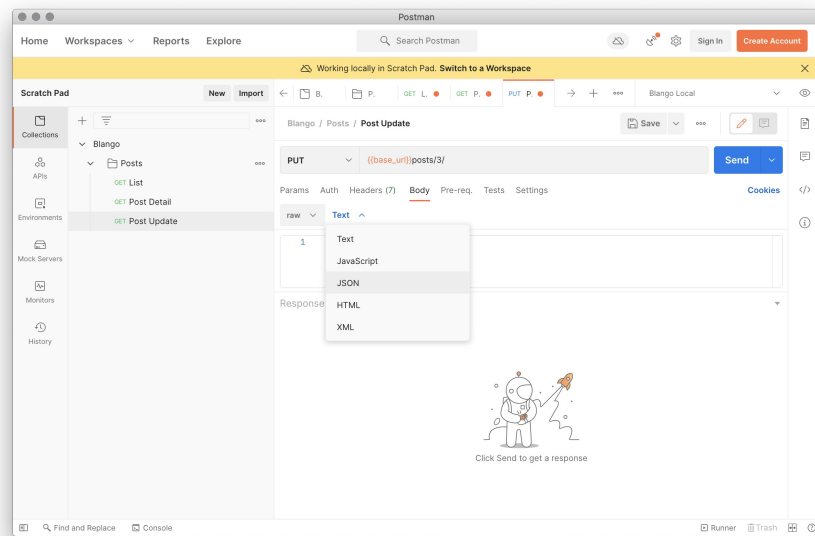
Postman Part 2

Next we'll try a PUT response, to let us update the content of an existing Post. Add a new request, called *Post Update*. The URL can be the same as the *Post Detail* request, for example, `{{base_url}}posts/3/`, but from the dropdown menu select the method **PUT**. Just below this, select the **Body** tab. This is where we can enter the data of the request body to send. Select **raw** from the left dropdown menu.



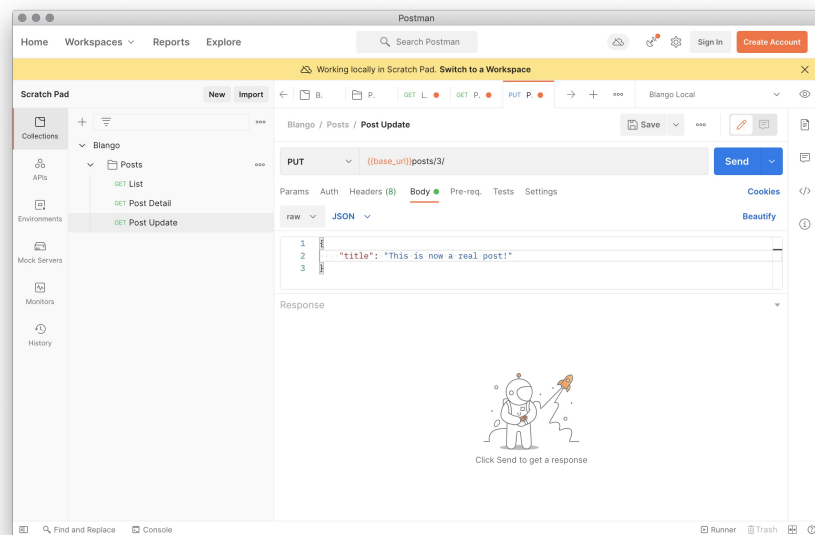
[Click here to see a larger version of the image](#)

Then select **JSON** from the menu to the right of it.



[Click here to see a larger version of the image](#)

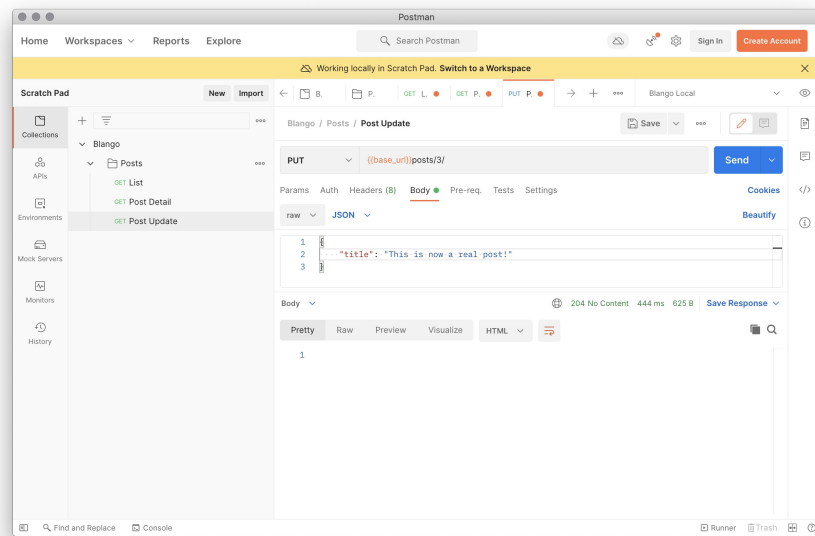
In the text field below you can add some JSON data to update fields of your Post. Due to the way we've written the API, you don't need to enter all the fields, so it's not a strict PUT (which implies replacing all the data with what's sent). Instead it will only overwrite the fields that you send and leave the others. This is just one of the quirks of our simple API which we'll go into at the end of this module (and fix in the next module).



[Click here to see a larger version of the image](#)

Notice that any syntax errors in your JSON will be highlighted. Be sure the dev server is running.

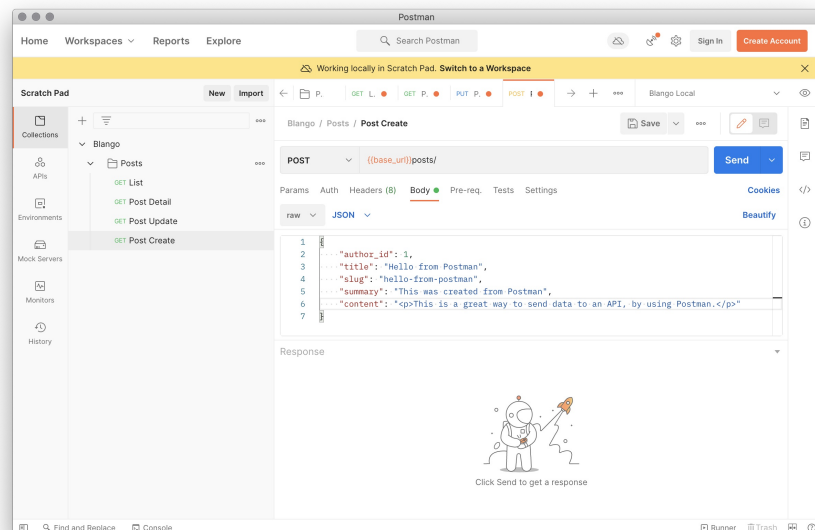
Click **Send** and you'll see an empty response come back, with a 204 status code. This is what we expect.



[Click here to see a larger version of the image](#)

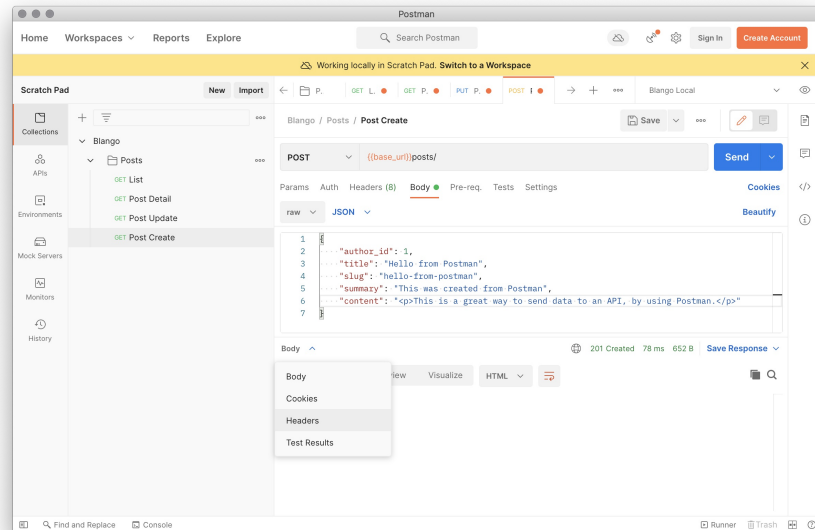
Make sure to save this request. If you try to fetch that Post again, with Postman or a browser, you should see that your change was saved successfully.

Next, Post creation. Add a new request called *Post Create*. Its URL should be the post list URL, i.e. `{{base_url}}posts/`. Set its method to **POST**. Then, as with the *Post Update* request, set the content kind to **raw/JSON**. Enter a JSON representation of the new Post object into the body field. Note that you'll have to provide all the required fields (`author_id`, `title`, `slug`, `summary` and `content`) or the Post won't be saved.



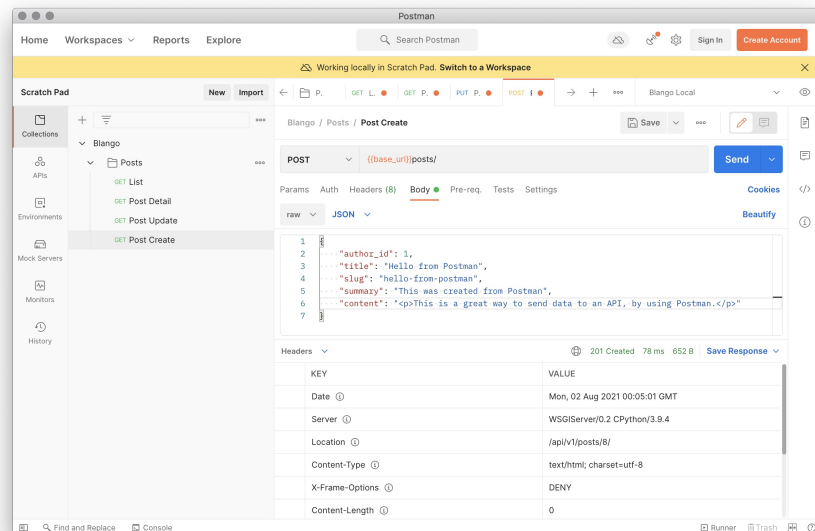
[Click here to see a larger version of the image](#)

Click **Send** to try it out. We'll get back an empty request with 201 status. But how can we see the Location header of where it was created? On the far left of the status code, there's a dropdown menu for selecting which part of the request you want to see. Currently **Body** is selected. Choose **Headers** from this menu.



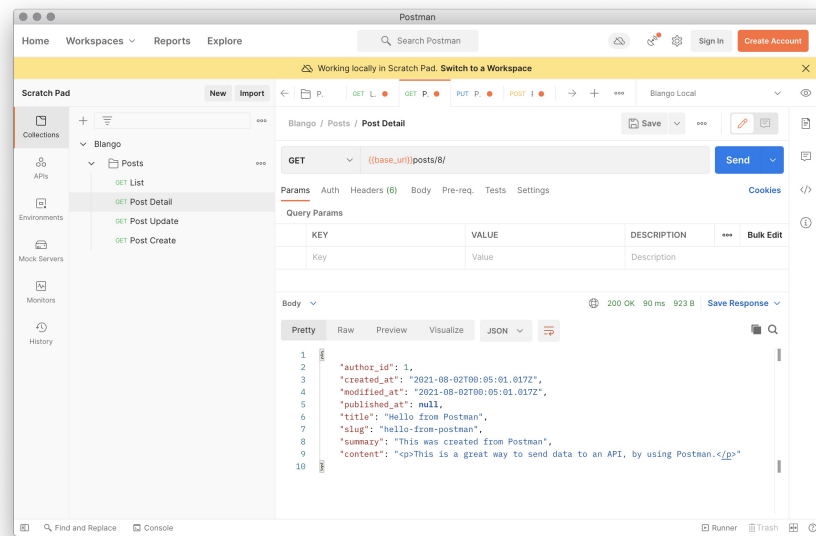
[Click here to see a larger version of the image](#)

Find the *Location* header in the list of headers



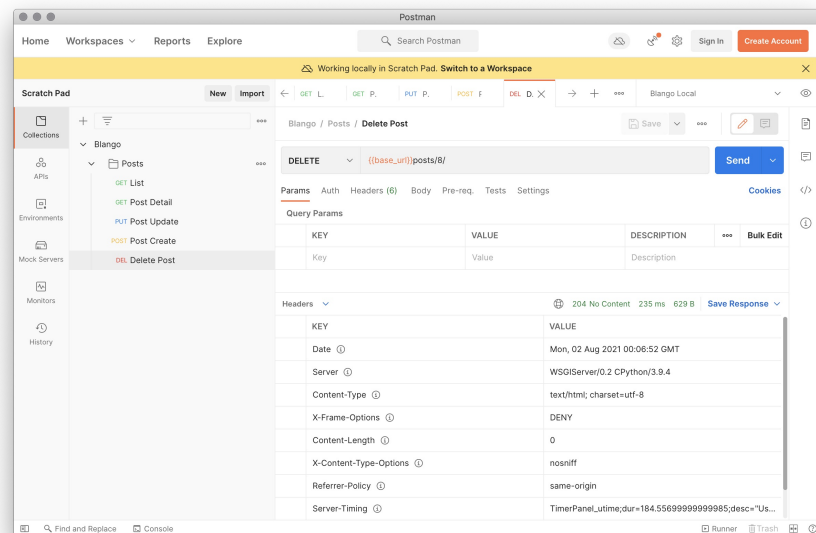
[Click here to see a larger version of the image](#)

From this path, you can see the ID of the new Post, and you can put that into your *Post Detail* request's URL, and see the new Post that was created.



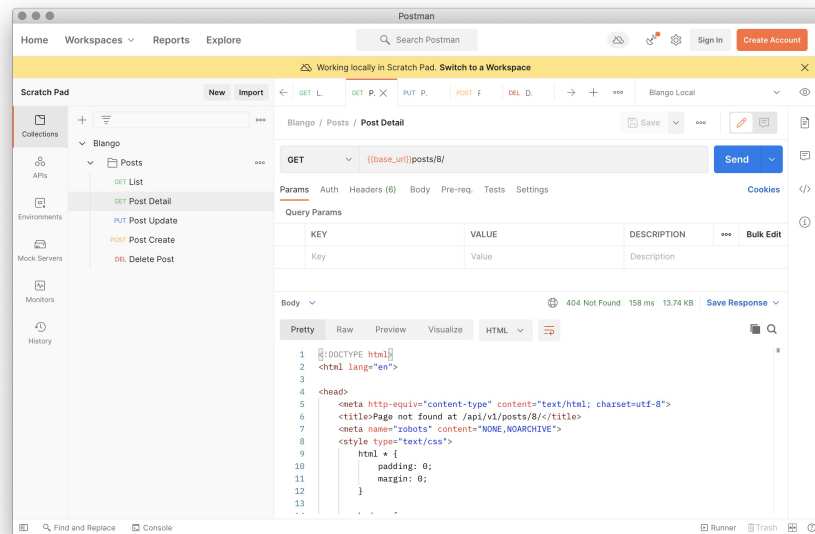
[Click here to see a larger version of the image](#)

Return back to the *Post Create* request and make sure to save it. Then we'll set up our final request. Create a new request called *Delete Post*, and set its URL to the same as the detail request's URL. It can be the one you just created, for example, `{{base_url}}posts/8/` in our screenshot. Then, set the method to **DELETE**. When you're ready to delete the Post, click **Send**.



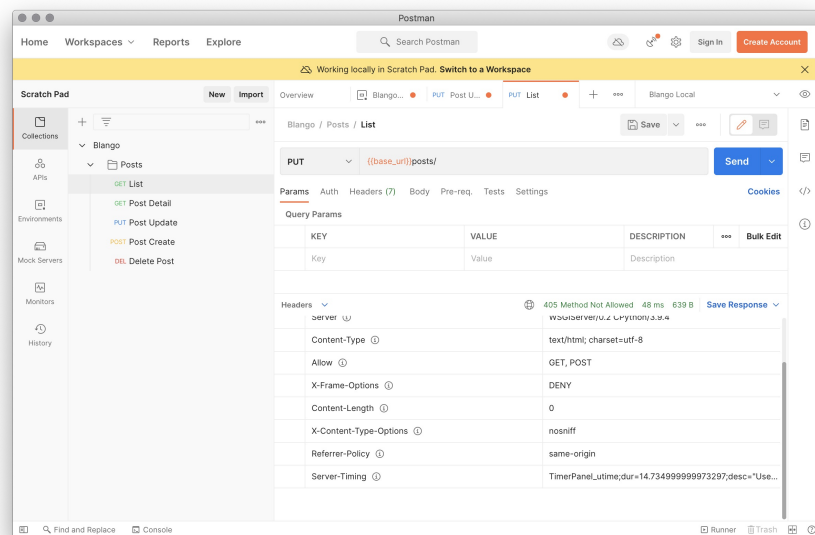
[Click here to see a larger version of the image](#)

You'll notice that the request returns another 204 response with no content, indicating that the Post was deleted successfully. Save the delete request and go back into the *Post Detail* request, you'll notice that if you send it again, you'll receive a 404 Not Found response. Also to note that this response is the default Django plain HTML error page, and not one that is structured data. This is another limitation of our basic API.



[Click here to see a larger version of the image](#)

Finally, just for fun, try sending an incorrect method. Temporarily change the method on the *List* request to **PUT** and send it. You'll get back a 405 response, and if you check the headers, under **Allow** you'll see the methods it supports: GET and POST.



[Click here to see a larger version of the image](#)

Change the method of this request back to **GET**.

That's all we're going to look at with Postman right now, we will be using it to test our API endpoints throughout the course. But since it's uncovered some of the limitations of our API, let's talk about them.

Our Primitive API Limitations

We'll just list some of the things wrong with our API as we've built it:

- There's no authentication. We could decorate the views with the `login_required` decorator, but then API would need to log in through the HTML login form and supply the CSRF token. Not an ideal solution.
- We're not strict on REST ideals, for example, being able to update only part of a Post using a PUT request.
- No support for validating the data in the request, except for when saving of objects failed, and even then:
- Errors will be returned in the format of the standard Django error display, there's no support for a structured text version of errors that can be parsed and shown to end users easily. For example, to render the text in a native mobile app alert panel.
- Our 405 response for incorrect method needs to be kept up to date manually. If we added support for, say, `OPTIONS` or `HEAD`, we'd need to remember to add them to the `HttpResponseNotAllowed` return value.
- We only support JSON, and assume that all content is sent as JSON.

We could implement fixes to these issues ourselves, but in the next module we'll introduce Django Rest Framework and will show how it can help resolve these issues while also requiring us to write less code.