



Python and REST APIs: Interacting With Web Services

by [Jason Van Schooneveld](#) ⌚ Jul 28, 2021 💬 56 Comments 🏷️ [api](#) [intermediate](#) [web-dev](#)

Mark as Completed

🔖

🔗 Share

📱 Share

✉️ Email

Table of Contents

- [REST Architecture](#)
- [REST APIs and Web Services](#)
 - [HTTP Methods](#)
 - [Status Codes](#)
 - [API Endpoints](#)
- [REST and Python: Consuming APIs](#)
 - [GET](#)
 - [POST](#)
 - [PUT](#)
 - [PATCH](#)
 - [DELETE](#)
- [REST and Python: Building APIs](#)
 - [Identify Resources](#)
 - [Define Your Endpoints](#)
 - [Pick Your Data Interchange Format](#)
 - [Design Success Responses](#)
 - [Design Error Responses](#)
- [REST and Python: Tools of the Trade](#)
 - [Flask](#)
 - [Django REST Framework](#)
 - [FastAPI](#)
- [Conclusion](#)

Help



Master Real-World Python Skills
With a Community of Experts
Level Up With Unlimited Access to Our Vast Library
of Python Tutorials and Video Lessons

Watch Now »

 [Remove ads](#)


There's an amazing amount of data available on the Web. Many **web services**, like YouTube and GitHub, make their data accessible to third-party applications through an **application programming interface (API)**. One of the most popular ways to build APIs is the **REST** architecture style. Python provides some great tools not only to get data from REST APIs but also to build your own Python REST APIs.

In this tutorial, you'll learn:

- What **REST** architecture is
- How **REST APIs** provide access to web data
- How to consume data from REST APIs using the **requests** library
- What steps to take to **build a REST API**
- What some popular **Python tools** are for building REST APIs

By using Python and REST APIs, you can retrieve, parse, update, and manipulate the data provided by any web service you're interested in.

Free Bonus: [Click here to download a copy of the "REST API Examples" Guide](#) and get a hands-on introduction to Python + REST API principles with actionable examples.

 **Take the Quiz:** Test your knowledge with our interactive “Python and REST APIs: Interacting With Web Services” quiz. Upon completion you will receive a score so you can track your learning progress over time:

[Take the Quiz »](#)

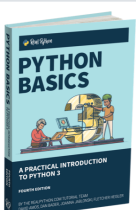
REST Architecture

REST stands for [representational state transfer](#) and is a software architecture style that defines a pattern for [client and server](#) communications over a network. REST provides a set of constraints for software architecture to promote performance, scalability, simplicity, and reliability in the system.

REST defines the following architectural constraints:

- **Stateless:** The server won't maintain any state between requests from the client.
- **Client-server:** The client and server must be decoupled from each other, allowing each to develop independently.
- **Cacheable:** The data retrieved from the server should be cacheable either by the client or by the server.
- **Uniform interface:** The server will provide a uniform interface for accessing resources without defining their representation.
- **Layered system:** The client may access the resources on the server indirectly through other layers such as a [proxy](#) or [load balancer](#).
- **Code on demand (optional):** The server may transfer code to the client that it can run, such as [JavaScript](#) for a single-page application.

Note, REST is *not* a specification but a set of guidelines on how to architect a network-connected software system.



[Your Practical Introduction to Python 3 »](#)

 [Remove ads](#)

REST APIs and Web Services

A **REST web service** is any web service that adheres to REST architecture constraints. These web services expose their data to the outside world through an API. REST APIs provide access to web service data through public web URLs.

For example, here’s one of the URLs for [GitHub’s REST API](#):

```
https://api.github.com/users/<username>
```

This URL allows you to access information about a specific GitHub user. You access data from a REST API by sending an [HTTP request](#) to a specific URL and processing the response.

HTTP Methods

REST APIs listen for [HTTP methods](#) like GET, POST, and DELETE to know which operations to perform on the web service’s resources. A **resource** is any data available in the web service that can be accessed and manipulated with **HTTP requests** to the REST API. The HTTP method tells the API which action to perform on the resource.

While there are many HTTP methods, the five methods listed below are the most commonly used with REST APIs:

HTTP method	Description
GET	Retrieve an existing resource.
POST	Create a new resource.
PUT	Update an existing resource.
PATCH	Partially update an existing resource.
DELETE	Delete a resource.

A REST API client application can use these five HTTP methods to manage the state of resources in the web service.

Status Codes

Once a REST API receives and processes an HTTP request, it will return an **HTTP response**. Included in this response is an **HTTP status code**. This code provides information about the results of the request. An application sending requests to the API can check the status code and perform actions based on the result. These actions could include handling errors or displaying a success message to a user.

Below is a list of the most common status codes returned by REST APIs:

Code	Meaning	Description
200	OK	The requested action was successful.
201	Created	A new resource was created.
202	Accepted	The request was received, but no modification has been made yet.
204	No Content	The request was successful, but the response has no content.
400	Bad Request	The request was malformed.
401	Unauthorized	The client is not authorized to perform the requested action.
404	Not Found	The requested resource was not found.

Code	Meaning	Description
415	Unsupported Media Type	The request data format is not supported by the server.
422	Unprocessable Entity	The request data was properly formatted but contained invalid or missing data.
500	Internal Server Error	The server threw an error when processing the request.

These ten status codes represent only a small subset of the available [HTTP status codes](#). Status codes are numbered based on the category of the result:

Code range	Category
2xx	Successful operation
3xx	Redirection
4xx	Client error
5xx	Server error

HTTP status codes come in handy when working with REST APIs as you’ll often need to perform different logic based on the results of the request.

API Endpoints

A REST API exposes a set of public URLs that client applications use to access the resources of a web service. These URLs, in the context of an API, are called **endpoints**.

To help clarify this, take a look at the table below. In this table, you’ll see API endpoints for a hypothetical [CRM](#) system. These endpoints are for a customer resource that represents potential customers in the system:

HTTP method	API endpoint	Description
GET	/customers	Get a list of customers.
GET	/customers/<customer_id>	Get a single customer.
POST	/customers	Create a new customer.
PUT	/customers/<customer_id>	Update a customer.
PATCH	/customers/<customer_id>	Partially update a customer.
DELETE	/customers/<customer_id>	Delete a customer.

Each of the endpoints above performs a different action based on the HTTP method.

Note: The base URL for the endpoints has been omitted for brevity. In reality, you’ll need the full URL path to access an API endpoint:

```
https://api.example.com/customers
```

This is the full URL you’d use to access this endpoint. The base URL is everything besides /customers.

You'll note that some endpoints have `<customer_id>` at the end. This notation means you need to append a numeric `customer_id` to the URL to tell the REST API which customer you'd like to work with.

The endpoints listed above represent only one resource in the system. Production-ready REST APIs often have tens or even hundreds of different endpoints to manage the resources in the web service.



[Learn Python »](#)

 [Remove ads](#)

REST and Python: Consuming APIs

To write code that interacts with REST APIs, most Python developers turn to [requests](#) to send HTTP requests. This library abstracts away the complexities of making HTTP requests. It's one of the few projects worth treating as if it's part of the standard library.

To start using `requests`, you need to install it first. You can use [pip](#) to install it:

Shell



```
$ python -m pip install requests
```

Now that you've got `requests` installed, you can start sending HTTP requests.

GET

GET is one of the most common HTTP methods you'll use when working with REST APIs. This method allows you to retrieve resources from a given API. GET is a **read-only** operation, so you shouldn't use it to modify an existing resource.

To test out GET and the other methods in this section, you'll use a service called [JSONPlaceholder](#). This free service provides fake API endpoints that send back responses that `requests` can process.

To try this out, start up the [Python REPL](#) and run the following commands to send a GET request to a JSONPlaceholder endpoint:

Python



```
>>> import requests
>>> api_url = "https://jsonplaceholder.typicode.com/todos/1"
>>> response = requests.get(api_url)
>>> response.json()
{'userId': 1, 'id': 1, 'title': 'delectus aut autem', 'completed': False}
```

This code calls `requests.get()` to send a GET request to `/todos/1`, which responds with the todo item with the ID 1. Then you can call `.json()` on the response object to view the data that came back from the API.

The response data is formatted as [JSON](#), a key-value store similar to a [Python dictionary](#). It's a very popular data format and the de facto interchange format for most REST APIs.

Beyond viewing the JSON data from the API, you can also view other things about the response:

Python



```
>>> response.status_code
200

>>> response.headers["Content-Type"]
'application/json; charset=utf-8'
```

Here, you access `response.status_code` to see the HTTP status code. You can also view the response's [HTTP headers](#) with `response.headers`. This dictionary contains metadata about the response, such as the Content-Type of the response.

POST

Now, take a look at how you use `requests` to `POST` data to a REST API to create a new resource. You'll use `JSONPlaceholder` again, but this time you'll include JSON data in the request. Here's the data that you'll send:

JSON

```
{
  "userId": 1,
  "title": "Buy milk",
  "completed": false
}
```

This JSON contains information for a new `todo` item. Back in the Python REPL, run the following code to create the new `todo`:

Python

```
>>> import requests
>>> api_url = "https://jsonplaceholder.typicode.com/todos"
>>> todo = {"userId": 1, "title": "Buy milk", "completed": False}
>>> response = requests.post(api_url, json=todo)
>>> response.json()
{'userId': 1, 'title': 'Buy milk', 'completed': False, 'id': 201}

>>> response.status_code
201
```

Here, you call `requests.post()` to create a new `todo` in the system.

First, you create a dictionary containing the data for your `todo`. Then you pass this dictionary to the `json` keyword argument of `requests.post()`. When you do this, `requests.post()` automatically sets the request's HTTP header `Content-Type` to `application/json`. It also serializes `todo` into a JSON string, which it appends to the body of the request.

If you don't use the `json` keyword argument to supply the JSON data, then you need to set `Content-Type` accordingly and serialize the JSON manually. Here's an equivalent version to the previous code:

Python

```
>>> import requests
>>> import json
>>> api_url = "https://jsonplaceholder.typicode.com/todos"
>>> todo = {"userId": 1, "title": "Buy milk", "completed": False}
>>> headers = {"Content-Type": "application/json"}
>>> response = requests.post(api_url, data=json.dumps(todo), headers=headers)
>>> response.json()
{'userId': 1, 'title': 'Buy milk', 'completed': False, 'id': 201}

>>> response.status_code
201
```

In this code, you add a `headers` dictionary that contains a single header `Content-Type` set to `application/json`. This tells the REST API that you're sending JSON data with the request.

You then call `requests.post()`, but instead of passing `todo` to the `json` argument, you first call `json.dumps(todo)` to serialize it. After it's serialized, you pass it to the `data` keyword argument. The `data` argument tells `requests` what data to include in the request. You also pass the `headers` dictionary to `requests.post()` to set the HTTP headers manually.

When you call `requests.post()` like this, it has the same effect as the previous code but gives you more control over the request.

Note: `json.dumps()` comes from the `json` package in the standard library. This package provides useful methods for working with [JSON in Python](#).

Once the API responds, you call `response.json()` to view the JSON. The JSON includes a generated `id` for the new `todo`. The `201` status code tells you that a new resource was created.



Become a Python Expert »

 [Remove ads](#)

PUT

Beyond GET and POST, requests provides support for all the other HTTP methods you would use with a REST API. The following code sends a PUT request to update an existing todo with new data. Any data sent with a PUT request will completely replace the existing values of the todo.

You'll use the same JSONPlaceholder endpoint you used with GET and POST, but this time you'll append 10 to the end of the URL. This tells the REST API which todo you'd like to update:

Python



```
>>> import requests
>>> api_url = "https://jsonplaceholder.typicode.com/todos/10"
>>> response = requests.get(api_url)
>>> response.json()
{'userId': 1, 'id': 10, 'title': 'illo est ... aut', 'completed': True}

>>> todo = {"userId": 1, "title": "Wash car", "completed": True}
>>> response = requests.put(api_url, json=todo)
>>> response.json()
{'userId': 1, 'title': 'Wash car', 'completed': True, 'id': 10}

>>> response.status_code
200
```

Here, you first call `requests.get()` to view the contents of the existing todo. Next, you call `requests.put()` with new JSON data to replace the existing to-do's values. You can see the new values when you call `response.json()`. Successful PUT requests will always return 200 instead of 201 because you aren't creating a new resource but just updating an existing one.

PATCH

Next up, you'll use `requests.patch()` to modify the value of a specific field on an existing todo. PATCH differs from PUT in that it doesn't completely replace the existing resource. It only modifies the values set in the JSON sent with the request.

You'll use the same todo from the last example to try out `requests.patch()`. Here are the current values:

Python

```
{'userId': 1, 'title': 'Wash car', 'completed': True, 'id': 10}
```

Now you can update the title with a new value:

Python



```
>>> import requests
>>> api_url = "https://jsonplaceholder.typicode.com/todos/10"
>>> todo = {"title": "Mow lawn"}
>>> response = requests.patch(api_url, json=todo)
>>> response.json()
{'userId': 1, 'id': 10, 'title': 'Mow lawn', 'completed': True}

>>> response.status_code
200
```

When you call `response.json()`, you can see that title was updated to Mow lawn.

DELETE

Last but not least, if you want to completely remove a resource, then you use DELETE. Here’s the code to remove a todo:

Python

```
>>> import requests
>>> api_url = "https://jsonplaceholder.typicode.com/todos/10"
>>> response = requests.delete(api_url)
>>> response.json()
{}

>>> response.status_code
200
```

You call `requests.delete()` with an API URL that contains the ID for the `todo` you would like to remove. This sends a DELETE request to the REST API, which then removes the matching resource. After deleting the resource, the API sends back an empty JSON object indicating that the resource has been deleted.

The `requests` library is an awesome tool for working with REST APIs and an indispensable part of your Python tool belt. In the next section, you’ll change gears and consider what it takes to build a REST API.

REST and Python: Building APIs

REST API design is a huge topic with many layers. As with most things in technology, there’s a wide range of opinions on the best approach to building APIs. In this section, you’ll look at some recommended steps to follow as you build an API.

Identify Resources

The first step you’ll take as you build a REST API is to identify the resources the API will manage. It’s common to describe these resources as plural nouns, like `customers`, `events`, or `transactions`. As you identify different resources in your web service, you’ll build out a list of nouns that describe the different data users can manage in the API.

As you do this, make sure to consider any nested resources. For example, `customers` may have `sales`, or `events` may contain `guests`. Establishing these resource hierarchies will help when you define API endpoints.

Write Cleaner & More Pythonic Code

realpython.com



 [Remove ads](#)

Define Your Endpoints

Once you’ve identified the resources in your web service, you’ll want to use these to define the API endpoints. Here are some example endpoints for a `transactions` resource you might find in an API for a payment processing service:

HTTP method	API endpoint	Description
GET	/transactions	Get a list of transactions.
GET	/transactions/<transaction_id>	Get a single transaction.
POST	/transactions	Create a new transaction.
PUT	/transactions/<transaction_id>	Update a transaction.
PATCH	/transactions/<transaction_id>	Partially update a transaction.
DELETE	/transactions/<transaction_id>	Delete a transaction.

These six endpoints cover all the operations that you’ll need to create, read, update, and delete transactions in the web service. Each resource in your web service would have a similar list of endpoints based on what actions a user can perform with the API.

Note: An endpoint shouldn’t contain verbs. Instead, you should select the appropriate HTTP methods to convey the endpoint’s action. For example, the endpoint below contains an unneeded verb:

HTTP

```
GET /getTransactions
```

Here, `get` is included in the endpoint when it isn’t needed. The HTTP method `GET` already provides the semantic meaning for the endpoint by indicating the action. You can remove `get` from the endpoint:

HTTP

```
GET /transactions
```

This endpoint contains only a plural noun, and the HTTP method `GET` communicates the action.

Now take a look at an example of endpoints for a nested resource. Here, you’ll see endpoints for `guests` that are nested under `events` resources:

HTTP method	API endpoint	Description
GET	<code>/events/<event_id>/guests</code>	Get a list of guests.
GET	<code>/events/<event_id>/guests/<guest_id></code>	Get a single guest.
POST	<code>/events/<event_id>/guests</code>	Create a new guest.
PUT	<code>/events/<event_id>/guests/<guest_id></code>	Update a guest.
PATCH	<code>/events/<event_id>/guests/<guest_id></code>	Partially update a guest.
DELETE	<code>/events/<event_id>/guests/<guest_id></code>	Delete a guest.

With these endpoints, you can manage `guests` for a specific event in the system.

This isn’t the only way to define an endpoint for nested resources. Some people prefer to use [query strings](#) to access a nested resource. A query string allows you to send additional parameters with your HTTP request. In the following endpoint, you append a query string to get `guests` for a specific `event_id`:

HTTP

```
GET /guests?event_id=23
```

This endpoint will filter out any `guests` that don’t reference the given `event_id`. As with many things in API design, you need to decide which method fits your web service best.

Note: It’s very unlikely that your REST API will stay the same throughout the life of your web service. Resources will change, and you’ll need to update your endpoints to reflect these changes. This is where **API versioning** comes in. API versioning allows you to modify your API without fear of breaking existing integrations.

There’s a wide range of versioning strategies. Selecting the right option depends on the requirements of your API. Below are some of the most popular options for API versioning:

- [URI versioning](#)
- [HTTP header versioning](#)
- [Query string versioning](#)

- [Media type versioning](#)

No matter what strategy you select, versioning your API is an important step to ensuring it can adapt to changing requirements while supporting existing users.

Now that you’ve covered endpoints, in the next section you’ll look at some options for formatting data in your REST API.

Pick Your Data Interchange Format

Two popular options for formatting web service data are [XML](#) and JSON. Traditionally, XML was very popular with [SOAP](#) APIs, but JSON is more popular with REST APIs. To compare the two, take a look at an example book resource formatted as XML and JSON.

Here’s the book formatted as XML:

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<book>
  <title>Python Basics</title>
  <page_count>635</page_count>
  <pub_date>2021-03-16</pub_date>
  <authors>
    <author>
      <name>David Amos</name>
    </author>
    <author>
      <name>Joanna Jablonski</name>
    </author>
    <author>
      <name>Dan Bader</name>
    </author>
    <author>
      <name>Fletcher Heisler</name>
    </author>
  </authors>
  <isbn13>978-1775093329</isbn13>
  <genre>Education</genre>
</book>
```

XML uses a series of **elements** to encode data. Each element has an opening and closing tag, with the data in between. Elements can be nested inside other elements. You can see this above, where several `<author>` tags are nested inside of `<authors>`.

Now, take a look at the same book in JSON:

JSON

```
{
  "title": "Python Basics",
  "page_count": 635,
  "pub_date": "2021-03-16",
  "authors": [
    {"name": "David Amos"},
    {"name": "Joanna Jablonski"},
    {"name": "Dan Bader"},
    {"name": "Fletcher Heisler"}
  ],
  "isbn13": "978-1775093329",
  "genre": "Education"
}
```

JSON stores data in key-value pairs similar to a Python dictionary. Like XML, JSON supports nesting data to any level, so you can model complex data.

Neither JSON nor XML is inherently better than the other, but there’s a preference for JSON among REST API developers. This is especially true when you pair a REST API with a front-end framework like [React](#) or [Vue](#).

Python Tricks The Book

A Buffet of Awesome Python Features

Get Your Free Sample Chapter



Remove ads

Design Success Responses

Once you’ve picked a data format, the next step is to decide how you’ll respond to HTTP requests. All responses from your REST API should have a similar format and include the proper HTTP status code.

In this section, you’ll look at some example HTTP responses for a hypothetical API that manages an inventory of cars. These examples will give you a sense of how you should format your API responses. To make things clear, you’ll look at raw HTTP requests and responses instead of using an HTTP library like requests.

To start things off, take a look at a GET request to `/cars`, which returns a list of cars:

HTTP

```
GET /cars HTTP/1.1
Host: api.example.com
```

This HTTP request is made up of four parts:

- 1. `GET` is the HTTP method type.
- 2. `/cars` is the API endpoint.
- 3. `HTTP/1.1` is the HTTP version.
- 4. `Host: api.example.com` is the API host.

These four parts are all you need to send a GET request to `/cars`. Now take a look at the response. This API uses JSON as the data interchange format:

HTTP

```
HTTP/1.1 200 OK
Content-Type: application/json
...

[
  {
    "id": 1,
    "make": "GMC",
    "model": "1500 Club Coupe",
    "year": 1998,
    "vin": "1D7RV1GTXAS806941",
    "color": "Red"
  },
  {
    "id": 2,
    "make": "Lamborghini",
    "model": "Gallardo",
    "year": 2006,
    "vin": "JN1BY1PR0FM736887",
    "color": "Mauve"
  },
  {
    "id": 3,
    "make": "Chevrolet",
    "model": "Monte Carlo",
    "year": 1996,
    "vin": "1G4HP54K714224234",
    "color": "Violet"
  }
]
```

The API returns a response that contains a list of cars. You know that the response was successful because of the 200 OK status code. The response also has a Content-Type header set to application/json. This tells the user to parse the response as JSON.

Note: When you're working with a real API, you're going to see more HTTP headers than this. These headers differ between APIs, so they've been excluded in these examples.

It's important to always set the correct Content-Type header on your response. If you send JSON, then set Content-Type to application/json. If XML, then set it to application/xml. This header tells the user how they should parse the data.

You also want to include an appropriate status code in your response. For any successful GET request, you should return 200 OK. This tells the user that their request was processed as expected.

Take a look at another GET request, this time for a single car:

HTTP

```
GET /cars/1 HTTP/1.1
Host: api.example.com
```

This HTTP request queries the API for car 1. Here's the response:

HTTP

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "id": 1,
  "make": "GMC",
  "model": "1500 Club Coupe",
  "year": 1998,
  "vin": "1D7RV1GTXAS806941",
  "color": "Red"
},
```

This response contains a single JSON object with the car's data. Since it's a single object, it doesn't need to be wrapped in a list. Like the last response, this also has a 200 OK status code.

Note: A GET request should never modify an existing resource. If the request contains data, then this data should be ignored and the API should return the resource unchanged.

Next up, check out a POST request to add a new car:

HTTP

```
POST /cars HTTP/1.1
Host: api.example.com
Content-Type: application/json

{
  "make": "Nissan",
  "model": "240SX",
  "year": 1994,
  "vin": "1N6AD0CU5AC961553",
  "color": "Violet"
}
```

This POST request includes JSON for the new car in the request. It sets the Content-Type header to application/json so the API knows the content type of the request. The API will create a new car from the JSON.

Here's the response:

HTTP

```
HTTP/1.1 201 Created
Content-Type: application/json

{
  "id": 4,
  "make": "Nissan",
  "model": "240SX",
  "year": 1994,
  "vin": "1N6AD0CU5AC961553",
  "color": "Violet"
}
```

This response has a 201 Created status code to tell the user that a new resource was created. Make sure to use 201 Created instead of 200 OK for all successful POST requests.

This response also includes a copy of the new car with an id generated by the API. It's important to send back an id in the response so that the user can modify the resource again.

Note: It's important to always send back a copy of a resource when a user creates it with POST or modifies it with PUT or PATCH. This way, the user can see the changes that they've made.

Now take a look at a PUT request:

HTTP

```
PUT /cars/4 HTTP/1.1
Host: api.example.com
Content-Type: application/json

{
  "make": "Buick",
  "model": "Lucerne",
  "year": 2006,
  "vin": "4T1BF3EK8AU335094",
  "color": "Maroon"
}
```

This request uses the id from the previous request to update the car with all new data. As a reminder, PUT updates all fields on the resource with new data. Here's the response:

HTTP

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "id": 4,
  "make": "Buick",
  "model": "Lucerne",
  "year": 2006,
  "vin": "4T1BF3EK8AU335094",
  "color": "Maroon"
}
```

The response includes a copy of the car with the new data. Again, you always want to send back the full resource for a PUT request. The same applies to a PATCH request:

HTTP


```
PATCH /cars/4 HTTP/1.1
Host: api.example.com
Content-Type: application/json

{
  "vin": "VNKKTUD32FA050307",
  "color": "Green"
}
```

PATCH requests only update a part of a resource. In the request above, the `vin` and `color` fields will be updated with new values. Here’s the response:

```
HTTP
HTTP/1.1 200 OK
Content-Type: application/json

{
  "id": 4,
  "make": "Buick",
  "model": "Lucerne",
  "year": 2006,
  "vin": "VNKKTUD32FA050307",
  "color": "Green"
}
```

The response contains a full copy of the `car`. As you can see, only the `vin` and `color` fields have been updated.

Finally, take a look at how your REST API should respond when it receives a DELETE request. Here’s a DELETE request to remove a car:

```
HTTP
DELETE /cars/4 HTTP/1.1
```

This DELETE request tells the API to remove the `car` with the ID 4. Here’s the response:

```
HTTP
HTTP/1.1 204 No Content
```

This response only includes the status code `204 No Content`. This status code tells a user that the operation was successful, but no content was returned in the response. This makes sense since the `car` has been deleted. There’s no reason to send a copy of it back in the response.

The responses above work well when everything goes as planned, but what happens if there’s a problem with the request? In the next section, you’ll look at how your REST API should respond when errors occur.

Learn Python Programming, By Example

realpython.com

```
GET /motorcycles HTTP/1.1
Host: api.example.com
```

Here, the user sends a GET request to `/motorcycles`, which doesn't exist. The API sends back the following response:

HTTP

```
HTTP/1.1 404 Not Found
Content-Type: application/json
...

{
  "error": "The requested resource was not found."
}
```

This response includes a `404 Not Found` status code. Along with this, the response contains a JSON object with a descriptive error message. Providing a descriptive error message gives the user more context for the error.

Now take a look at the error response when the user sends an invalid request:

HTTP

```
POST /cars HTTP/1.1
Host: api.example.com
Content-Type: application/json

{
  "make": "Nissan",
  "year": 1994,
  "color": "Violet"
```

This POST request contains JSON, but it isn't formatted correctly. It's missing a closing curly brace (`}`) at the end. The API won't be able to process this data. The error response tells the user about the issue:

HTTP

```
HTTP/1.1 400 Bad Request
Content-Type: application/json

{
  "error": "This request was not properly formatted. Please send again."
}
```

This response includes a descriptive error message along with the `400 Bad Request` status code, telling the user they need to fix the request.

There are several other ways that the request can be wrong even if it's formatted properly. In this next example, the user sends a POST request but includes an unsupported media type:

HTTP

```
POST /cars HTTP/1.1
Host: api.example.com
Content-Type: application/xml

<?xml version="1.0" encoding="UTF-8" ?>
<car>
  <make>Nissan</make>
  <model>240SX</model>
  <year>1994</year>
  <vin>1N6AD0CU5AC961553</vin>
  <color>Violet</color>
</car>
```

In this request, the user sends XML, but the API only supports JSON. The API responds with this:

HTTP

```
HTTP/1.1 415 Unsupported Media Type
Content-Type: application/json

{
  "error": "The application/xml mediatype is not supported."
}
```

This response includes the 415 `Unsupported Media Type` status code to indicate that the `POST` request included a data format that isn't supported by the API. This error code makes sense for data that's in the wrong format, but what about data that's invalid even with the correct format?

In this next example, the user sends a `POST` request but includes car data that doesn't match fields of the other data:

```
HTTP

POST /cars HTTP/1.1
Host: api.example.com
Content-Type: application/json

{
  "make": "Nissan",
  "model": "240SX",
  "topSpeed": 120
  "warrantyLength": 10
}
```

In this request, the user adds `topSpeed` and `warrantyLength` fields to the JSON. These fields aren't supported by the API, so it responds with an error message:

```
HTTP

HTTP/1.1 422 Unprocessable Entity
Content-Type: application/json

{
  "error": "Request had invalid or missing data."
}
```

This response includes the 422 `Unprocessable Entity` status code. This status code indicates that there weren't any issues with the request, but the data was invalid. A REST API needs to validate incoming data. If the user sends data with the request, then the API should validate the data and inform the user of any errors.

Responding to requests, both successful and erroneous, is one of the most important jobs of a REST API. If your API is intuitive and provides accurate responses, then it'll be easier for users to build applications around your web service. Luckily, some great Python web frameworks abstract away the complexities of processing HTTP requests and returning responses. You'll look at three popular options in the next section.



 [Remove ads](#)

REST and Python: Tools of the Trade

In this section, you'll look at three popular frameworks for building REST APIs in Python. Each framework has pros and cons, so you'll have to evaluate which works best for your needs. To this end, in the next sections, you'll look at a REST API in each framework. All the examples will be for a similar API that manages a collection of countries.

Each country will have the following fields:

- **name** is the name of the country.
- **capital** is the capital of the country.

- **area** is the area of the country in square kilometers.

The fields `name`, `capital`, and `area` store data about a specific country somewhere in the world.

Most of the time, data sent from a REST API comes from a database. [Connecting to a database](#) is beyond the scope of this tutorial. For the examples below, you'll store your data in a Python list. The exception to this is the Django REST framework example, which runs off the SQLite database that Django creates.

Note: It's advised that you create individual folders for each of the examples to separate the source files. You'll also want to use [virtual environments](#) to isolate dependencies.

To keep things consistent, you'll use `countries` as your main endpoint for all three frameworks. You'll also use JSON as your data format for all three frameworks.

Now that you've got the background for the API, you can move on to the next section, where you'll look at the REST API in **Flask**.

Flask

[Flask](#) is a Python microframework used to build web applications and REST APIs. Flask provides a solid backbone for your applications while leaving many design choices up to you. Flask's main job is to handle HTTP requests and route them to the appropriate function in the application.

Note: The code in this section uses the new [Flask 2](#) syntax. If you're running an older version of Flask, then use `@app.route("/countries").` instead of `@app.get("/countries").` and `@app.post("/countries").`

To handle POST requests in older versions of [Flask](#), you'll also need to add the `methods` parameter to `@app.route()`:

Python

```
@app.route("/countries", methods=["POST"])
```

This route handles POST requests to `/countries` in Flask 1.

Below is an example Flask application for the REST API:

Python

```
# app.py
from flask import Flask, request, jsonify

app = Flask(__name__)

countries = [
    {"id": 1, "name": "Thailand", "capital": "Bangkok", "area": 513120},
    {"id": 2, "name": "Australia", "capital": "Canberra", "area": 7617930},
    {"id": 3, "name": "Egypt", "capital": "Cairo", "area": 1010408},
]

def _find_next_id():
    return max(country["id"] for country in countries) + 1

@app.get("/countries")
def get_countries():
    return jsonify(countries)

@app.post("/countries")
def add_country():
    if request.is_json:
        country = request.get_json()
        country["id"] = _find_next_id()
        countries.append(country)
        return country, 201
    return {"error": "Request must be JSON"}, 415
```

This application defines the API endpoint `/countries` to manage the list of countries. It handles two different kinds of requests:

1. `GET /countries` returns the list of countries.
2. `POST /countries` adds a new country to the list.

Note: This Flask application includes functions to handle only two types of requests to the API endpoint, `/countries`. In a full REST API, you'd want to expand this to include functions for all the required operations.

You can try out this application by installing `flask` with `pip`:

Shell



```
$ python -m pip install flask
```

Once `flask` is installed, save the code in a file called `app.py`. To run this Flask application, you first need to set an environment variable called `FLASK_APP` to `app.py`. This tells Flask which file contains your application.

Run the following command inside the folder that contains `app.py`:

Shell



```
$ export FLASK_APP=app.py
```

This sets `FLASK_APP` to `app.py` in the current shell. Optionally, you can set `FLASK_ENV` to `development`, which puts Flask in **debug mode**:

Shell



```
$ export FLASK_ENV=development
```

Besides providing helpful error messages, debug mode will trigger a reload of the application after all code changes. Without debug mode, you'd have to restart the server after every change.

Note: The above commands work on macOS or Linux. If you're running this on Windows, then you need to set `FLASK_APP` and `FLASK_ENV` like this in the Command Prompt:

Windows Command Prompt



```
C:\> set FLASK_APP=app.py
C:\> set FLASK_ENV=development
```

Now `FLASK_APP` and `FLASK_ENV` are set inside the Windows shell.

With all the environment variables ready, you can now start the Flask development server by calling `flask run`:

Shell



```
$ flask run
* Serving Flask app "app.py" (lazy loading)
* Environment: development
* Debug mode: on
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

This starts up a server running the application. Open up your browser and go to `http://127.0.0.1:5000/countries`, and you'll see the following response:

JSON


```
[
  {
    "area": 513120,
    "capital": "Bangkok",
    "id": 1,
    "name": "Thailand"
  },
  {
    "area": 7617930,
    "capital": "Canberra",
    "id": 2,
    "name": "Australia"
  },
  {
    "area": 1010408,
    "capital": "Cairo",
    "id": 3,
    "name": "Egypt"
  }
]
```

This JSON response contains the three countries defined at the start of `app.py`. Take a look at the following code to see how this works:

Python

```
@app.get("/countries")
def get_countries():
    return jsonify(countries)
```

This code uses `@app.get()`, a Flask route [decorator](#), to connect GET requests to a function in the application. When you access `/countries`, Flask calls the decorated function to handle the HTTP request and [return](#) a response.

In the code above, `get_countries()` takes `countries`, which is a [Python list](#), and converts it to JSON with `jsonify()`. This JSON is returned in the response.

Note: Most of the time, you can just return a Python dictionary from a Flask function. Flask will automatically convert any Python dictionary to JSON. You can see this in action with the function below:

Python

```
@app.get("/country")
def get_country():
    return countries[1]
```

In this code, you return the second dictionary from `countries`. Flask will convert this dictionary to JSON. Here's what you'll see when you request `/country`:

JSON

```
{
  "area": 7617930,
  "capital": "Canberra",
  "id": 2,
  "name": "Australia"
}
```

This is the JSON version of the dictionary you returned from `get_country()`.

In `get_countries()`, you need to use `jsonify()` because you're returning a list of dictionaries and not just a single dictionary. Flask doesn't automatically convert lists to JSON.

Now take a look at `add_country()`. This function handles POST requests to `/countries` and allows you to add a new country to the list. It uses the Flask [request](#) object to get information about the current HTTP request:

Python

```
@app.post("/countries")
def add_country():
    if request.is_json:
        country = request.get_json()
        country["id"] = _find_next_id()
        countries.append(country)
        return country, 201
    return {"error": "Request must be JSON"}, 415
```

This function performs the following operations:

1. Using `request.is_json` to check that the request is JSON
2. Creating a new country instance with `request.get_json()`
3. Finding the next id and setting it on the country
4. Appending the new country to `countries`
5. Returning the country in the response along with a 201 Created status code
6. Returning an error message and 415 Unsupported Media Type status code if the request wasn't JSON

`add_country()` also calls `_find_next_id()` to determine the id for the new country:

Python

```
def _find_next_id():
    return max(country["id"] for country in countries) + 1
```

This helper function uses a [generator expression](#) to select all the country IDs and then calls `max()` on them to get the largest value. It increments this value by 1 to get the next ID to use.

You can try out this endpoint in the shell using the command-line tool [curl](#), which allows you to send HTTP requests from the command line. Here, you'll add a new country to the list of countries:

Shell



```
$ curl -i http://127.0.0.1:5000/countries \
-X POST \
-H 'Content-Type: application/json' \
-d '{"name": "Germany", "capital": "Berlin", "area": 357022}'

HTTP/1.0 201 CREATED
Content-Type: application/json
...

{
  "area": 357022,
  "capital": "Berlin",
  "id": 4,
  "name": "Germany"
}
```

This curl command has some options that are helpful to know:

- `-X` sets the HTTP method for the request.
- `-H` adds an HTTP header to the request.
- `-d` defines the request data.

With these options set, curl sends JSON data in a POST request with the Content-Type header set to `application/json`. The REST API returns 201 CREATED along with the JSON for the new country you added.

Note: In this example, `add_country()` doesn't contain any validation to confirm that the JSON in the request matches the format of `countries`. Check out [flask-expects-json](#) if you'd like to validate the format of JSON in Flask.

You can use curl to send a GET request to `/countries` to confirm that the new country was added. If you don't use `-x` in your curl command, then it sends a GET request by default:

Shell



```
$ curl -i http://127.0.0.1:5000/countries
```

```
HTTP/1.0 200 OK
Content-Type: application/json
```

```
...
```

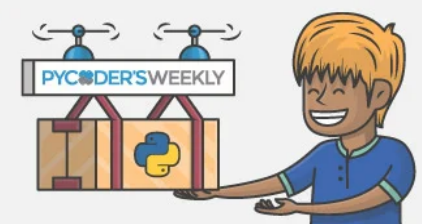
```
[
  {
    "area": 513120,
    "capital": "Bangkok",
    "id": 1,
    "name": "Thailand"
  },
  {
    "area": 7617930,
    "capital": "Canberra",
    "id": 2,
    "name": "Australia"
  },
  {
    "area": 1010408,
    "capital": "Cairo",
    "id": 3,
    "name": "Egypt"
  },
  {
    "area": 357022,
    "capital": "Berlin",
    "id": 4,
    "name": "Germany"
  }
]
```

This returns the full list of countries in the system, with the newest country at the bottom.

This is just a sampling of what Flask can do. This application could be expanded to include endpoints for all the other HTTP methods. Flask also has a large ecosystem of extensions that provide additional functionality for REST APIs, such as [database integrations](#), [authentication](#), and background processing.

Your Weekly Dose of All Things Python!

pycoders.com



[Remove ads](#)

Django REST Framework

Another popular option for building REST APIs is [Django REST framework](#). Django REST framework is a [Django](#) plugin that adds REST API functionality on top of an existing Django project.

To use Django REST framework, you need a Django project to work with. If you already have one, then you can apply the patterns in the section to your project. Otherwise, follow along and you'll build a Django project and add in Django REST framework.

First, install Django and `django-rest-framework` with pip:

Shell



```
$ python -m pip install Django django-rest-framework
```

This installs Django and `django-rest-framework`. You can now use the `django-admin` tool to create a new Django project. Run the following command to start your project:

Shell



```
$ django-admin startproject countryapi
```

This command creates a new folder in your current directory called `countryapi`. Inside this folder are all the files you need to run your Django project. Next, you're going to create a new **Django application** inside your project. Django breaks up the functionality of a project into applications. Each application manages a distinct part of the project.

Note: You're only going to scratch the surface of what Django can do in this tutorial. If you're interested in learning more, then check out the available [Django tutorials](#).

To create the application, change directories to `countryapi` and run the following command:

Shell



```
$ python manage.py startapp countries
```

This creates a new `countries` folder inside your project. Inside this folder are the base files for this application.

Now that you've created an application to work with, you need to tell Django about it. Alongside the `countries` folder that you just created is another folder called `countryapi`. This folder contains configurations and settings for your project.

Note: This folder has the same name as the root folder that Django created when you ran `django-admin startproject countryapi`.

Open up the `settings.py` file that's inside the `countryapi` folder. Add the following lines to `INSTALLED_APPS` to tell Django about the `countries` application and Django REST framework:

Python

```
# countryapi/settings.py
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "rest_framework",
    "countries",
]
```

You've added a line for the `countries` application and `rest_framework`.

You may be wondering why you need to add `rest_framework` to the applications list. You need to add it because Django REST framework is just another Django application. Django plugins are Django applications that are packaged up and distributed and that anyone can use.

The next step is to create a Django model to define the fields of your data. Inside of the `countries` application, update `models.py` with the following code:

Python

```
# countries/models.py
from django.db import models

class Country(models.Model):
    name = models.CharField(max_length=100)
    capital = models.CharField(max_length=100)
    area = models.IntegerField(help_text="(in square kilometers)")
```

This code defines a `Country` model. Django will use this model to create the database table and columns for the country data.

Run the following commands to have Django update the database based on this model:

Shell



```
$ python manage.py makemigrations
Migrations for 'countries':
  countries/migrations/0001_initial.py
    - Create model Country

$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, countries, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  ...
```

These commands use [Django migrations](#) to create a new table in the database.

This table starts empty, but it would be nice to have some initial data so you can test Django REST framework. To do this, you're going to use a [Django fixture](#) to load some data in the database.

Copy and save the following JSON data into a file called `countries.json` and save it inside the `countries` directory:

JSON

```
[
  {
    "model": "countries.country",
    "pk": 1,
    "fields": {
      "name": "Thailand",
      "capital": "Bangkok",
      "area": 513120
    }
  },
  {
    "model": "countries.country",
    "pk": 2,
    "fields": {
      "name": "Australia",
      "capital": "Canberra",
      "area": 7617930
    }
  },
  {
    "model": "countries.country",
    "pk": 3,
    "fields": {
      "name": "Egypt",
      "capital": "Cairo",
      "area": 1010408
    }
  }
]
```

This JSON contains database entries for three countries. Call the following command to load this data in the database:

Shell



```
$ python manage.py loaddata countries.json
Installed 3 object(s) from 1 fixture(s)
```

This adds three rows to the database.

With that, your Django application is all set up and populated with some data. You can now start adding Django REST framework to the project.

Django REST framework takes an existing Django model and converts it to JSON for a REST API. It does this with **model serializers**. A model serializer tells Django REST framework how to convert a model instance into JSON and what data to include.

You’ll create your serializer for the Country model from above. Start by creating a file called `serializers.py` inside of the `countries` application. Once you’ve done that, add the following code to `serializers.py`:

Python

```
# countries/serializers.py
from rest_framework import serializers
from .models import Country

class CountrySerializer(serializers.ModelSerializer):
    class Meta:
        model = Country
        fields = ["id", "name", "capital", "area"]
```

This serializer, `CountrySerializer`, subclasses `serializers.ModelSerializer` to automatically generate JSON content based on the model fields of `Country`. Unless specified, a `ModelSerializer` subclass will include all fields from the Django model in the JSON. You can modify this behavior by setting `fields` to a list of data you wish to include.

Just like Django, Django REST framework uses [views](#) to query data from the database to display to the user. Instead of writing REST API views from scratch, you can subclass Django REST framework’s [ModelViewSet](#) class, which has default views for common REST API operations.

Note: The Django REST framework documentation refers to these views as [actions](#).

Here’s a list of the actions that `ModelViewSet` provides and their equivalent HTTP methods:

HTTP method	Action	Description
GET	<code>.list()</code>	Get a list of countries.
GET	<code>.retrieve()</code>	Get a single country.
POST	<code>.create()</code>	Create a new country.
PUT	<code>.update()</code>	Update a country.
PATCH	<code>.partial_update()</code>	Partially update a country.
DELETE	<code>.destroy()</code>	Delete a country.

As you can see, these actions map to the standard HTTP methods you’d expect in a REST API. You can [override these actions](#) in your subclass or [add additional actions](#) based on the requirements of your API.

Below is the code for a `ModelViewSet` subclass called `CountryViewSet`. This class will generate the views needed to manage Country data. Add the following code to `views.py` inside the `countries` application:

Python

```
# countries/views.py
from rest_framework import viewsets

from .models import Country
from .serializers import CountrySerializer

class CountryViewSet(viewsets.ModelViewSet):
    serializer_class = CountrySerializer
    queryset = Country.objects.all()
```

In this class, `serializer_class` is set to `CountrySerializer` and `queryset` is set to `Country.objects.all()`. This tells Django REST framework which serializer to use and how to query the database for this specific set of views.

Once the views are created, they need to be mapped to the appropriate URLs or endpoints. To do this, Django REST framework provides a `DefaultRouter` that will automatically generate URLs for a `ModelViewSet`.

Create a `urls.py` file in the `countries` application and add the following code to the file:

Python

```
# countries/urls.py
from django.urls import path, include
from rest_framework.routers import DefaultRouter

from .views import CountryViewSet

router = DefaultRouter()
router.register(r"countries", CountryViewSet)

urlpatterns = [
    path("", include(router.urls))
]
```

This code creates a `DefaultRouter` and registers `CountryViewSet` under the `countries` URL. This will place all the URLs for `CountryViewSet` under `/countries/`.

Note: Django REST framework automatically appends a forward slash (/) to the end of any endpoints generated by `DefaultRouter`. You can disable this behavior like so:

Python

```
router = DefaultRouter(trailing_slash=False)
```

This will disable the forward slash at the end of endpoints.

Finally, you need to update the project's base `urls.py` file to include all the `countries` URLs in the project. Update the `urls.py` file inside of the `countryapi` folder with the following code:

Python

```
# countryapi/urls.py
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path("admin/", admin.site.urls),
    path("", include("countries.urls")),
]
```

This puts all the URLs under `/countries/`. Now you're ready to try out your Django-backed REST API. Run the following command in the root `countryapi` directory to start the Django development server:

Shell



```
$ python manage.py runserver
```

The development server is now running. Go ahead and send a GET request to `/countries/` to get a list of all the countries in your Django project:

Shell



```
$ curl -i http://127.0.0.1:8000/countries/ -w '\n'

HTTP/1.1 200 OK
...

[
  {
    "id": 1,
    "name": "Thailand",
    "capital": "Bangkok",
    "area": 513120
  },
  {
    "id": 2,
    "name": "Australia",
    "capital": "Canberra",
    "area": 7617930
  },
  {
    "id": 3,
    "name": "Egypt",
    "capital": "Cairo",
    "area": 1010408
  }
]
```

Django REST framework sends back a JSON response with the three countries you added earlier. The response above is formatted for readability, so your response will look different.

The [DefaultRouter](#) you created in `countries/urls.py` provides URLs for requests to all the standard API endpoints:

- GET `/countries/`
- GET `/countries/<country_id>/`
- POST `/countries/`
- PUT `/countries/<country_id>/`
- PATCH `/countries/<country_id>/`
- DELETE `/countries/<country_id>/`

You can try out a few more endpoints below. Send a POST request to `/countries/` to create a new Country in your Django project:

Shell



```
$ curl -i http://127.0.0.1:8000/countries/ \
-X POST \
-H 'Content-Type: application/json' \
-d '{"name": "Germany", "capital": "Berlin", "area": 357022}' \
-w '\n'

HTTP/1.1 201 Created
...

{
  "id": 4,
  "name": "Germany",
  "capital": "Berlin",
  "area": 357022
}
```

This creates a new Country with the JSON you sent in the request. Django REST framework returns a `201 Created` status code and the new Country.

Note: By default, the response doesn't include a new line at the end. This means that the JSON may run into your command prompt. The curl command above includes `-w '\n'` to add a newline character after the JSON to fix this issue.

You can view an existing Country by sending a request to GET `/countries/<country_id>/` with an existing id. Run the following command to get the first Country:

Shell



```
$ curl -i http://127.0.0.1:8000/countries/1/ -w '\n'

HTTP/1.1 200 OK
...

{
  "id":1,
  "name":"Thailand",
  "capital":"Bangkok",
  "area":513120
}
```

The response contains the information for the first Country. These examples only covered GET and POST requests. Feel free to try out PUT, PATCH, and DELETE requests on your own to see how you can fully manage your model from the REST API.

As you’ve seen, Django REST framework is a great option for building REST APIs, especially if you have an existing Django project and you want to add an API.

5 Thoughts on Mastering Python

A free email class for Python developers

realpython.com



[Remove ads](#)

FastAPI

[FastAPI](#) is a Python web framework that’s optimized for building APIs. It uses [Python type hints](#) and has built-in support for [async operations](#). FastAPI is built on top of [Starlette](#) and [Pydantic](#) and is very performant.

Below is an example of the REST API built with FastAPI:

Python

```
# app.py
from fastapi import FastAPI
from pydantic import BaseModel, Field

app = FastAPI()

def _find_next_id():
    return max(country.country_id for country in countries) + 1

class Country(BaseModel):
    country_id: int = Field(default_factory=_find_next_id, alias="id")
    name: str
    capital: str
    area: int

countries = [
    Country(id=1, name="Thailand", capital="Bangkok", area=513120),
    Country(id=2, name="Australia", capital="Canberra", area=7617930),
    Country(id=3, name="Egypt", capital="Cairo", area=1010408),
]

@app.get("/countries")
async def get_countries():
    return countries

@app.post("/countries", status_code=201)
async def add_country(country: Country):
    countries.append(country)
    return country
```

This application uses the features of FastAPI to build a REST API for the same country data you’ve seen in the other examples.

You can try this application by installing `fastapi` with `pip`:

Shell



```
$ python -m pip install fastapi
```

You’ll also need to install `uvicorn[standard]`, a server that can run FastAPI applications:

Shell



```
$ python -m pip install uvicorn[standard]
```

If you’ve installed both `fastapi` and `uvicorn`, then save the code above in a file called `app.py`. Run the following command to start up a development server:

Shell



```
$ uvicorn app:app --reload
INFO: Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
```

The server is now running. Open up a browser and go to `http://127.0.0.1:8000/countries`. You’ll see FastAPI respond with this:

JSON


```
[
  {
    "id": 1,
    "name": "Thailand",
    "capital": "Bangkok",
    "area": 513120
  },
  {
    "id": 2,
    "name": "Australia",
    "capital": "Canberra",
    "area": 7617930
  },
  {
    "id": 3,
    "name": "Egypt",
    "capital": "Cairo",
    "area": 1010408
  }
]
```

FastAPI responds with a JSON array containing a list of countries. You can also add a new country by sending a POST request to `/countries`:

Shell



```
$ curl -i http://127.0.0.1:8000/countries \
-X POST \
-H 'Content-Type: application/json' \
-d '{"name": "Germany", "capital": "Berlin", "area": 357022}' \
-w '\n'

HTTP/1.1 201 Created
content-type: application/json
...

{"id":4,"name":"Germany","capital":"Berlin","area": 357022}
```

You added a new country. You can confirm this with GET `/countries`:

Shell



```
$ curl -i http://127.0.0.1:8000/countries -w '\n'

HTTP/1.1 200 OK
content-type: application/json
...

[
  {
    "id":1,
    "name":"Thailand",
    "capital":"Bangkok",
    "area":513120,
  },
  {
    "id":2,
    "name":"Australia",
    "capital":"Canberra",
    "area":7617930
  },
  {
    "id":3,
    "name":"Egypt",
    "capital":"Cairo",
    "area":1010408
  },
  {
    "id":4,
    "name": "Germany",
    "capital": "Berlin",
    "area": 357022
  }
]
```

FastAPI returns a JSON list including the new country you just added.

You'll notice that the FastAPI application looks similar to the Flask application. Like Flask, FastAPI has a focused feature set. It doesn't try to handle all aspects of web application development. It's designed to build APIs with modern Python features.

If you look near the top of `app.py`, then you'll see a class called `Country` that extends `BaseModel`. The `Country` class describes the structure of the data in the REST API:

Python

```
class Country(BaseModel):
    country_id: int = Field(default_factory=_find_next_id, alias="id")
    name: str
    capital: str
    area: int
```

This is an example of a [Pydantic model](#). Pydantic models provide some helpful features in FastAPI. They use Python type annotations to enforce the data type for each field in the class. This allows FastAPI to automatically generate JSON, with the correct data types, for API endpoints. It also allows FastAPI to validate incoming JSON.

It's helpful to highlight the first line as there's a lot going on there:

Python

```
country_id: int = Field(default_factory=_find_next_id, alias="id")
```

In this line, you see `country_id`, which stores an [integer](#) for the ID of the Country. It uses the [Field function](#) from Pydantic to modify the behavior of `country_id`. In this example, you're passing `Field` the keyword arguments `default_factory` and `alias`.

The first argument, `default_factory`, is set to `_find_next_id()`. This argument specifies a function to run whenever a new `Country` is created. The return value will be assigned to `country_id`.

The second argument, `alias`, is set to `id`. This tells FastAPI to output the key `"id"` instead of `"country_id"` in the JSON:

JSON

```
{
  "id":1,
  "name":"Thailand",
  "capital":"Bangkok",
  "area":513120,
},
```

This alias also means you can use `id` when you create a new `Country`. You can see this in the `countries` list:

Python

```
countries = [
    Country(id=1, name="Thailand", capital="Bangkok", area=513120),
    Country(id=2, name="Australia", capital="Canberra", area=7617930),
    Country(id=3, name="Egypt", capital="Cairo", area=1010408),
]
```

This list contains three instances of `Country` for the initial countries in the API. Pydantic models provide some great features and allow FastAPI to easily process JSON data.

Now take a look at the two API functions in this application. The first, `get_countries()`, returns a list of countries for GET requests to `/countries`:

Python

```
@app.get("/countries")
async def get_countries():
    return countries
```

FastAPI will automatically create JSON based on the fields in the Pydantic model and set the right JSON data type from the Python type hints.

The Pydantic model also provides a benefit when you make a POST request to `/countries`. You can see in the second API function below that the parameter `country` has a `Country` annotation:

Python

```
@app.post("/countries", status_code=201)
async def add_country(country: Country):
    countries.append(country)
    return country
```

This type annotation tells FastAPI to validate the incoming JSON against `Country`. If it doesn't match, then FastAPI will return an error. You can try this out by making a request with JSON that doesn't match the Pydantic model:

Shell



```
$ curl -i http://127.0.0.1:8000/countries \
-X POST \
-H 'Content-Type: application/json' \
-d '{"name":"Germany", "capital": "Berlin"}' \
-w '\n'

HTTP/1.1 422 Unprocessable Entity
content-type: application/json
...

{
  "detail": [
    {
      "loc":["body","area"],
      "msg":"field required",
      "type":"value_error.missing"
    }
  ]
}
```

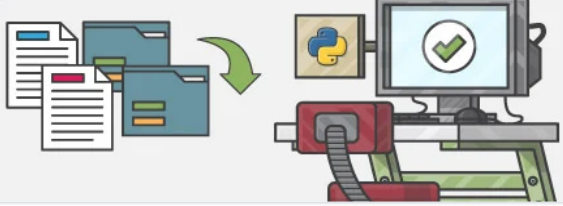
The JSON in this request was missing a value for area, so FastAPI returned a response with the status code 422 Unprocessable Entity as well as details about the error. This validation is made possible by the Pydantic model.


This example only scratches the surface of what FastAPI can do. With its high performance and modern features like async functions and automatic documentation, FastAPI is worth considering for your next REST API.

Free PDF Download: Python 3 Cheat Sheet

Download Now

realpython.com



 [Remove ads](#)


Conclusion

REST APIs are everywhere. Knowing how to leverage Python to consume and build APIs allows you to work with the vast amount of data that web services provide.

In this tutorial, you’ve learned how to:

- Identify the **REST architecture** style
- Work with **HTTP methods** and status codes
- Use **requests** to get and consume data from an external API
- Define **endpoints, data,** and **responses** for a REST API
- Get started with Python tools to build a **REST API**

Using your new Python REST API skills, you’ll be able to not only interact with web services but also build REST APIs for your applications. These tools open the door to a wide range of interesting, data-driven applications and services.

 **Take the Quiz:** Test your knowledge with our interactive “Python and REST APIs: Interacting With Web Services” quiz. Upon completion you will receive a score so you can track your learning progress over time:


Take the Quiz »

Mark as Completed







 Python Tricks 