Hello world wide web!

View on GitHub

Modal makes it easy to turn your Python functions into serverless web services: access them via a browser or call them from any client that speaks HTTP, all without having to worry about setting up servers or managing infrastructure.

This tutorial shows the path with the shortest "time to 200": `modal.web_endpoint` .

On Modal, web endpoints have all the superpowers of Modal Functions: they can be accelerated with GPUs, they can access Secrets or Volumes, and they automatically scale to handle more traffic.

Under the hood, we use the FastAPI library, which has high-quality documentation, linked throughout this tutorial.

Turn a Modal Function into an endpoint with a single decorator

Modal Functions are already accessible remotely — when you add the `@app.function` decorator to a Python function and run `modal deploy` , you make it possible for your other Python functions to call it.

That's great, but it's not much help if you want to share what you've written with someone running code in a different language — or not running code at all!

And that's where most of the power of the Internet comes from: sharing information and functionality across different computer systems.

So we provide the `web_endpoint` decorator to wrap your Modal Functions in the lingua franca of the web: HTTP. Here's what that looks like:

```
import modal

app = modal.App(name="example-lifecycle-web")


@app.function()
@modal.web_endpoint(
    docs=True  # adds interactive documentation in the browser
)
def hello():
    return "Hello world!"
```

You can turn this function into a web endpoint by running `modal serve basic_web.py` . In the output, you should see a URL that ends with `hello-dev.modal.run` . If you navigate to this URL, you should see the `"Hello world!"` message appear in your browser.

You can also find interactive documentation, powered by OpenAPI and Swagger, if you add `/docs` to the end of the URL. From this documentation, you can interact with your endpoint, sending HTTP requests and receiving HTTP responses. For more details, see the FastAPI documentation.

By running the endpoint with `modal serve` , you created a temporary endpoint that will disappear if you interrupt your terminal. These temporary endpoints are great for debugging — when you save a change to any of your dependent files, the endpoint will redeploy. Try changing the message to something else, hitting save, and then hitting refresh in your browser or re-sending the request from `/docs` or the command line. You should see the new message, along with logs in your terminal showing the redeploy and the request.

When you're ready to deploy this endpoint permanently, run `modal deploy basic_web.py` . Now, your function will be available even when you've closed your terminal or turned off your computer.

Send data to a web endpoint

The web endpoint above was a bit silly: it always returns the same message.

Most endpoints need an input to be useful. There are two ways to send data to a web endpoint:

- in the URL as a query parameter
- in the body of the request as JSON

Sending data in query parameters

By default, your function's arguments are treated as query parameters: they are extracted from the end of the URL, where they should be added in the form `?arg1=foo&arg2=bar` .

From the Python side, there's hardly anything to do:

```python
@app.function()
@modal.web_endpoint(docs=True)
def greet(user: str) -> str:
    return f"Hello {user}!"
```

If you are already running `modal serve basic_web.py` , this endpoint will be available at a URL, printed in your terminal, that ends with `greet-dev.modal.run` .

We provide Python type-hints to get type information in the docs and automatic validation. For example, if you navigate directly to the URL for `greet` , you will get a detailed error message indicating that the `user` parameter is missing. Navigate instead to `/docs` to see how to invoke the endpoint properly.

You can read more about query parameters in the FastAPI documentation.

Sending data in the request body

For larger and more complex data, it is generally preferrable to send data in the body of the HTTP request. This body is formatted as JSON, the most common data interchange format on the web.

To set up an endpoint that accepts JSON data, add an argument with a `dict` type-hint to your function. This argument will be populated with the data sent in the request body.

```python
@app.function()
@modal.web_endpoint(method="POST", docs=True)
def goodbye(data: dict) -> str:
    name = data.get("name") or "world"
    return f"Goodbye {name}!"
```

Note that we gave a value of `"POST"` for the `method` argument here. This argument defines the HTTP request method that the endpoint will respond to, and it defaults to `"GET"` . If you head to the URL for the `goodbye` endpoint in your browser, you will get a 405 Method Not Allowed error, because browsers only send GET requests by default. While this is technically a separate concern from query parameters versus request bodies and you can define an endpoint that accepts GET requests and uses data from the body, it is considered bad form.

Navigate to `/docs` for more on how to invoke the endpoint properly. You will need to send a POST request with a JSON body containing a `name` key. To get the same typing and validation benefits as with query parameters, use a Pydantic model for this argument.

You can read more about request bodies in the FastAPI documentation.

Handle expensive startup with `modal.Cls`

Sometimes your endpoint needs to do something before it can handle its first request, like get a value from a database or set the value of a variable. If that step is expensive, like loading a large ML model, it'd be a shame to have to do it every time a request comes in!

Web endpoints can be methods on a `modal.Cls`. Note that they don't need the `modal.method` decorator.

This example will only set the `start_time` instance variable once, on container startup.

```python
@app.cls()
class WebApp:
    @modal.enter()
    def startup(self):
        from datetime import datetime, timezone

        print("🏁 Starting up!")
        self.start_time = datetime.now(timezone.utc)

    @modal.web_endpoint(docs=True)
    def web(self):
        from datetime import datetime, timezone

        current_time = datetime.now(timezone.utc)
        return {"start_time": self.start_time, "current_time": current_time}
```

What next?

Modal's `web_endpoint` decorator is opinionated and designed for relatively simple web applications — one or a few independent Python functions that you want to expose to the web.

Three additional decorators allow you to serve more complex web applications with greater control:

- `asgi_app` to serve applications compliant with the ASGI standard, like FastAPI
- `wsgi_app` to serve applications compliant with the WSGI standard, like Flask
- `web_server` to serve any application that listens on a port