Flask-Jeroboam

Release 0.1.0b3

Jean-Christophe Bianic

Nov 26, 2023

CONTENTS

Welcome to **Flask-Jeroboam**'s documentation.

Flask-Jeroboam is a Flask extension modelled after FastAPI. Like the former, it uses Pydantic to provide easy-to-configure data validation in request parsing and response serialization, as well as OpenAPI-compliant documentation auto-generation.

Start with *Installation*, then jump right in with our *Getting Started Guide*. Next, the *In-depth Features Tour* dive deep into how to use the extension, while the *Tutorial* walks you through a comprehensive example. Finally, the *API* section gives you details on the extension's internals.

Note: This documentation assumes a certain familiarity with Flask and Pydantic. If you're new to either, please refer to their respective documentation. They are both fantastic.

- Flask documentation
- Pydantic documentation

CHAPTER

ONE

USER'S GUIDE

This guide will walk you through how to use Flask-Jeroboam.

1.1 Installation

1.1.1 Install Flask-Jeroboam

I publish **Flask-Jeroboam** to PyPI, the Python Package Index, and as such, it is easily installable using one of the following commands, depending and your tooling for dependency management:

poetry

```
$ poetry add flask-jeroboam
```

pip

```
$ pip install flask-jeroboam
```

With that command, you have installed **Flask-Jeroboam** with its two direct dependencies, Flask and Pydantic and their own dependencies tree (*check it out here*).

Note: We highly recommend installing **Flask-Jeroboam** in a virtual environment. If you need directions on how to do that check out the *Setting Things Up* section of our tutorial for more information.

1.1.2 Dependencies

Installing **Flask-Jeroboam** will automatically install these packages along with their dependencies:

- Flask the web framework heavy lifting is still performed by Flask.
- Pydantic to provide data validation using Python type annotations.

These two direct dependencies come with their own dependencies tree. In total, you will have up to 9 new packages installed. There is a nice poetry command to explore that tree. It goes like this:

```
$ poetry show flask-jeroboam --tree
flask-jeroboam 0.1.0b0 A Flask extension, inspired by FastAPI that uses Pydantic to␣
↪provide easy-to-configure data validation for request parsing and response␣
↪serialization.
```

(continues on next page)

(continued from previous page)

```
├── flask >=2.1.3,<3.0.0
│   ├── click >=8.0
│   │   └── colorama *
│   ├── itsdangerous >=2.0
│   ├── jinja2 >=3.0
│   │   └── markupsafe >=2.0
│   └── werkzeug >=2.2.2
│       └── markupsafe >=2.1.1 (circular dependency aborted here)
└── pydantic >=1.10.2,<2.0.0
    └── typing-extensions >=4.2.0
```

1.1.3 Testing your installation

Let's make sure you set up everything correctly. Create and open a simple file at the root of your project: `app.py`.

```python
from flask_jeroboam import Jeroboam


app = Jeroboam(__name__)


@app.get("/ping")
def ping():
    return "pong"


if __name__ == "__main__":
    app.run(port=5000)
```

Running this file should start a server on `localhost:5000`. You can hit that endpoint with the command `curl 'http://localhost:5000/ping'` or directly in your browser by going to http://localhost:5000/ping. If either answer with "pong", your installation is functional, and you are ready to jump to our *Getting Started Guide*.

1.1.4 Uninstall Flask-Jeroboam

Removing **Flask-Jeroboam** from your project's dependencies is as straightforward as adding it to your project:

poetry

```
$ poetry remove flask-jeroboam
```

pip

```
$ pip uninstall flask-jeroboam
```

1.2 Getting Started

Let's walk you through a simple example step by step.

In this guide, we will:

- create a Jeroboam app
- register a view function
- add some view argument configuration to it for inbound data validation
- add a response_model to the endpoint for outbound data validation
- register the OpenAPI blueprint to get a look at the generated documentation

1.2.1 Create a Jeroboam App

Let's start with creating the application object.

```
1 from flask_jeroboam import Jeroboam
2
3
4 app = Jeroboam("Jeroboam Getting Started App")
5 app.init_app()
6
7
8 if __name__ == "__main__":
9     app.run(host="localhost", port=5000)
```

As you can see, there is nothing special about the app creation on line 4. The **Jeroboam** class from flask_jeroboam subclasses flask's Flask application object, and you can use it as a drop-in replacement of the former.

```
1 from flask_jeroboam import Jeroboam
2
3
4 app = Jeroboam("Jeroboam Getting Started App")
5 app.init_app()
6
7
8 if __name__ == "__main__":
9     app.run(host="localhost", port=5000)
```

On line 5, we are calling the init_app method of the app instance. You should call this method after loading the configuration to your app: it will register OpenAPI blueprints and generic error handlers. You can always opt-out of these with appropriate configuration values (see *here*).

```
1 from flask_jeroboam import Jeroboam
2
3
4 app = Jeroboam("Jeroboam Getting Started App")
5 app.init_app()
6
7
8 if __name__ == "__main__":
9     app.run(host="localhost", port=5000)
```

Finally, lines 8 and 9 are a convenient way to start the app by running the file directly.

Note: The application factory pattern is usually a good practice [1] and should be followed when you start an actual project.

1.2.2 Register a view function

Registering a view function means binding a python function to an URL. Whenever a request sent to your server matches the rule you defined, the registered function, called a view function, will be run.

You can register a view function in several ways in Flask. The preferred way to do it in **Flask_Jeroboam** is to use method decorators, like on line 8 in the example below:

```
1  from flask_jeroboam import Jeroboam
2
3
4  app = Jeroboam("Jeroboam Getting Started App")
5  app.init_app()
6
7
8  @app.get("/health")
9  def get_health():
10     return {"status": "Ok"}
11
12 if __name__ == "__main__":
13     app.run(host="localhost", port=5000)
```

Here we are telling the app instance that when it receives an incoming GET Request to the URL `/health`, it should call the `get_health` function and return the result to the client. Let's try it. Run your file and start poking.

```
$ curl http://localhost:5000/health
{"status": "ok"}
```

Note: Although you can register view functions directly on the app instance, any project beyond the size of a classical tutorial will benefit from using the modularity of Blueprints [2], and you will find yourself using blueprints more than your app instance. The good news is that you register view functions on blueprints as you do on the app instance.

1.2.3 Adding View Arguments

Let's try something more interesting. So far, our Jeroboam application behaves like a regular Flask application.

Let's register a view function that takes parameters. On line 13, you will find the method decorator we saw in the previous section. But on line 14, the view function takes two parameters with type hints and default values. It then returns them without modifying them.

```
1  from flask_jeroboam import Jeroboam
2
3
4  app = Jeroboam("Jeroboam Getting Started App")
5  app.init_app()
```

(continues on next page)

(continued from previous page)

```python


@app.get("/health")
def get_health():
    return {"status": "Ok"}


@app.get("/query_string_arguments")
def get_query_string_arguments(page: int = 1, per_page: int = 10):
    return {"page": page, "per_page": per_page}


if __name__ == "__main__":
    app.run(host="localhost", port=5000)
```

This view function 's only purpose is to help us inspecting the values the function actually receives when it is called and this is precisely what we will do.

```
$ curl 'http://localhost:5000/query_string_arguments'
{"page":1,"per_page":10}
```

So far, so good. The result was predictable. The function received the default values for the parameters. Let's try something else.

```
$ curl 'http://localhost:5000/query_string_arguments?page=2&per_page=50'
{"page":2,"per_page":50}
```

Let's take a closer look at the url we're hitting: `/query_string_arguments?page=2&per_page=50`. The part after the `?` is called a query string. It is a way to pass parameters through a URL. `page=2&per_page=50` translates to two parameters, `page` and `per_page` with respective values of `2` and `50`. Luckily that's exactly what our view function is expecting. **Flask-Jeroboam** will parse the query string, validate the values (check they can be cast as int) and inject them into the view function.

The previous example showed us the parsing and injecting part. Let's take a look at its validation capabilities by passing a page value that can't be cast to an int. We will add the `-w 'Status Code: %{http_code}\n'` option to our curl command to print the status code of the response.

```
$ curl -w 'Status Code: %{http_code}\n' 'http://localhost:5000/query_string_arguments?
↪page=not_a_int&per_page=50'
{"detail":[{"loc":["query","page"],"msg":"value is not a valid integer","type":"type_
↪error.integer"}]}
Status Code: 400
```

Here, we got a 400 Bad Request response, with details about the error, telling us that the value of the `page` argument located in the `query` (*"loc":["query","page"]*) is not a valid integer (*"msg":"value is not a valid integer"*).

Note: By default, the arguments of a `GET` view function are expected to get their values from the query string. It is their `location`. You can explicitly set the location of the arguments by using argument functions as default values (Exemple: `page: int = Query(1)`). Possible location for parameters include `Path`, `Query`, `Header` and `Cookie`. For request bodies, you can set locations to `Body`, `Form` and `File`. `Body` is the default location for `POST` and `PUT` requests.

You will find more information about this mechanics *here*.

Now that we have covered the basics of inbound handling, let's look at the outbound parsing and validation. This is done by adding a `response_model` to our decorator.

1.2.4 Response Models

We start by defining a Pydantic BaseModel for our response. This model will be used to validate the outbound data of our view function. We first import BaseModel and Field from pydantic on line 1 and 2. On line 11-14, we define a subclass of pydnatic's BaseModel named `Item` with three fields: `name`, `price` and `count`. The `name` field is a string, the `price` field is a float and the `count` field is an int with a default value of 1.

```
1  from pydantic import BaseModel
2  from pydantic import Field
3
4  from flask_jeroboam import Jeroboam
5
6
7  app = Jeroboam("Jeroboam Getting Started App")
8  app.init_app()
9
10
11 class Item(BaseModel):
12     name: str
13     price: float
14     count: int = Field(1)
15
16
17 @app.get("/item", response_model=Item)
18 def get_an_item():
19     return {"name": "Bottle", "price": 5}
20
21
22 if __name__ == "__main__":
23     app.run(host="localhost", port=5000)
```

We then pass the `Item` model as the `response_model` argument of the `@app.get` decorator on line 17. Our view function's purpose is to demonstrate that our return value will be processed through the `Item` model and not simply returning the `{"name": "Bottle", "price": 5}` dictionnary, casting the price into a float, and adding a default value of 1 to the count field.

```
1  from pydantic import BaseModel
2  from pydantic import Field
3
4  from flask_jeroboam import Jeroboam
5
6
7  app = Jeroboam("Jeroboam Getting Started App")
8  app.init_app()
9
10
11 class Item(BaseModel):
12     name: str
13     price: float
14     count: int = Field(1)
```

(continues on next page)

(continued from previous page)

```
15
16
17 @app.get("/item", response_model=Item)
18 def get_an_item():
19     return {"name": "Bottle", "price": 5}
20
21
22 if __name__ == "__main__":
23     app.run(host="localhost", port=5000)
```

Let's try it out.

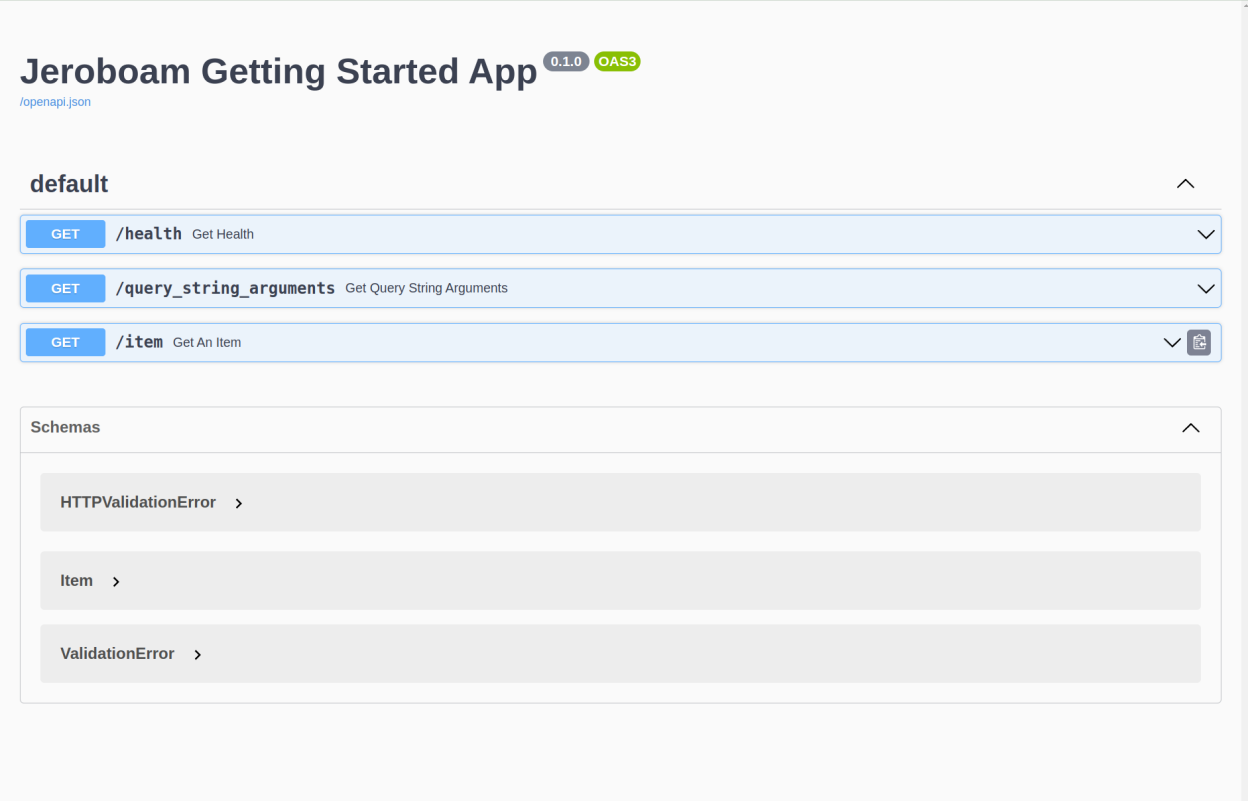
```
$ curl 'http://localhost:5000/item'
{"name": "Bottle", "price": 5.0, "count": 1}%
```

What happened is that the return value of the view function has been fed to the `Item` model. The price have been casted as a float, and the missing key-value of count has been added with its default value. The values have been validated and finally serialized into a JSON string.

Finally, to wrap up this first tour of **Flask-Jeroboam**, let's take a look at the OpenAPI-complaint documentation our app was able to produce.

1.2.5 OpenAPI Documentation

When you visit http://localhost:5000/docs in your browser you should see the OpenAPI documentation for your API. It will look something like this:



1.2.6 Wrapping Up

In this page, we covered the three main features of **Flask-Jeroboam**:

- Inbound data parsing and validation based on view function signatures
- Outbound data validation and serialization based on response models
- OpenAPI auto-documentation

1.2.7 To go further

If you want to learn more, you can check out our *in-depth features tour*.

We also mentioned:

- Flask's App Factories
- Flask's Modular App with Blueprints

Complete example code for this page can be found here. Examples are tested in this file.

1.3 In-Depth Features Tour

In this part of the documentation, we will cover **Flask-Jeroboam**'s features in-depth to give you a solid understanding of how the package works and how to best use it.

Flask-Jeroboam's purpose is to let you focus on your web app business logic by providing a way to easily define your endpoints inbound and outbound interfaces. The inbound interface defines how clients can interact with your endpoints, while the outbound interface defines what clients can expect from them.

1.3.1 Defining endpoints' inbound interface with view function arguments

You define the parsing, validation, and injection of inbound data into your views functions simply through their arguments, using a combination of type hints, default values, and sensible implicit values to make it as concise as possible.

Learn *how to define your endpoints' inbound interface* to make them more concise and robust.

Inbound interface & view function arguments

We will focus here on how to use your view functions arguments to define an endpoint's inbound interface. By inbound interface, we refer to the data a client can send to the server and how.

With **Flask-Jeroboam**, you use a combination of type hints and default values on your view function's arguments to define the inbound interface of your endpoints. This feature enables you to delegate the parsing, validation, and injection of inbound data into your views functions to the extension and focus on your endpoints' business logic.

Flask-Jeroboam will handle inbound data based on their location, type and additional validation options. The location refers to how the client passes data over an incoming HTTP request (e.g. query strings, headers, request body…). The type defines the expected shape or schema of the incoming data. Additionally, you can specify extra validation options.

Location

The location concept refers to how the client passes data over an incoming HTTP request. First, we will look into the seven possible locations. Then we will look at the implicit location **Flask-Jeroboam** will assume, and finally at how you can explicitly define the location of every argument of your view functions.

What are locations

There are various ways to pass data with an HTTP request. Flask and Werkzeug are already parsing the request's raw incoming data to populate members of the `request` object accordingly. Defining the location of an argument is a way to tell **Flask-Jeroboam** which `request`'s member to use to retrieve the incoming data and inject them into your view functions.

There are seven possible locations for view functions arguments.

- Four parameters:
  - `PATH`: Path parameters are the dynamic parts of an URL found before any `?` separator (e.g. `/items/12`). They are typically used to pass ids. Flask already injects them into your view function.
  - `QUERY`: Query Strings are equal-sign-separated key-value pairs found in the part of an URL after the `?` separator (e.g. `?page=1`). They serve a variety of purposes. We retrived them from Flask's `request.args`
  - `HEADER`: Header parameters are fields destined to pass additional context or metadata about the request to the server. They are colon-separated key-value pair like this `X-My-Header: 42`. We retrived them ffrom Flask's `request.headers`
  - `COOKIE`: Cookies are stored on the client side to keep track of a state in a stateless protocol. They look like this `Cookie: username=john`, and we retrived them from Flask's `request.cookies`
- Three variations of the request body:
  - `BODY`: The request body of an HTTP request is the optional content of the request. Request fetching resources[1] usually don't have one. The request body has a content-type property (like: `application/json`) that gives the server indication on how to parse them. We retrieve the reuqest body from Flask's `request.data` (or `request.json` when its mimetype is `application/json`).
  - `FORM`: A Request Body with a content-type value of `application/x-www-form-urlencoded`. We retrieved data from Flask's `request.form`
  - `FILE`: Request Body with a enctype of `multipart/form-data` and We retrieved data from Flask's `request.files`

Solving Location

Most of the time, you won't have to explicitly define your arguments' location, thanks to **Flask-Jeroboam**'s implicit location mechanism. However, you also can explicitly define the location for each argument of your view functions by using special functions on your arguments' default values.

[1] Like `GET`, `OPTIONS`, `HEAD`

Implicit Location

We can boiled down **Flask-Jeroboam**'s heuristic to determine the location of an argument to a few simple rules:

- the argument's name will be checked against the the path parameters' names from the endpoint's rule
- arguments of ressource fetching verbs are assumed to be `QUERY` parameters
- arguments of ressource creation verbs are assumed to be `BODY` parameters

> **Warning:** This is slighly different from **FastAPI**'s heuristic, where singular values are assumed to be `QUERY` parameters no matter the verb, and pydantic Models are assumed to be `BODY` parameters.

Apart from Path parameters, **Flask-Jeroboam** derives the implicit location of an argument from the HTTP verb of your view function, based on the assumption that for a `GET` request, the client typically pass parameters through the query string and that for `PUT` and `POST` requests the client will mainly use the request body.

So if you look at these view functions on line 9 and 14, without any explicit location definition, **Flask-Jeroboam** will assume a `QUERY` location for the `GET` endpoint and `BODY` for the `POST` endpoint.

```
1  from flask_jeroboam import Jeroboam
2
3
4  app = Jeroboam("Jeroboam Inbound Features App")
5  app.init_app()
6
7
8  @app.get("/implicit_location_is_query_string")
9  def get_implicit_argument(page: int):
10     return f"Received Page Argument is : {page}"
11
12
13 @app.post("/implicit_location_is_body")
14 def post_implicit_argument(page: int):
15     return f"Received Page Argument is : {page}"
16
17
18 if __name__ == "__main__":
19     app.run(host="localhost", port=5000)
```

If you run the above file, you can test it out. The `/implicit_location_is_query_string` endpoint will expect a page parameter in the query string.

```
$ curl 'localhost:5000/implicit_location_is_query_string?page=42'
Received Page Argument is : 42
```

while the `/implicit_location_is_body` endpoint will expect a page field in the request body.

```
$ curl -X POST 'localhost:5000/implicit_location_is_body' -d '{"page": 42}' -H "Content-
↪Type: application/json"
Received Page Argument is : 42
```

Although the two view functions received the same parameter values, notice that we build our request differently by hosting the parameters in two different locations.

In addition to this verb-based mechanism, **Flask-Jeroboam** will automatically detect Path parameters. In the following example, **Flask-Jeroboam** will be recognized the argument `id` as a Path Parameter. Indeed, it is first declared on line 8 with the `<int:id>` part of the rule, so when **Flask-Jeroboam** comes across an argument in the decorated function with the same name but without any explicit location definition, it will safely assume that this is a Path Parameter.

```
1  from flask_jeroboam import Jeroboam
2
3
4  app = Jeroboam("Jeroboam Inbound Features App")
5  app.init_app()
6
7
8  @app.get("/item/<int:id>/implicit")
9  def get_path_parameter(id: int):
10     return f"Received id Argument is : {id}"
11
12
13 if __name__ == "__main__":
14     app.run(host="localhost", port=5000)
```

You can test it out:

```
$ curl 'localhost:5000/item/42/implicit'
Received id Argument is : 42
```

It also works with other HTTP verbs and overrides the verb-based location. **Flask-Jeroboam** will also recognized the argument `id` as a Path Parameter in the following example.

```
1  from flask_jeroboam import Jeroboam
2
3
4  app = Jeroboam("Jeroboam Inbound Features App")
5  app.init_app()
6
7
8  @app.post("/item/<int:id>/implicit")
9  def post_path_parameter(id: int):
10     return f"Received id Argument is : {id}"
11
12
13 if __name__ == "__main__":
14     app.run(host="localhost", port=5000)
```

```
$ curl -X POST 'localhost:5000/item/42/implicit'
Received id Argument is : 42
```

Note: This implicit location mechanism is one of the reasons why the method decorator (@app.get/put/post VS @app.route) is the preferred way to register a view function in **Flask-Jeroboam**. It enforces the good practice of having a single HTTP verb per view function. View functions assigned to more than one HTTP verb tends to be split up in two mostly independent branches, which depletes their readability.

Although the implicit location will cover most of the cases, you can also define them explicitly.

Explicit Locations

To define explicit locations, you must use one of **Flask-Jeroboam**'s special functions (`Path`, `Query`, `Cookie`, `Header`, `Body`, `Form` or `File`) to assign default values to your arguments.

For example, these two endpoints will behave the same way, line 10 (implicit) and 15 (explicit) are equivalent:

```
1  from flask_jeroboam import Jeroboam
2  from flask_jeroboam import Query
3
4
5  app = Jeroboam("Jeroboam Inbound Features App")
6  app.init_app()
7
8
9  @app.get("/implicit_location_is_query_string")
10 def get_implicit_argument(page: int):
11     return f"Received Page Argument is : {page}"
12
13
14 @app.get("/explicit_location_is_query_string")
15 def get_explicit_argument(page: int = Query()):
16     return f"Received Page Argument is : {page}"
17
18
19 if __name__ == "__main__":
20     app.run(host="localhost", port=5000)
```

And same goes for `POST` (or `PUT`) view functions. Line 10 and 15 are equivalent.

```
1  from flask_jeroboam import Body
2  from flask_jeroboam import Jeroboam
3
4
5  app = Jeroboam("Jeroboam Inbound Features App")
6  app.init_app()
7
8
9  @app.post("/implicit_location_is_body")
10 def post_implicit_argument(page: int):
11     return f"Received Page Argument is : {page}"
12
13
14 @app.post("/explicit_location_is_body")
15 def post_explicit_argument(page: int = Body()):
16     return f"Received Page Argument is : {page}"
17
18
19 if __name__ == "__main__":
20     app.run(host="localhost", port=5000)
```

Let's test it out.

```
$ curl 'localhost:5000/implicit_location_is_query_string?page=42'
Received Page Argument is : 42
$ curl 'localhost:5000/explicit_location_is_query_string?page=42'
Received Page Argument is : 42
$ curl -X POST 'localhost:5000/implicit_location_is_body' -d '{"page": 42}' -H "Content-
↪Type: application/json"
Received Page Argument is : 42
$ curl -X POST 'localhost:5000/explicit_location_is_body' -d '{"page": 42}' -H "Content-
↪Type: application/json"
Received Page Argument is : 42
```

You can also point to another location than the default one, and define different locations for each argument and mix implicit locations with explicit locations. In the following example, we define an explicit `Cookie` location for the argument `username`. On line 11, it shares the signature with another explicitly-query-located page argument, but on line 16 we define a similar view function in which page is implicitly located.

```
from flask_jeroboam import Cookie
from flask_jeroboam import Jeroboam
from flask_jeroboam import Query


app = Jeroboam("Jeroboam Inbound Features App")
app.init_app()


@app.get("/explicit_location_is_query_string_and_cookie")
def get_explicit_arguments(page: int = Query(), username: str = Cookie()):
    return f"Received Page Argument is : {page}. Username is : {username}"


@app.get("/implicit_and_explicit")
def get_implicit_and_explicit_arguments(page: int, username: str = Cookie()):
    return f"Received Page Argument is : {page}. Username is : {username}"


if __name__ == "__main__":
    app.run(host="localhost", port=5000)
```

Let's test it out.

```
$ curl 'localhost:5000/explicit_location_is_query_string_and_cookie?page=42' --cookie
↪"username=john"
Received Page Argument is : 42. Username is : john
$ curl 'localhost:5000/implicit_and_explicit?page=42' --cookie "username=john"
Received Page Argument is : 42. Username is : john
```

Note: Linters like Flake8 will likely complain about making a function call in an argument default. While this is good advice, it won't cause any unwanted effect in this particular case. You should consider disabling `B008` warnings for the files in which you define your view functions.

Assigning default values

As you may have guessed, the special functions are highjacking the default value mechanism to let you easily define an explicit location for your arguments. As a result, their returned value won't be used as a fallback when the client don't provide any argument. In fact, so far, all arguments we have defined are implicitly required because they have no default values to fall back to when the request does not provided them.

Don't take my word for it, let's try it out on the previously defined `/implicit_location_is_query_string` endpoint.

```
$ curl -w 'Status Code: %{http_code}\n' 'localhost:5000/implicit_location_is_query_string
↪'
{"detail":[{"loc":["query","page"],"msg":"field required","type":"value_error.missing"}]}
Status Code: 400
```

We received a 400 Bad Request response because we did not provide the required parameter page in our query string. What if we want to define our default value for the page parameter to 1 ? They are two ways to go about it:

- If you go with the implicit location, you can define a default value as you normally would, as shown on line 10.
- If you use an explicit definition, you must pass the default value as the first argument of your function call, like in line 15.

```
1  from flask_jeroboam import Jeroboam
2  from flask_jeroboam import Query
3
4
5  app = Jeroboam("Jeroboam Inbound Features App")
6  app.init_app()
7
8
9  @app.get("/implicit_location_with_default_value")
10 def get_implicit_argument_with_default(page: int = 1):
11     return f"Received Page Argument is : {page}"
12
13
14 @app.get("/explicit_location_with_default_value")
15 def get_explicit_argument_with_default(page: int = Query(1)):
16     return f"Received Page Argument is : {page}"
17
18
19 if __name__ == "__main__":
20     app.run(host="localhost", port=5000)
```

Let's test it out.

```
1  $ curl 'localhost:5000/implicit_location_with_default_value'
2  Received Page Argument is : 1
3  $ curl 'localhost:5000/implicit_location_with_default_value?page=42'
4  Received Page Argument is : 42
5  $ curl 'localhost:5000/explicit_location_with_default_value'
6  Received Page Argument is : 1
7  $ curl 'localhost:5000/explicit_location_with_default_value?page=42'
8  Received Page Argument is : 42
```

The default value is correctly inserted when you don't provide the parameter in the query string for either endpoint.

The server returns a valid response meaning the page parameter is no longer required. We also check that the parsing-validation-injection mechanism is still working on lines 3 and 7.

The special functions also provide a way to define additional validation options, but first, let's take a closer look at defining the second part of our inbound arguments: the type.

Type

The type refers to the shape or schema of the data you expect. You can assign a type to an argument using type hints. Types can be python built-in types (e.g. `str`, `int`, `float`), pydantic's `BaseModel` subclasses or a container of the previous two (e.g. `List[str]`)

In addition to using type hints, you can also use the first argument of special functions like `Query`.

Let's look at an example. First, on lines 12 to 14, we define the `Item` class inheriting from pydantic's `BaseModel`.

```
1  from typing import List
2
3  from pydantic import BaseModel
4  from flask_jeroboam import Jeroboam
5  from flask_jeroboam import Query
6
7
8  app = Jeroboam("Jeroboam Inbound Features App")
9  app.init_app()
10
11
12 class Item(BaseModel):
13     name: str
14     count: int
15
16
17 @app.get("/defining_type_with_type_hints")
18 def get_type_hints(page: int, search: List[str], item: Item, price=Query(float)):
19     return (
20         f"Received arguments are :\n"
21         f"page : {page}\n"
22         f"search : {search}\n"
23         f"price : {price}\n"
24         f"item : {item}"
25     )
26
27
28 if __name__ == "__main__":
29     app.run(host="localhost", port=5000)
```

Then on line 18, we demonstrate the full extent of how to define types of arguments. We define the `page`, `search` and `item` arguments' types with a type hint. For the `price` argument type, on the other hand, we pass `float` as the first argument of the `Query` function call assigned as a default value. `page` and `price` are built-in python types, `int` and `float` respectively. `search` is a list of strings and `item` is a pydantic `BaseModel`.

```
1  from typing import List
2
3  from pydantic import BaseModel
```

(continues on next page)

(continued from previous page)

```python
from flask_jeroboam import Jeroboam
from flask_jeroboam import Query


app = Jeroboam("Jeroboam Inbound Features App")
app.init_app()


class Item(BaseModel):
    name: str
    count: int


@app.get("/defining_type_with_type_hints")
def get_type_hints(page: int, search: List[str], item: Item, price=Query(float)):
    return (
        f"Received arguments are :\n"
        f"page : {page}\n"
        f"search : {search}\n"
        f"price : {price}\n"
        f"item : {item}"
    )


if __name__ == "__main__":
    app.run(host="localhost", port=5000)
```

Let's test it out.

```
$ curl 'localhost:5000/defining_type_with_type_hints?page=42&search=foo&search=bar&
↪price=42.42&name=test&count=3'
Received arguments are :
page : 42
search : ['foo', 'bar']
price : 42.42
item : name='test' count=3
```

You'll notice we didn't attempt to pass a dictionary to an `item` field to the query string. Instead, we passed two arguments, `name` and `count`, corresponding to the inner fields of Item. Query Strings are usually not a good way to pass nested data structures. If you need to pass a complex data structure, use a different location like a JSON body or a form.

With request bodies, you can choose between embeded or flat arguments.

Note: Type hints are not initially supposed to provide run-time functionality. But this principle was spearheaded by pydantic and later picked up by FastAPI. So we are in good company.

Validation Options

View function arguments are essentially pydantic model fields, meaning that when you define them, you can leverage every validation feature pydantic offers on model fields.

For number types for instance, you can add `ge` (meaning greater or equal to) or `lt` (less than) values to define validation conditions on your parameters.

Let's see a simple example in which we want to make sure that the `page` argument is greater or equal to 1 (`Query(ge=1)`)

```
from flask_jeroboam import Jeroboam
from flask_jeroboam import Query


app = Jeroboam("Jeroboam Inbound Features App")
app.init_app()


@app.get("/argument_with_validation")
def get_validation_option(page: int = Query(ge=1)):
    return f"Received page argument is : {page}"


if __name__ == "__main__":
    app.run(host="localhost", port=5000)
```

Let's see what happens when we pass a page value of 0. Note that 0 is a valid int, but it is not greater or equal to 1.

```
$ curl -w 'Status Code: %{http_code}\n' 'localhost:5000/argument_with_validation?page=0'
{"detail":[{"ctx":{"limit_value":1},"loc":["query","page"],"msg":"ensure this value is␣
↪greater than or equal to 1","type":"value_error.number.not_ge"}]}
Status Code: 400
```

The server returns a 400 status code with a body giving you direction about the error: it didn't pass the `ge=1` validation.

Note: Although you can do it using a special function, I believe that this would lead to bloated view function signatures code in many cases. Beyond a couple of elementary arguments, my preference goes to defining a pydantic BaseModel first with full-blown fields and validation conditions on each of them, and then use that BaseModel as a type hint to define the shape of an inbound argument.

Cheat Sheet

In summary, your inbound arguments are defined by:

- their location: defined either implicitly or explicitly using special functions as default values (`page:int = Query()`)
- their optional default values: defined either as regular default values `page:int = 1``or explicitly using special functions (``page:int = Query(1)`)
- their type: using type hint (`page: int`) or the first argument of special functions (`page = Query(int)`)
- their optional validation options: passing additional named arguments to special functions calls (`page:int = Query(ge=1)`)

Next, check out *how outbound interfaces are defined*.

1.3.2 Defining endpoints' outbound interface with route decorators

You define your endpoints' outbound interface through the route decorators, typically by passing a `response_model` to the decorator. **Flask-Jeroboam** will use it to validate and serialize your view function's returned value.

Learn *how to define your endpoints' outbound interface* and be confident that the data you send back follows the schema you choose for them.

Outbound interface & route decorators parameters

You've seen in the *previous section* how to lay out your view functions' argument to define your endpoint's inbound interface. Now we will focus on defining the outbound interface with named arguments to your route decorators.

By outbound interface, we mean the shape of the response your server sends back when hit by the client. It encompasses both the response's *payload schema* and *status code*.

Response Model

The response model defines your response payload schema, or in other words, the shape of the data you're sending back to the client. The preferred way to do that is to pass a Pydantic BaseModel to your route decorator's `response_model` argument. Alternatively, your view function return value can implicitly define this response model.

Explicit Response Model

The most straightforward way of defining the outbound interface of your endpoint is to use the `response_model` argument of your route decorator like this `@app.get("/tasks/<int:task_id>", response_model=Task)`. This argument takes a Pydantic model as a value and will use it to validate and serialize the data returned by your view function.

Let's say that you have a `GET` endpoint that returns a `Task`. First, we define a `Task` model, inheriting from pydantic's `BaseModel`. Our `Task` model has three fields: `id`, `name`, and `description`. The `description` field is optional and has a default value of `Just here to make a point.` that will help us understand the mechanics later on.

```
from typing import Optional

from pydantic import BaseModel
from pydantic import Field

from flask_jeroboam import Jeroboam


app = Jeroboam("Jeroboam Outbound Features App")
app.init_app()


class Task(BaseModel):
    id: int
    name: str
    description: Optional[str] = Field("Just here to make a point.")
```

(continues on next page)

(continued from previous page)

```
17
18
19 @app.get("/tasks/<int:task_id>", response_model=Task)
20 def read_explicit_task(task_id: int):
21     return {"id": task_id, "name": "Find the answer."}
22
23
24 if __name__ == "__main__":
25     app.run(host="localhost", port=5000)
```

Then on line 19, we feed it to the `response_model` argument of our route decorator on line 19. Note that on line 21 we only return a dictionary with an `id` and a `name` field. The `description` field is missing, but that's okay. **Flask-Jeroboam** will add it for us through the `Task` model.

```
1  from typing import Optional
2
3  from pydantic import BaseModel
4  from pydantic import Field
5
6  from flask_jeroboam import Jeroboam
7
8
9  app = Jeroboam("Jeroboam Outbound Features App")
10 app.init_app()
11
12
13 class Task(BaseModel):
14     id: int
15     name: str
16     description: Optional[str] = Field("Just here to make a point.")
17
18
19 @app.get("/tasks/<int:task_id>", response_model=Task)
20 def read_explicit_task(task_id: int):
21     return {"id": task_id, "name": "Find the answer."}
22
23
24 if __name__ == "__main__":
25     app.run(host="localhost", port=5000)
```

Flask-Jeroboam takes the view function returned value and feeds it into your reponse_model, validates the data, serialize it into JSON, and finally wraps it into a `Response` object before handling it back to Flask.

Let's test it out:

```
$ curl http://localhost:5000/tasks/42
{"id": 42, "name": "Find the answer.", "description": "Just here to make a point."}
```

As you can see, the endpoint uses the data returned by this view function but also adds the default value of the `description` field. This is because **Flask-Jeroboam** uses the `Task` model to validate the data returned by the view function. It will add any missing fields and fill them with their default values.

To demonstrate this, let's define another endpoint that returns the same dictionary without the `response_model` argument.

```python
from flask_jeroboam import Jeroboam


app = Jeroboam("Jeroboam Outbound Features App")
app.init_app()


@app.get("/tasks/<int:task_id>/no_response_model")
def read_task_dictionnary_without_response_mode(task_id: int):
    return {"id": task_id, "name": "I'm from the dictionary."}


if __name__ == "__main__":
    app.run(host="localhost", port=5000)
```

and test it out:

```
$ curl http://localhost:5000/tasks/42/no_response_model
{"id":42,"name":"I'm from the dictionary."}
```

This time, Flask receive a plain dictionary. It will not add any default value or try to validate it against any schema. It just returns it.

Pydantic's BaseModels are a compelling way to define a complex schema. They are highly reusable and have proven an excellent tool for defining data models. For example, you can nest models, assigning a BaseModel as the type of a parent model field. You can also define validation rules, such as minimum and maximum values, regex patterns… You can even define custom validation rules. For more information on Pydantic models, check out the Pydantic documentation.

Alternatively to explicit declarations, you can also let **Flask-Jeroboam** infer the response model from the return values of your view function.

Implicit Response Model

Flask-Jeroboam can also derive your response model from the view function return type, but it has to be from annotation. In the following examples, the first endpoint will work similarly to the one from the previous section, but the second one will raise an error because Flask doesn't know what to do with the `Task` object.

```python
from typing import Optional

from pydantic import BaseModel
from pydantic import Field

from flask_jeroboam import Jeroboam


app = Jeroboam("Jeroboam Outbound Features App")
app.init_app()


class Task(BaseModel):
    id: int
    name: str
    description: Optional[str] = Field("Just here to make a point.")
```

(continues on next page)

(continued from previous page)

```python


@app.get("/tasks/<int:task_id>/implicit_from_annotation")
def read_implicit_task(task_id: int) -> Task:
    return Task.parse_obj({"id": task_id, "name": "Implicit from Annotation"})


@app.get("/tasks/<int:task_id>/implicit_no_annotation")
def read_implicit_no_annotation(task_id: int):
    return Task.parse_obj({"id": task_id, "name": "Implicit from Annotation"})


if __name__ == "__main__":
    app.run(host="localhost", port=5000)
```

Let's test it out.

```
$ curl http://localhost:5000/tasks/42/implicit_from_annotation
{"id": 42, "name": "Implicit from Annotation", "description": "Just here to make a point.
↪"}%
$ curl -w 'Status Code: %{http_code}\n' http://localhost:5000/tasks/42/implicit_no_
↪annotation
{"message":"Internal Error"}
Status Code: 500
```

However, explicit is better than implicit, so you should prefer the `response_model` argument over this approach. Plus, creating the `Task` instance feels unnecessarily wordy because, as seen before, you can directly return the dictionary. Speaking of this, let's take a look at allowed return values.

Allowed return values

When a response model is defined, **Flask-Jeroboam** can accept the following body from view functions' return values:

- a dictionary
- a dataclass instance
- a list
- a Pydantic model instance
- a Flask response instance (although it will skip the serialization part of its algorithm)

Note that, in addition to the above, you can also return a tuple of the form `(body, status_code)`, `(body, headers)`, `(body, status_code, headers)`. Both the status code and headers will be used in the response. Notably, the *Status Code* will be overridden.

Turning it off

If you don't want to use **Flask-Jeroboam**'s outbound features, turn it off by setting the `response_model` argument to `None`. It will make **Flask-Jeroboam** ignore the outbound interface of your endpoint.

```
from flask_jeroboam import Jeroboam


app = Jeroboam("Jeroboam Outbound Features App")
app.init_app()


@app.get("/tasks/<int:task_id>/response_model_off", response_model=None)
def read_response_model_off(task_id: int):
    return {"id": task_id, "name": "Response Model is off."}


if __name__ == "__main__":
    app.run(host="localhost", port=5000)
```

The endpoint still works.

```
$ curl http://localhost:5000/tasks/42/response_model_off
{"id": 1, "name": "Response Model is off."}
```

Next, let's look at another aspect of the outbound interface of an endpoint: the successful status code.

Status Code

Flask-Jeroboam supports both *registration-time* status codes and *return values* status codes.

Registration-time status code

When you register your view function, **Flask-Jeroboam** will try to solve the status code of the successful response. It will first look at the parameter *status_code* of the route decorator, then at the package-defined default value for the HTTP verb of the route decorator, and finally, the status code attribute of the response class, if any.

> **Warning:** **Flask-Jeroboam** will only be able to use this *registration-time* status code in the OpenAPI documentation of your operation.

We use the following default values for each HTTP verb:

- `GET`: 200
- `HEAD`: 200
- `POST`: 201
- `PUT`: 201
- `DELETE`: 204
- `CONNECT`: 200

- `OPTIONS`: 200
- `TRACE`: 200
- `PATCH`: 200

As you can see, you won't have to set an explicit status code most of the time.

For example, the following endpoint will have a *registration-time* status code of 201. As the view function does not return any status code, a successful put request will give us a 201 status code.

```
@app.put("/tasks", response_model=TaskOut)
def create_task(task: TaskIn):
    return {"task_id": task.id}
```

```
$ curl -w 'Status Code: %{http_code}\n' -PUT http://localhost:5000/tasks -d '{"name":
↪"My Task"}'
Status Code: 201
{"task_id": 1}
```

Now, let's say we define a second endpoint that takes a task and starts running it. In that case, you might want to override the default (`201`) with a more appropriate `202` standing for "Accepted but not done" (see RFC 7231). You would do it this way:

```
@app.put("/tasks", response_model=TaskOut, status_code=202)
def create_task(task: TaskIn):
    # Save the Task and Launch it
    return {"task_id": task.id}
```

This time, when we make a successful request, we will get a `202` status code in the response.

```
$ curl -PUT http://localhost:5000/tasks -d '{"name": "My Task"}'
Status Code: 202
{"task_id": 1}
```

Return-value status code

Flask-Jeroboam also supports returning a status code as a tuple, just like in **Flask**. It will override the *registration-time* status code, but **Flask-Jeroboam** won't be able to adjust the documentation. This could lead to inconsistencies between your documentation and the actual behaviour of the API.

If we revisit the previous example, you could achieve the same *request-handling* result with the following code:

```
@app.put("/tasks", response_model=TaskOut)
def create_task(task: TaskIn):
    # Save the Task and Launch it
    return {"task_id": task.id}, 202
```

The successful PUT request will still give us a `202` status code in the response.

```
$ curl -PUT http://localhost:5000/tasks -d '{"name": "My Task"}'
Status Code: 202
{"task_id": 1}
```

However, the resulting documentation would be different. See *OpenAPI's Auto-Documentation* for more details.

In summary, when **Flask-Jeroboam** handles the request, it will use the status code inferred at registration time unless the view function returns a value containing a status code.

If you use the OpenAPI documentation, the preferred way is to add a **registration-time** status code to guarantee consistency between your documentation and your API. Also returned status code is also supported to avoid breaking existing code.

Cheatsheet

- To define your responses' payload schema, you pass a pydantic BaseModel to the route decorator named argument `response_model` (eg. `@app.get("/task/<int:id>", response_model=TaskOut)`).
- If you want to override the implicit status code, you can use the named argument `status_code` (e.g. `@app.put("/task/<int:id>/run", status_code=202)`).
- If you want to disable the implicit response model, use the named argument `response_model=None`. (eg. `@app.get("/task/<int:id>", response_model=None)`)

Next, check out how to get the most of *OpenAPI's documentation* auto-generation.

Planned Features

- Templated views (https://github.com/jcbianic/flask-jeroboam/issues/105)
- Support Serialization options (e.g. exclude_unset) (https://github.com/jcbianic/flask-jeroboam/issues/105)

1.3.3 OpenAPI AutoDocumentation

While defining the inbound and outbound interfaces' primary purpose is to provide run-time parsing, validation, and de/serialization of inbound and outbound data for your endpoint, they also offer an excellent opportunity to generate an OpenAPI documentation automatically for your API.

Although most of it will happen without you having to write a single line of code, learn *how* you can improve your documentation.

OpenAPI's Auto-Documentation

In addition to providing request-handling functionalities, defining inbound and outbound interfaces lets you benefit from auto-generated documentation with little extra effort. It is as easy as calling the `init_app` method on your Jeroboam app instance. This will register an internal blueprint with two endpoints. One serves the OpenaAPI documentation in JSON format, and another serves the Swagger UI.

```
from typing import List
from typing import Optional

from pydantic import BaseModel

from flask_jeroboam import Blueprint
from flask_jeroboam import Jeroboam
from flask_jeroboam import Path


app = Jeroboam("Jeroboam Inbound Features App")
```

(continues on next page)

(continued from previous page)

```python
app.init_app()


class TaskIn(BaseModel):
    name: str
    description: Optional[str] = None


class TaskOut(TaskIn):
    id: int


@app.get("/health")
def get_health():
    return {"status": "Ok"}


blueprint = Blueprint("tasks", __name__, tags=["tasks"])


@blueprint.get("/tasks", response_model=List[TaskOut])
def get_tasks(page: int = 1, per_page: int = Path(10), search: Optional[str] = None):
    return [{"id": 1, "name": "Task 1"}, {"id": 2, "name": "Task 2"}]


@blueprint.get("/tasks/<int:item_id>", response_model=TaskOut)
def get_task(item_id: int):
    return {"id": 1, "name": "Task 1"}


@blueprint.put("/tasks", status_code=202, tags=["sensitive"])
def create_task(task: TaskIn):
    return {}


@blueprint.post("/tasks/<int:item_id>", tags=["sensitive"])
def edit_task(item_id: int, task: TaskIn):
    return {}


app.register_blueprint(blueprint)

if __name__ == "__main__":
    app.run(host="localhost", port=5000)
```

You can check it out at localhost:5000/openapi and localhost:5000/docs.

Turning it off

If you don't want to use the auto-documentation feature, turn it off by setting the configuration `JEROBOAM_REGISTER_OPENAPI` flag to `False`.

1.3.4 Configuration

Configuration options let you:

- Opt out of high-level features (e.g. OpenAPI AutoDocumentation)
- Handle OpeanAPI MetaData (e.g. API title, version, description, etc.)

We prefixed them with `JEROBOAM_` to avoid name collisions with other packages.

Configuration

Configuration options let you:

- Opt out of high-level features (e.g. OpenAPI AutoDocumentation)
- Handle OpeanAPI MetaData (e.g. API title, version, description, etc.)

They are prefixed with `JEROBOAM_` to avoid name collisions with other packages.

Content

General Options

General Options let you opt out of some of the **Flask-Jeroboam**'s features, in case you don't need them, need to customize, or if they are interfering with the rest of your app.

`JEROBOAM_REGISTER_OPENAPI`

It controls whether the OpenAPI Blueprint will be registered when you call the `init_app` method on the app instance.

Set to `False` if you don't want the OpenAPI Blueprint to be registered or if you want to plug in your own view functions to serve OpenAPI functionnalities.

Default: `True`

`JEROBOAM_REGISTER_ERROR_HANDLERS`

It controls whether package-defined error handlers of **Flask-Jeroboam**'s Exceptions will be registered when you call the `init_app` method on the app instance.

Set to `False` if you don't want the package-defined error handlers registered. Note that if you do this, you will need to define your own error handlers for the following exception: `RessourceNotFound`, `InvalidRequest` and `ResponseValidationError`.

Default: `True`

OpenAPI MetaData

OpenAPI MetaData Configuration options let you control the MetaData of your OpenAPI Documentation, like its title, versions, contact information, etc… Setting these is optional, meaning you will have an OpenAPI page up and running before setting these options.

`JEROBOAM_TITLE`

The title of your API. It will appear as the main title of your OpenAPI documentation page.

Default: `app.name`

`JEROBOAM_VERSION`

The version of your API. Not to be mistaken with the OPENAPI version. It will appear in the small grey tag next to your title.

Default: `0.1.0`

`JEROBOAM_DESCRIPTION`

A short description of your API. It will appear in the small grey tag next to your title.

Default: `None`

`JEROBOAM_TERMS_OF_SERVICE`

A link to the terms of service of your API. It will appear in the footer of your OpenAPI documentation page.

Default: `None`

`JEROBOAM_CONTACT`

A dictionary containing the contact information of your API. It will appear in the footer of your OpenAPI documentation page.

Default: `None`

`JEROBOAM_LICENCE_INFO`

A dictionary containing the licence information of your API. It will appear in the footer of your OpenAPI documentation page.

Default: `None`

`JEROBOAM_OPENAPI_VERSION`

The version of the OpenAPI specification that your API is compliant with. It will appear in the footer of your OpenAPI documentation page.

Default: `3.0.2`

`JEROBOAM_SERVERS`

A list of dictionaries containing the servers that your API is available on. It will appear in the footer of your OpenAPI documentation page.

Default: `[]`

`JEROBOAM_OPENAPI_URL`

The URL of your OpenAPI documentation page.

Default: `/docs`

1.4 Tutorial (coming soon)

This tutorial will walk you through creating a wine-tasting API with **Flask-Jeroboam**. We will focus on the specifics of **Flask-Jeroboam**, and will only touch on the general aspects of Flask as they are already excellent resources on the matter. If you're new to the latter, we can only recommend that you go over the Flask Tutorial, which is excellent and will put you on the right track. You will be in a much better place to appreciate what **Flask-Jeroboam** actually brings to the table and decide wether it's for you or not.

(coming soon)

1.4.1 Setting Things up

If you already have dependency management in Python figured out, skip our next section and jump to the *walkthrough*. If not, *check it out* or go to the *Documentation Overview*.

About Dependency Management

Virtual environments

Virtual environments are essential to isolate your project's dependencies from the system-wide packages, preventing version conflicts and providing reproducibility and ease of deployment to contributors and yourself. Additionally, it allows for using different versions of packages for various projects on the same machine without conflicts.

It's a good practice, a necessary one even.

I find the combination of pyenv and poetry to work very well together, and I will walk you through an example. But anything that's already working for you is fine.

Python Version

Your first dependency, and the main one at that, is your Python installation. When you overlook this, you end up using your system's default, often outdated, Python installation.

The best practice is to use the latest stable version of Python, which is 3.11 as I write this. *see how to install a specific python version*. The Python core team is doing a fantastic job, and it would be a shame to miss out on all the improvement they bring to the game release after release.

That being said, **Flask-Jeroboam** supports Python down to its 3.8 installment. It means that the CI/CD pipeline tests the package from Python 3.8 to the most recent release. As python versions reach their end of life, we will drop supporting them but keep up with the newest releases.

A complete installation walkthrough

To follow this section, you must have pyenv and poetry installed on your system. If this is not the case, follow the following instructions: installing pyenv and installing poetry.

Install the latest Python version

First, you want to pick a specific Python version to install and activate. As said above, the latest stable version is your best option. Let's install it using pyenv:

```
# Output may vary
# Install the latest version of Python
$ pyenv install 3.11
Downloading Python-3.11.1.tar.xz...
-> https://www.python.org/ftp/python/3.11.1/Python-3.11.1.tar.xz
Installing Python-3.11.1...
Installed Python-3.11.1 to XXXX/.pyenv/versions/3.11.1
# Activate it
$ pyenv local 3.11
# Check if it worked
$ python --version
Python 3.11.1
```

Once you've secured the correct Python version, you can create a virtual environment for your project.

Create an environment

The poetry CLI can either start the project from scratch (with minimal scaffolding) or hook to an existing project.

In the latter case, the poetry CLI will prompt you for meta information like your project's title, description, author, and license. Don't worry too much about it now: you can edit any of this information in the `pyproject.toml` file later.

Let's assume you're starting a new project without using poetry's scaffolding capabilities.

```
# Make root dir and move to it
$ mdir jeroboam-demo && cd jeroboam-demo
# Create a poetry environment
$ poetry init
# Make sure you hooked the env to the intended version of Python
$ poetry use 3.11
```

Activate the environment

Before you do anything on your project, you must activate the corresponding environment:

```
$ poetry shell
```

If configured with the right plugins, your shell prompt will change to show the name of the activated environment, which will come in handy.

Note: Alternatively, you can use shell plugins to *activate automatically virtual environments created by Poetry* like zsh-poetry.

Add & Install Flask-Jeroboam in your environment

Now you are ready to install **Flask-Jeroboam**. As we've seen before, this would go like this:

```
$ poetry add flask-jeroboam
```

1.4.2 Other Ressources

- Michael Grinberg's Mega Tutorial is a must read. It's a great introduction to Flask and covers a lot of ground.
- FastAPI Documentation is in my opinion a gold standard. It's very well written, and thorough and the examples are easy to follow. You should definitely check it out. It holds value even if you're not going to use FastAPI.

CHAPTER

TWO

API REFERENCE

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

2.1 API Reference

In this part of the documentation, we will cover **Flask-Jeroboam**'s internals.

2.1.1 App & Blueprint Objects

class flask_jeroboam.jeroboam.**Jeroboam**(**args*, ***kwargs*)

Bases: `JeroboamScaffoldOverRide`, `Flask`

A Flask Object with extra functionalities.

The route method is overriden by a custom flask_jeroboam route decorator.

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

init_app(*app=None*)

Setup is performed after app has received all its configuration.

Parameters
app (`Jeroboam` | *None*) –

Return type
None

property openapi: OpenAPI

Get the OpenApi object.

query_string_key_transformer: Callable | None = None

response_class

alias of `JSONResponse`

url_rule_class

alias of `JeroboamRule`

class flask_jeroboam.blueprint.**Blueprint**(**args*, *tags=None*, *include_in_openapi=True*, ***kwargs*)

Bases: `JeroboamScaffoldOverRide`, `Blueprint`

Regular Blueprint with extra behavior on route definition.

Parameters

- **args** (*Any*) –
- **tags** (*List[str] | None*) –
- **include_in_openapi** (*bool*) –
- **kwargs** (*Any*) –

2.1.2 Jeroboam View

class flask_jeroboam.view.**JeroboamView**(*rule*, *original_view_func*, *options*, *response_class=<class 'flask_jeroboam.responses.JSONResponse'>*)

Bases: `object`

Adds flask-jeroboam features to a regular flask view function.

The InboundHandler and OutboundHandler are configured here. The as_view property returns an augmented view function ready to be used by Flask, adding inbound and outbound data handling behavior if needed. The resulting view function is a regular flask view function, and any overhead related to figuring out what needs to be done is spent at registration time.

Parameters

- **rule** (*str*) –
- **original_view_func** (*Callable[[...], Response | DataClassType | BaseModel | str | bytes | List[Any] | List[BaseModel] | Mapping[str, Any] | Iterator[str] | Iterator[bytes] | DataclassProtocol | Tuple[Response | DataClassType | BaseModel | str | bytes | List[Any] | List[BaseModel] | Mapping[str, Any] | Iterator[str] | Iterator[bytes] | DataclassProtocol, Headers | Mapping[str, str | List[str] | Tuple[str, ...]] | Sequence[Tuple[str, str | List[str] | Tuple[str, ...]]] | Tuple[Response | DataClassType | BaseModel | str | bytes | List[Any] | List[BaseModel] | Mapping[str, Any] | Iterator[str] | Iterator[bytes] | DataclassProtocol, int] | Tuple[Response | DataClassType | BaseModel | str | bytes | List[Any] | List[BaseModel] | Mapping[str, Any] | Iterator[str] | Iterator[bytes] | DataclassProtocol, int, Headers | Mapping[str, str | List[str] | Tuple[str, ...]] | Sequence[Tuple[str, str | List[str] | Tuple[str, ...]]] | WSGIApplication] | Callable[[...], Awaitable[Response | DataClassType | BaseModel | str | bytes | List[Any] | List[BaseModel] | Mapping[str, Any] | Iterator[str] | Iterator[bytes] | DataclassProtocol | Tuple[Response | DataClassType | BaseModel | str | bytes | List[Any] | List[BaseModel] | Mapping[str, Any] | Iterator[str] | Iterator[bytes] | DataclassProtocol, Headers | Mapping[str, str | List[str] | Tuple[str, ...]] | Sequence[Tuple[str, str | List[str] | Tuple[str, ...]]] | Tuple[Response | DataClassType | BaseModel | str | bytes | List[Any] | List[BaseModel] | Mapping[str, Any] | Iterator[str] | Iterator[bytes] | DataclassProtocol, int] | Tuple[Response | DataClassType | BaseModel | str | bytes |*

List[Any] | List[BaseModel] | Mapping[str, Any] | Iterator[str] | Iterator[bytes] | DataclassProtocol, int, Headers | Mapping[str, str | List[str] | Tuple[str, ...]] | Sequence[Tuple[str, str | List[str] | Tuple[str, ...]]]] | WSGIApplication]]) –

- **options** (*Dict[str, Any]*) –
- **response_class** (*Type | None*) –

property as_view: Callable[[...], Response | DataClassType | BaseModel | str | bytes | List[Any] | List[BaseModel] | Mapping[str, Any] | Iterator[str] | Iterator[bytes] | DataclassProtocol | Tuple[Response | DataClassType | BaseModel | str | bytes | List[Any] | List[BaseModel] | Mapping[str, Any] | Iterator[str] | Iterator[bytes] | DataclassProtocol, Headers | Mapping[str, str | List[str] | Tuple[str, ...]] | Sequence[Tuple[str, str | List[str] | Tuple[str, ...]]]] | Tuple[Response | DataClassType | BaseModel | str | bytes | List[Any] | List[BaseModel] | Mapping[str, Any] | Iterator[str] | Iterator[bytes] | DataclassProtocol, int] | Tuple[Response | DataClassType | BaseModel | str | bytes | List[Any] | List[BaseModel] | Mapping[str, Any] | Iterator[str] | Iterator[bytes] | DataclassProtocol, int, Headers | Mapping[str, str | List[str] | Tuple[str, ...]] | Sequence[Tuple[str, str | List[str] | Tuple[str, ...]]]] | WSGIApplication] | Callable[[...], Awaitable[Response | DataClassType | BaseModel | str | bytes | List[Any] | List[BaseModel] | Mapping[str, Any] | Iterator[str] | Iterator[bytes] | DataclassProtocol | Tuple[Response | DataClassType | BaseModel | str | bytes | List[Any] | List[BaseModel] | Mapping[str, Any] | Iterator[str] | Iterator[bytes] | DataclassProtocol, Headers | Mapping[str, str | List[str] | Tuple[str, ...]] | Sequence[Tuple[str, str | List[str] | Tuple[str, ...]]]] | Tuple[Response | DataClassType | BaseModel | str | bytes | List[Any] | List[BaseModel] | Mapping[str, Any] | Iterator[str] | Iterator[bytes] | DataclassProtocol, int] | Tuple[Response | DataClassType | BaseModel | str | bytes | List[Any] | List[BaseModel] | Mapping[str, Any] | Iterator[str] | Iterator[bytes] | DataclassProtocol, int, Headers | Mapping[str, str | List[str] | Tuple[str, ...]] | Sequence[Tuple[str, str | List[str] | Tuple[str, ...]]]] | WSGIApplication]]

Decorate the orinal view function with handlers..

TODO: Deal with name, docs, before decorating

property main_method: str

Return the main HTTP verb of the Endpoint.

property parameters: List[SolvedArgument]

Return the main HTTP verb of the Endpoint.

2.1.3 API Reference

flask_jeroboam.view_arguments.functions.**Path**(**args*, ***kwargs*)

Declare A Path parameter.

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type

Any

flask_jeroboam.view_arguments.functions.**Query**(**args*, ***kwargs*)

Declare A Query parameter.

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
Any

flask_jeroboam.view_arguments.functions.**Header**(**args*, ***kwargs*)

Declare A Header parameter.

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
Any

flask_jeroboam.view_arguments.functions.**Cookie**(**args*, ***kwargs*)

Declare A Cookie parameter.

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
Any

flask_jeroboam.view_arguments.functions.**Body**(**args*, ***kwargs*)

Declare A Body parameter.

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
Any

flask_jeroboam.view_arguments.functions.**Form**(**args*, ***kwargs*)

Declare A Form parameter.

Parameters

- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type
Any

flask_jeroboam.view_arguments.functions.**File**(**args*, ***kwargs*)

Declare A File parameter.

Parameters

- **args** (*Any*) –

- **kwargs** (*Any*) –

Return type
Any

CHAPTER

THREE

ADDITIONAL NOTES

3.1 Contributor Guide

I appreciate your interest in improving this project. This project is open-source under the MIT license and welcomes contributions in the form of bug reports, feature requests, and pull requests. Currently, our focus is on **improving documentation** and **hunting for bugs**.

We intend to use the Issue Tracker to coordinate the community and provide templates for bug reports, feature requests, documentation updates, and implementation improvements. So be sure to use the appropriate template with further instructions on writing any.

3.1.1 Ressources

Here is a list of important resources for contributors:

- Source Code
- Documentation
- Issue Tracker
- *Code of Conduct*

3.1.2 How to set up your development environment

You need Python 3.8+ and the following tools:

- Poetry
- Nox
- nox-poetry

Install the package with development requirements:

```
$ poetry install
```

You can now run an interactive Python session, or the command-line interface:

```
$ poetry run python
$ poetry run flask-jeroboam
```

3.1.3 How to test the project

Run the full test suite:

```
$ nox
```

List the available Nox sessions:

```
$ nox --list-sessions
```

You can also run a specific Nox session. For example, invoke the unit test suite like this:

```
$ nox --session=tests
```

Unit tests are located in the *tests* directory, and are written using the pytest testing framework.

3.1.4 How to submit changes

Open a pull request to submit changes to this project.

Your pull request needs to meet the following guidelines for acceptance:

- The Nox test suite must pass without errors and warnings.
- Include unit tests. This project maintains 100% code coverage.
- If your changes add functionality, update the documentation accordingly.

Feel free to submit early, though—we can always iterate on this.

To run linting and code formatting checks before committing your change, you can install pre-commit as a Git hook by running the following command:

```
$ nox --session=pre-commit -- install
```

It is recommended to open an issue before starting work on anything. This will allow a chance to talk it over with the owners and validate your approach.

3.2 Contributor Covenant Code of Conduct

3.2.1 Our Pledge

We as members, contributors, and leaders pledge to make participation in our community a harassment-free experience for everyone, regardless of age, body size, visible or invisible disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, caste, color, religion, or sexual identity and orientation.

We pledge to act and interact in ways that contribute to an open, welcoming, diverse, inclusive, and healthy community.

3.2.2 Our Standards

Examples of behavior that contributes to a positive environment for our community include:

- Demonstrating empathy and kindness toward other people
- Being respectful of differing opinions, viewpoints, and experiences
- Giving and gracefully accepting constructive feedback
- Accepting responsibility and apologizing to those affected by our mistakes, and learning from the experience
- Focusing on what is best not just for us as individuals, but for the overall community

Examples of unacceptable behavior include:

- The use of sexualized language or imagery, and sexual attention or advances of any kind
- Trolling, insulting or derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or email address, without their explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

3.2.3 Enforcement Responsibilities

Community leaders are responsible for clarifying and enforcing our standards of acceptable behavior and will take appropriate and fair corrective action in response to any behavior that they deem inappropriate, threatening, offensive, or harmful.

Community leaders have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, and will communicate reasons for moderation decisions when appropriate.

3.2.4 Scope

This Code of Conduct applies within all community spaces, and also applies when an individual is officially representing the community in public spaces. Examples of representing our community include using an official e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event.

3.2.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported to the community leaders responsible for enforcement at jc.bianic@gmail.com. All complaints will be reviewed and investigated promptly and fairly.

All community leaders are obligated to respect the privacy and security of the reporter of any incident.

3.2.6 Enforcement Guidelines

Community leaders will follow these Community Impact Guidelines in determining the consequences for any action they deem in violation of this Code of Conduct:

1. Correction

Community Impact: Use of inappropriate language or other behavior deemed unprofessional or unwelcome in the community.

Consequence: A private, written warning from community leaders, providing clarity around the nature of the violation and an explanation of why the behavior was inappropriate. A public apology may be requested.

2. Warning

Community Impact: A violation through a single incident or series of actions.

Consequence: A warning with consequences for continued behavior. No interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, for a specified period of time. This includes avoiding interactions in community spaces as well as external channels like social media. Violating these terms may lead to a temporary or permanent ban.

3. Temporary Ban

Community Impact: A serious violation of community standards, including sustained inappropriate behavior.

Consequence: A temporary ban from any sort of interaction or public communication with the community for a specified period of time. No public or private interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, is allowed during this period. Violating these terms may lead to a permanent ban.

4. Permanent Ban

Community Impact: Demonstrating a pattern of violation of community standards, including sustained inappropriate behavior, harassment of an individual, or aggression toward or disparagement of classes of individuals.

Consequence: A permanent ban from any sort of public interaction within the community.

3.2.7 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 2.1, available at https://www.contributor-covenant.org/version/2/1/code_of_conduct.html.

Community Impact Guidelines were inspired by Mozilla's code of conduct enforcement ladder.

For answers to common questions about this code of conduct, see the FAQ at https://www.contributor-covenant.org/faq. Translations are available at https://www.contributor-covenant.org/translations.

3.3 License

3.4 Changes

3.4.1 beta releases

The beta releases mark a turning point and the intention to outgrow **Flask-Jeroboam**'s initial internal use. The 0.1.0 version is primarily based on forking FastAPI and departs from the first naive implementation of alpha releases.

Version 0.1.0.beta2

Released March, 9th 2023

- Writing Documentation
- Fixing bugs
- Breaking Change: Fixing the embed mechnism for request bodies

Version 0.1.0.beta

Released February, 22nd 2023

- Support for explicit location of inbound arguments with special functions
- Support for validation options on explicit inbound arguments
- Response Serialization mechanism is improved
- OpenAPI Documentation Auto-generation
- You can add Tags to endpoints and blueprints
- Extensive Refactoring of the codebase

3.4.2 alpha releases

The alpha releases share a minimalist implementation of a tiny portion of the targetted features set. They are a packaged version of helper functions that I used to level up one particular flask project and are largely overfitted to this specific context.

Version 0.0.3.alpha

Released January, 16th 2023

- Improved request parsing
- Introducing Model Utils (Parsers and Serializers)
- Improvement of type hinting

Version 0.0.2.alpha

Released January, 9th 2023

- Upgrade Dependencies

Version 0.0.1.alpha

Released August, 16th 2022

- First public version
- Basic request parsing-validation-injection using type hints of view function arguments
- Basic response serialization using response_model on route decorators

PYTHON MODULE INDEX

f

INDEX