Gofri Documentation

Release 1.0.3

ThomasKenyeres

Apr 12, 2018

Getting started:

CHAPTER 1

Installation

Install from pip To install latest stable version run `$ pip3 install Gofri`.

To install current development version check out the GitHub repository.

CHAPTER 2

Initializing a project

`$ python3 -m gofri.generate_project <project-name>` is the common way to create a project, which will have the following structure

```
My-First-Project
 my_first_project
     __init__.py
     start.py
     conf.xml
     modules.py
     generate.py
     back
         __init__.py
         controller
             __init__.py
             ...
         dao
             __init__.py
             ...
         ...
     web
         <web content if needed>
```

CHAPTER 3

Configuration file

The main configuration file is `conf.xml` and it's in the root package.

3.1 HTTP

```
<configuration>
    <project>
        <app-path>example.app</app-path>
    </project>
    <hosting>
        <host></host>
        <port>8080</port>
    </hosting>

    ...

</configuration>
```

If you leave `<host>` empty, the default is 127.0.0.1.

3.2 Dependencies

```
<configuration>
    <dependencies>
        <dependency>matplotlib</dependency>
        <dependency>pygame</dependency>
    </dependencies>

    ...

</configuration>
```

This is how you can specify project dependencies in current version. There will be more effective solutions for this purpose.

Note: Dependencies are automatically installed at startup, if they are not installed yet.

CHAPTER 4

Start your application

To start your webapp run `$ python3 start.py` in the root package.

Expected output after startup:

```
GOFRI -- version: 1.0.1
################################################################################

All required dependencies are installed
* Running on http://127.0.0.1:8080/ (Press CTRL+C to quit)
```

CHAPTER 5

Generator

Each generated project has a `generate.py` in the root package. It's a tool for rapid module generation for your project.

Main features:

Command: generate

Generate custom module:

```
generate
    module <name> <path>
```

Generate predefined modules:

```
generate
    controller <name>
    filter <name>
    service <name>
    dto <name>
    entity <name> [column names separated with space]
```

Example usage:

```
$ python3 generate.py generate entity dog name breed birth_year
```

CHAPTER 6

HTTP request handling

6.1 Configuration

Configuration with conf.xml

6.2 HTTP controllers

HTTP controllers are modules which are responsible for receiving requests and sending responses to a client through HTTP connection.

Tip: It's recommended to separate some request handler functions into more controllers to avoid too big, messy files.

Generate a controller The common way to create a controller is using generate.py for this purpose:

```
$ python3 generate.py generate controller <name>
```

6.3 Decorator usage

The basic gofri HTTP decorators are in the `gofri.lib.decorate.http` module, and they are responsible for route management. They are `GET`, `POST`, `PUT`, `HEAD` and `DELETE` depending on what request method do you want to accept on the endpoint.

```
from gofri.lib.decorate.http import GET, POST, PUT, HEAD, DELETE
```

This is the first (positional) argument, which is always required:

- **`path`** Defines the url path of the given endpoint, you can also define path variables: `path="/people/<person_id>"`, which have to be the first arguments of the decorated function (They are positional arguments of the function).

Example usage with `@GET`:

```
@GET(path="people/<person_id>")
def get_person(person_id):
    return person_service.get(person_id)
```

The remaining variables are strings which contain the names of the request part attributes seperated by semicolons(`;`). Example usage: `header='username;password'`. The decorated function expects arguments with the defined names, so avoid using the same names even in different decorator values!

- **`params`** The variables defined here are the path parameters of a request, e.g. `/search?keyword=house&where=images` so the function's arguments would be `keyword` and `where`.
- **`headers`** The request header parameters are defined in this string.

@GET

```
@GET("/people")
def get_people():
    return [Person("Jane"), Person("Jack")]
```

```
@GET(path="/person", params="person_id")
def get_person(person_id):
    return person_service.get(person_id)
```

Example: `<host>/person?person_id=1`

@POST

`@POST` has additional input values like:

- **body** Contains the request body.

```
@POST(path="/add", body="name;age")
def add_person(name, age):
    person_service.add(Person(name, age))
```

- **json** Contains the request body if it's in `application/json` format:

```
@POST(path="/add_more", json="people")
def add_people(people):
    person_service.add_more(people)
```

Request body (`application/json`):

```
{
    "people": [
        {"name": "John", "age": 23},
        {"name": "Jane", "age": 18},
        {"name": "Jack", "age": 34}
    ]
}
```

`headers` example:

```
@POST(path="/auth", headers="name;password")
def auth(name, password):
    return service.auth()
```

You can also use more request part decorator value at once:

```
@POST(path="/library/<room_id>", json="books", params="note"):
def add_books(room_id, books, note):
    library.rooms[room_id].add_books(books, note)
```

More `@HEAD`, `@PUT` and `@DELETE` is also available, they work the same way as `@POST`.

Note: The big advantage of `Gofri`'s HTTP decorators is that you don't have to read different request parts inside the function because you have them as parameters. If you want to use request parts differently, do as you would do it in `Flask`.

CHAPTER 7

Filters

7.1 What are these?

HTTP filters are tools for filtering requests and set different properties before the request handler functions run. To setup a filter is very easy you just have to create a class with `@GofriFilter()` decorator.

```
@GofriFilter()
class MyFilter(Filter):
    def filter(self, request, response):
        return self._continue(request, response)
```

Tip: It's recommended to inherit your class from `gofri.lib.http.filter.Filter`, but works anyway, so your IDE can easily recognize overrideable methods.

Methods of `Filter` class:

- **filter(request, response)** The filtering logic should be implemented here, and to let the request go forward `_continue(request, response)` should be called.
- **_continue(request, response)** When this method is called, the `request` and `response` is passed to the next filter or to a controller which has a method to handle requests on the specific url.

7.2 Filtering by urls

By default a filter doesn't filter anything, it's configurable in the decorator `GofriFilter()`. The URLs which you want to filter are given in the decorator's `urls` value, which is a list with the given URLs in string format.

```
@GofriFilter(urls=["/vpost", "/send"])
class MyFilter(Filter):
    def filter(self, request, response):
        return self._continue(request, response)
```

If you set `@GofriFilter()` decorator value `filter_all` to `True`, the filter activates on all URL endpoints on the server.

```
@GofriFilter(filter_all=True)
class MyFilter(Filter):
    def filter(self, request, response):
        return self._continue(request, response)
```

7.3 Specify order

You can configure that in which order do you want to make your filters work. The lower the order value is, the sooner the filter works.

```
@GofriFilter(filter_all=True, order=1)
class AFilter(Filter):
    ...

@GofriFilter(filter_all=True, order=0)
class BFilter(Filter):
    ...
```

In the example above, `BFilter` filters before `AFilter`. The default order is `0`. If two filters has the same order, they are ranked by definition order in the code.

Note: You don't have to specify the order of your filters strictly like `0-1-2...`, it also works well with orders `2-3-6`.