

## 1 Getting to know the REST API documentation <https://starkovden.github.io/intro-rest-api.html>

**REST API: REST (Representational State Transfer)** are web services that allow you to send requests to resources via URL paths. Specifies the operation to be performed using a path (for example, GET, CREATE, DELETE). As with other web service APIs, requests and responses are sent over HTTP over the Internet, and the servers that accept requests are independent of the language of the request (it doesn't have to be a specific programming language). Responses are usually returned in JSON or XML format. REST APIs have many different paths (endpoints) with different parameters that can be configured to determine the desired results.

### Free APIs

<https://apistlist.fun/collection/free-apis>

## 2 Flask Web Development

### Links

Documentation: <https://flask.palletsprojects.com/>

Changes: <https://flask.palletsprojects.com/changes/>

PyPI releases: <https://pypi.org/project/flask/>

*Install and update using pip:*

```
$ pip install -U Flask
```

*A Simple Example*

```
# save this as app.py
from flask import Flask
```

```
app = Flask(__name__)
```

```
@app.route("/")
```

```
def hello():
```

```
    return "Hello World!"
```

```
$ flask run
```

```
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

## 2.1 Initialization

All Flask programs must instantiate a program. A program instance is an object of the Flask class.

A web server uses a protocol called the Web Server Interface (WSGI) to forward all requests received from a client to this object for processing. It is often created using the following code:

```
from flask import Flask
app = Flask(__name__)
```

The Flask class constructor has only one parameter that must be specified, namely the name of the program's main module or package, `__name__`. If you don't pass an integer value, you'll get an error and won't be able to enter the default module name: this will affect access to static files and won't affect access to browsing functions.

## 2.2 Routing and browsing functions

The client (such as a web browser) sends a request to the web server, and the web server sends the request to an instance of the Flask program. The program instance needs to know what code to run for each request for a URL, so it keeps the relation of displaying a URL to a Python function.

**Routing:** A program that handles the relationship between URLs and features.

In a Flask program, use the `app.route` decorator provided by the program instance to define a route, and register the decorated function as a route. The following example shows how to use this decorator to declare a route:

```
@app.route("/")
def index():
    return '<h1>Hello World!</h1>'
```

**View functions and responses:** The previous example registers the `index()` function as a handler for the program's root address. When the browser accesses the server where the program is deployed, it starts the server to execute the `index()` function. The return value of this function is called `IsresponseIs` the content received by the client. If the client is a web browser, the response is a document that is displayed to the user for viewing. Functions like `index()` are called `ViewFunction(ViewFunction)`. The response returned by the view function can be a simple string containing HTML, or a complex form.

Some URL formats contain variable parts. The content within the angle brackets is the dynamic part, and any URL that matches the static part will be matched to that route. When calling the view function, Flask passes the dynamic part as a parameter to the function.

The dynamic part of the route uses strings by default, but `typedefs` can also be used. For example, route `/user/<int:id>` Only matches URLs whose dynamic clip ID is an integer Flask supports `int`, `float` and `path` types in routing. The code looks like this:

**path type:** Is a string, but the slash is not treated as a delimiter, but as part of a dynamic segment.

```
@app.route('/user/<name>')
def user(name):
    return '<h1>Hello, %s!</h1>' % name
```

## 2.3 Start the server

The program instance uses the `run` method to start a Flask-integrated development web server (the server provided by flask is not suitable for use in development mode): the `app.run()` function can receive parameters that set the web server's mode of operation: `debug mode->debug=True`:

```
if __name__ == '__main__':
```

```
app.run(debug=True)
```

`name=='main'` : is used to evaluate program entry to ensure that the development web server is only started when this script is run directly. If this script is injected by another script, the value displayed by `__name__` is the name of the module.

## 2.4 Complete program

The program code is shown in Example 2-1.

```
from flask import Flask
app = Flask(__name__)
@app.route("/")
def index():
    return '<h1>Hello World!</h1>'

@app.route("/user/<name>")
def user('/user/<name>'):
    return '<h1>Hello, %s!</h1>' % name
if __name__ == '__main__':
    app.run(debug=True)
```

## 2.5 Request-response cycle

Now that you've developed a simple Flask program, the following sections will introduce some of the concepts behind Flask's design.

### 2.5.1 Request procedure and context

When Flask receives a request from a client, the view function must have access to some objects in order for the request to be processed. A good example is the request object, it encapsulates the HTTP request sent by the client. Flask uses context to temporarily make certain objects globally available. With context, you can write the following view function:

```
from flask import request
@app.route("/")
def index():
    user_agent = request.headers.get('User Agent')
    return '<p>Your browser is %s</p>' % user_agent
```

**note:** How do we use query as a global variable in this view function. In fact, the request cannot be a global variable. Imagine that on a multi-threaded server, when multiple threads simultaneously process different requests sent by different clients, the request object seen by each thread must be different. Flask uses context to make certain variables globally available on the thread.

**context:** Equivalent to a container that stores some information while the Flask program is running.

Query context variables as shown in the following table:

| variable | content  |
|----------|--|
| request  | The content of the HTTP request is encapsulated and the HTTP request is the target. <code>user = request.args.get('user')</code> , get get request parameters. |

| variable | content  |
|----------|--|
| session  | It is used to record information in a request session and implement a stateful setting that is intended for user information. session.get('name') gets information about the user. |

Program context variables, as shown in the following table:

| variable    | content   |
|-------------|---|
| g           | The object is used as temporary storage when processing a request. This variable is reset on every request  |
| current_app | The program instance of the currently activated program, used to record configuration information while the program is running, such as a project log to record a project |

**note:** Flask activates (or pushes) the program and request context before propagating the request, and removes it after processing the request. After clicking on the program context in the thread, the current\_app and g variables can be used. request and session variables. If we don't activate a program or request context when using these variables, it will throw an error. Call app.app\_context() on the program instance to get the program context.

### 2.5.2 Query planning

When the program receives a request from a client, it needs to find the view function that handles the request. To accomplish this task, Flask will find the requested URL in the program's URL mapping. Flask uses the app.route decorator or the non-decorator app.add\_url\_rule() to generate the mapping. To see what the URL mapping looks like in a Flask program, we can check the mapping generated for hello.py in the Python shell.

**URL Mapping:** Correspondence between URL and view function.

```
print (app.url_map)
-----
Map([<rule '/' (HEAD,OPTIONS,GET) ->index>,
<rule '/static/<filename>' (HEAD,OPTIONS,GET) ->static>,
<rule '/user/<name>' (HEAD,OPTIONS,GET) ->user>])
```

**note:** HEAD, Options, and GET in the URL mapping are request methods that are handled by the route. Flask defines a request method for each route, so when different request methods are sent to the same URL, different view functions are used for processing. The HEAD and OPTIONS methods are automatically handled by Flask, and three routes use the GET method by default. You can add a request method to the decorator: @app.route('/', ["GET", "POST"])

### 2.5.3 Hook request

Sometimes it's useful to execute code before or after a request is processed. For example, at the start of a request, we may need to create a database connection or authenticate the user who initiated the request. Flask provides a feature for registering shared functions. Registered functions can be called before or after the request is passed to the view function. Request hooks are implemented using decorators. The flask supports the following 4 kinds of hooks.

| hook                 | content  |
|----------------------|--|
| before_first_request | Register a function to run before processing the first request.                                  |
| before_request       | Register a function to run before each request.  |
| after_request        | Register a function and run it after each request unless an unhandled exception is thrown.       |
| teardown_request     | Register a function that will run after every request, even if an unhandled exception is thrown. |

**note:** Communication between the request hook function and the view function usually uses the global context variable `g`. For example, the `before_request` handler might load the registered user from the database and store it in `g.user`. When the view function is subsequently called, the view function uses `g.user` to get the user.

#### 2.5.4 answer

Once Flask calls the view function, its return value will be used as the content of the response. In most cases, the response is a simple string that is sent back to the client as an HTML page. But the HTTP protocol requires more than a string in response to a request. An important part of the HTTP response is the status code.

**(1) Return different status codes as required:** If you want to use a different status code in the response returned by the view function, you can use a numeric code as the second return value and add it to the response text. An example program looks like this:

```
@app.route("/")
def index():
    return '<h1>Bad Request</h1>', 400
```

**(2) Flask's view function can also return a Response object:** The `make_response()` function can take 1, 2, or 3 parameters (similar to the return value of a view function) and returns a Response object. Sometimes we need to do this conversion in a view function and then call various methods on the response object to further set the response. The following example creates a response object and then sets a cookie:

```
from flask import make_response
@app.route("/")
def index():
    response = make_response('<h1>This document carries a cookie!</h1>')
    response.set_cookie('answer', '42')
    return response
```

**(3) Redirected as view function return type** There is no document page for this response, only the browser is told a new address to load the new page. Redirection is often used in web forms. A redirect is often indicated with a 302 status code, and the specified address is provided by the Location header. Flask provides a `redirect()` helper function to generate this response:

```
from flask import redirect
@app.route("/")
def index():
    return redirect("http://www.example.com")
```

**Response generated by the interrupt function,** Interrupt function: exception handling statement in flask. The interrupt function can only generate an abnormal status code corresponding

to the http protocol. In the following example, if the user corresponding to the dynamic parameter ID in the URL does not exist, it returns a 404 status code:

Feature: A break feature is typically used to implement a custom error message to make code more scalable and improve user experience.

```
from flask import abort
@app.route('/user/<id>')
def get_user(id):
    user = load_user(id)
    if not user:
        abort(404)
    return '<h1>Hello, %s</h1>' % user.name
```

Note: abort does not return control to the function that called it, but throws an exception and transfers control to the web server. While the code following the interrupt is not executed.

## 2.6 Flask expansion

Flask is designed to be extensible, so it doesn't provide some important features like database and user authentication, so developers are free to choose the most suitable package for the program or develop according to their needs. Next we can [hello.py](#) As an example, an extension has been added to use command line options to improve the function of the program. By using Flask-Script to support command line options, the Flask development web server supports many run customization options, but can only be passed as options in the script to the app.run() function.

**Flask-Script:** A Flask extension that adds a command line parser to a Flask program. Flask-Script comes with a set of commonly used options and also supports custom commands. The Flask-Script extension is installed with pip: pip install flask-script.

Program example:

```
from flask import Flask
from flask_script import Manager

app = Flask(__name__)
manager = Manager(app)

@app.route('/')
def index():
    return '<h1>Hello World!</h1>'

if __name__ == '__main__':
    manager.run()
```

**Viewing terminal commands on the command line :** python [hello.py](#) runserver --help

**Terminal command to start the server :** python3 [hello.py](#) runserver -p 5000

**Call start method instead of application:** Runserver must be added to the Python interpreter configuration script options

**Effect:** You can manually enter the IP and port in the terminal, you can configure the script parameters, you can achieve the database migration

The --host option is a very useful option, it tells the web server which network interface to listen on for the client.connect. By default, the Flask development web server listens for connections on the local host, so it only accepts services from a connection initiated by the computer it is

located on. server.

The following command allows the web server to listen for a connection over a public network interface, which allows other computers on the network to connect to the server:

- (one) pythonhello.pyrunserver --host 0.0.0.0
- (2) Running on<http://0.0.0.0:5000/>
- (3) Restarting with reloaderWeb server is now available<http://abcd:5000/>Access from any computer on the network, where "abcd" is the external IP address of the computer hosting the server.

### Task- Submit form data to database

#### 1- Check for libraries!

**pip install Flask-MySQLdb**

**App.py source code:**

```
from flask import Flask, render_template, request
from flask_mysql import MySQL
import yaml

app = Flask(__name__)

#DB configuration
db = yaml.load(open('db.yaml'))
app.config['MYSQL_HOST'] = db['mysql_host']
app.config['MYSQL_USER'] = db['mysql_user']
app.config['MYSQL_PASSWORD'] = db['mysql_password']
app.config['MYSQL_DB'] = db['mysql_db']
app.config['MYSQL_CURSORCLASS'] = 'DictCursor'
mysql = MySQL(app)

@app.route('/', methods=['GET', 'POST'])
def index():
    if request.method == 'POST':
        form = request.form
        name = form['name']
        age = form['age']
        cursor = mysql.connections.cursor()
        cursor.execute("INSERT INTO employee(name, age) VALUES(%, %s)",
            (name, age))
        mysql.connections.commit()
        return render_template('index.html')

@app.route('/employees')
def employees():
    cursor = mysql.connections.cursor()
    result_value = cursor.execute("SELECT * FROM employee")
    if result_value > 0:
        employees = cursor.fetchall()
        return render_template('employees.html', employees=employees)

if __name__ == '__main__':
    app.run(debug=True)
```

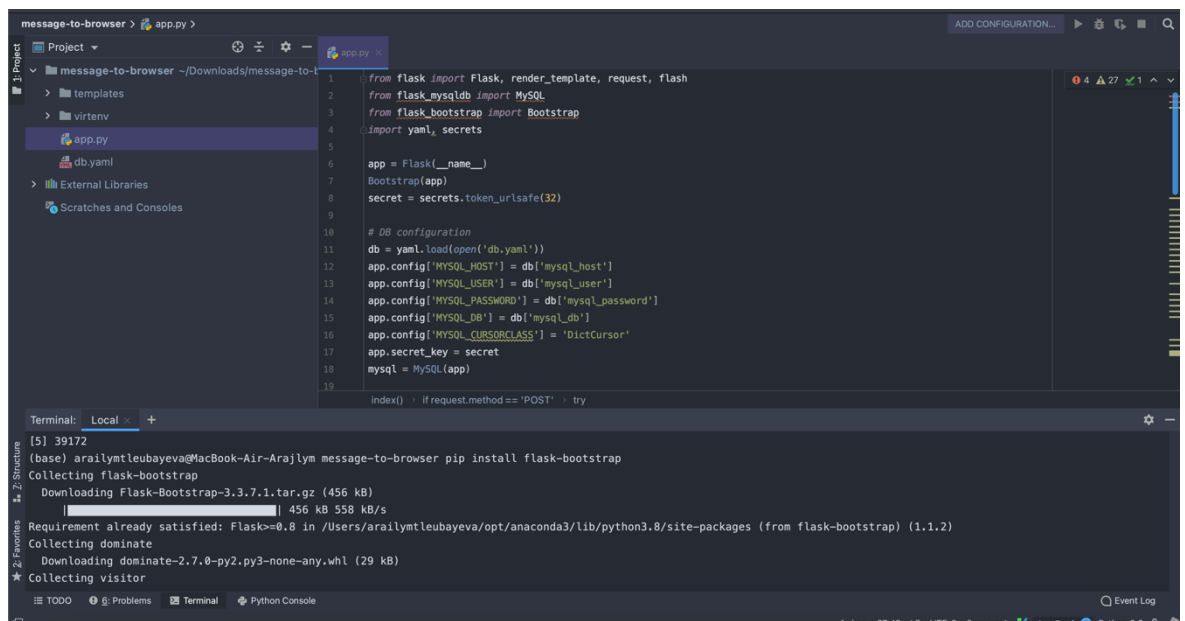
**db.yaml:**

```
mysql_host:'localhost'  
mysql_user:'root'  
mysql_password:'11111111'  
mysql_db:'employee_data'
```

## Task 2 - Browser Messages

### 2- Check for libraries!

**pip install Flask-MySQLdb**  
**pip install Flask bootstrap**



The screenshot shows a code editor with a project named 'message-to-browser'. The file 'app.py' is open, showing the following code:

```
1 from flask import Flask, render_template, request, flash  
2 from flask_mysql import MySQL  
3 from flask_bootstrap import Bootstrap  
4 import yaml, secrets  
5  
6 app = Flask(__name__)  
7 Bootstrap(app)  
8 secret = secrets.token_urlsafe(32)  
9  
10 # DB configuration  
11 db = yaml.load(open('db.yaml'))  
12 app.config['MYSQL_HOST'] = db['mysql_host']  
13 app.config['MYSQL_USER'] = db['mysql_user']  
14 app.config['MYSQL_PASSWORD'] = db['mysql_password']  
15 app.config['MYSQL_DB'] = db['mysql_db']  
16 app.config['MYSQL_CURSORCLASS'] = 'DictCursor'  
17 app.secret_key = secret  
18 mysql = MySQL(app)  
19  
index() if request.method == 'POST' try
```

The terminal output shows the command 'pip install flask-bootstrap' being executed, with the following output:

```
(base) arailymtleubayeva@MacBook-Air-Arajlym message-to-browser pip install flask-bootstrap  
Collecting flask-bootstrap  
  Downloading Flask-Bootstrap-3.3.7.1.tar.gz (456 kB)  
    | 456 kB 558 kB/s  
Requirement already satisfied: Flask<=0.8 in /Users/arailymtleubayeva/opt/anaconda3/lib/python3.8/site-packages (from flask-bootstrap) (1.1.2)  
Collecting dominate  
  Downloading dominate-2.7.0-py2.py3-none-any.whl (29 kB)  
Collecting visitor
```

### Source code: app.py

```
from flask import Flask, render_template, request, session  
from flask_mysql import MySQL  
import yaml  
import os  
  
app = Flask(__name__)  
  
#DB configuration  
db = yaml.load(open('db.yaml'))  
app.config['MYSQL_HOST'] = db['mysql_host']  
app.config['MYSQL_USER'] = db['mysql_user']  
app.config['MYSQL_PASSWORD'] = db['mysql_password']  
app.config['MYSQL_DB'] = db['mysql_db']  
app.config['MYSQL_CURSORCLASS'] = 'DictCursor'  
mysql = MySQL(app)  
  
app.config['SECRET_KEY'] = os.urandom(24)
```



```

@app.route('/', methods=['GET', 'POST'])
def index():
    if request.method == 'POST':
        form = request.form
        name = form['name']
        age = form['age']
        cursor = mysql.connections.cursor()
        cursor.execute("INSERT INTO employee(name, age) VALUES(%, %s)",
            (name, age))
        mysql.connections.commit()
        return render_template('index.html')

@app.route('/employees')
def employees():
    cursor = mysql.connections.cursor()
    result_value = cursor.execute("SELECT * FROM employee")
    if result_value > 0:
        employees = cursor.fetchall()
        session['username'] = employees[0]['name']
        return render_template('employees.html', employees=employees)

if __name__ == '__main__':
    app.run(debug=True)

```

**db.yaml:**

```

mysql_host: 'localhost'
mysql_user: 'root'
mysql_password: '11111111'
mysql_db: 'employee_data'

```