# C1 - Python

C-COD-140

# Flask Day01

Introduction to Flask

1.3

# Flask Day01

**repository name:** flask_d01
**repository rights:** ramassage-tek

- The totality of your source files, except all useless files (binary, temp files, obj files,...), must be included in your delivery.
- Error messages have to be written on the error output, and the program should then exit with the 84 error code (0 if there is no error).

## Foreword

Hi !

Welcome to Flask, a micro web framework for python. It's a very light framework but a very efficient one as well.

For this introduction we are going to build what we call a micro service, a login / registration microservice to be cristal clear.

A micro service is a tiny bit of an API that does only a few things, a entire website can be divided in several microservices.

So yes if you do it right today you won't have to make another login/registration service until the end of the formation. So let's do it right shall we?

As Flask is a light framework let's use a light sql language: **SQLite3**
Here is some documentation:

- On microservice
- On SQLite3
- On Flask
- On the devastating states of my brothers

## Exercise 01 (1pt)

**File to hand in:** flask_d01

First let's see how powerful and easy to use flask is.

```
Terminal
~/C-COD-140> python3 -m pip install flask
```

This command will install Flask (you don't say).

You might have to install **pip3** as well if you don't have it already.

Now let's create a python (app.py) file as such:

```
Terminal
~/C-COD-140> cat app.py
from flask import Flask
app = Flask(__name__)   # means that your app name will be the same as the file

@app.route('/')      # yeah this is a route, nicer than in Symfony, isn't it ?
def index():
    return ("My first page")
```

The function defined right after the route is the function called when the route is called, so, this is your router/dispatcher. The route notation is a python capability. It is named **decorator**. Go and grab some information on it.

## Exercise 02 (1pt)

**File to hand in:** flask_d01

OK, that was intense.

Let's run some command line now, and by "some", I mean two:

```
~/C-COD-140> export FLASK_APP=app.py export FLASK_DEBUG=true
~/C-COD-140> flask run
```

The first line tells flask that the entry point of your web app is going to be that file.
The second part of the first line tells Flask to enable the debug mode.

This **export** command is linked to your terminal, if you close it you'll need to run those commands again.

About the second command, you know, it's going to run the app.

Let's visit **localhost:5000** and here is your first page.

Now everything is sweet and all, but we have to make that stuborn database...

## EXERCISE 03 (2PTS)

**File to hand in:** flask_d01

Let's use SQLite3 now, I'll let you check on your own the differences between MySQL and SQLite3.
But for now let's see how to link it with your Flask application.

First you need to make a file **schema.sql**. In it you'll need to put your SQLite3 code.
We need 1 table: users. Composed as such: id (int), username (text), email (text), password (text)

Alrighty, now on your **app.py** file you'll need to add some stuff:

```
Terminal
~/C-COD-140> cat app.py
import os
import sqlite3
from flask import Flask, request, session, g, redirect, url_for, abort,
render_template, flash
app.config.from_object(__name__)    # load config from this file app.py

# Load default config and override config from an environment variable
app.config.update(dict(
    DATABASE=os.path.join(app.root_path, 'flaskr.db'),
    SECRET_KEY='development key',
    USERNAME='admin',
    PASSWORD='default'
))
app.config.from_envvar('FLASKR_SETTINGS', silent=True)
```

This is how you configure your database user, keep it like this for now.

Now you need to make 6 functions after this:

- connect_db()
- init_db()
- initdb_command()
- get_db()
- query_db()
- close_db()

This you cannot guess, but as I'm really bored of copying / pasting I'll let you find them on the official documentation of Flask. OK nice, to make a query to your database you will use something like this:

```
Terminal
db = get_db()
query = db.execute('SQLITE CODE')
db.commit() / query.fetchall()
```

## Exercise 04: Login and registration (8pts)

**File to hand in:** flask_d01

**Register:** (3pts)

Now, let's make the registration route. You will need to take two http methods **GET** & **POST** on the same route.

For the **POST** method you will have to do the register mechanics: hash the password and insert the user into the db. You do no need to check the validy of the fields just yet.

And for the "GET" method you will need to render the view (registration form) and the action will aim at "/register" with a "POST" method.

return render_template('register.html') # <= your views files will have to be in a *templates* folder.

Look at the template language of flask and how to write it.

**Login:** (3pts)

The login methods now, as the register you will need two methods on the same route: **/login**

Do the login.
You do not need to makes sessions yet.
Just check the validity of the credentials inserted.

The goal of an API is to be accessible from any language as long as it use an HTTP methods.

**Testing with other languages:** (2pts)

Use three different languages (python, php, c, ruby ...) to make an API call to your API (on '/register' for example) whithout using the form from the template (look at Postman for clues).
Put the three scripts in a specific folder.

import crypt, Postman

## EXERCISE 05: MVC AND CRUD (4PTS)

**File to hand in:** flask_d01

**A bit of MVC:** (2pts)

Now let's clean it up a tiny bit.

You will make a **controller.py** file and a **model.py** file.

Your route will call the right method of the controller and the controller will call the right method from the model. And then the result will be return to the controller whish will return it to the **app.py** file, and then display an adequate message to the view (check flash messages in flask).

When you call your controller method it will need to take the db and the request variables as parameters.

**CRUD:** (2pts)

Let's finish the CRUD for the users.

We will add a route to **update** ("PUT" method) to **destroy** ("DELETE" method) and to **display an user** ("GET" method). Those three methods needs to be on the same route. You will have to specify the user id that you want to update/destroy/display on the URL of the request, check how to put a variable in the url and how to retrieve it.

Find a way, on the same route, to add a **GET** method to retrieve all the users.
Do NOT make forms, test the routes with Postman.

## Exercise 06: API (4pts)

**File to hand in:** flask_d01

Final blow !!

An API doesn't render anything (you saw it with the queries that you made with other languages) so you need to return json content. Change your code a little bit so that every route returns json code (except the GET method of login and register of course)

To go a little further make another (client.py) flask app that will run on it's own server (Yes you will have two servers running at the same time). This app will not have any db, it will only render the json data of your Other flask project (the CRUD user one).

To launch a Flask app on a different port you need to type:

Terminal

```
∼/C-COD-140> flask run -port 3000
```

This is to specify that this one is not going to run on the port 5000 (the default port) but on the port **3000**.

You have to be able to show all users, show a specific user and delete a user in a html file.
Remember no db allowed on the client app.

jsonify, json.loads()

## BONUS: MICROSERVICES (3PTS)

**File to hand in:** flask_d01

Make another flask app that connects to a different database but implements the CRUD for products. So you will have three servers running; one for the login / register, one for the CRUD products and one that acts as the client and that renders template hmtl files.

The products table will have a userId representing the user that owns it.

And this, my friends, is what we call **microservices.**