

# **IBM Cloud: Create, Integrate & Deploy**

# Agenda

- Introduction
- Call for Code
- Python
- Flask
- Visual Recognition
- Tutorial – Flask App Building



Davide Pisoni  
IT Analyst



**CALL FOR CODE**  
GLOBAL INITIATIVE 2018

Commit to the cause. Push for change.

**Call for Code** inspires developers to solve **pressing global problems** with **sustainable software solutions**, delivering on their vast potential to do good.

Bringing together NGOs, academic institutions, enterprises, and startup developers to compete build effective **disaster mitigation solutions**, with a focus on health and well-being.

### Quick-Links:

[CallforCode FAQ](#)

[CallforCode Resources](#)

<https://developer.ibm.com/callforcode/>

Award winners will receive **long-term support** through **The Linux Foundation**, **financial prizes**, the **opportunity to present their solution to leading VCs**, and will deploy their solution through **IBM's Corporate Service Corps**.

Developers will jump-start their project with dedicated **IBM Code Patterns**, combined with **optional enterprise technology** to build projects over the course of three months.

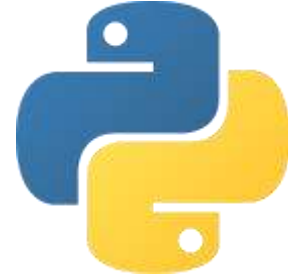
Judged by the world's most **renowned technologists**, the **grand prize** will be presented in **October** at an Award Event.

**LAUNCHED MAY 2018!**

# Python: Introduction

Python is a high-level, interpreted, interactive and object-oriented scripting language, designed to be highly readable.

Python was developed by Guido van Rossum in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands. Now is maintained by a core development team at the institute, although Guido van Rossum still holds a vital role in directing its progress.



Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

## Features

- Python code is more clearly defined and visible to the eyes.
- Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
- Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.
- It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.
- Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- Python provides interfaces to all major commercial databases.
- It can be used as a scripting language or can be compiled to byte-code for building large applications.
- It provides very high-level dynamic data types and supports dynamic type checking.

# Python: Syntax

## Variables

```
1 #Variables
2
3 x=12.3
4
5 str='string'
6
7 list=[1,2,1,'str']
8
9 dict={'a':1,'b':2}
```

## Loops

```
24 #Loops
25
26 v for i in 1:10:
27     print(i)
28
29 v while(i>2):
30     print('Hello')
```

## Functions

```
32 #Functions
33
34 v def test_function(arg1,arg2):
35     print(arg1+arg2)
36
```

## Operations

```
31 #Operations
32
33 x=1+2
34
35 x=2/3
36
37 str_all='str'+ing #Results->'string'
38
39 list[0] #Results->2
```

## Import Libraries

```
38 #Import Libraries
39
40 from Flask import Flask
41 import os
42
```

# Flask

A **Web Application Framework** or simply Web Framework represents a collection of libraries and modules that enables a web application developer to write applications without having to bother about low-level details such as protocols, thread management, etc...

**Flask** is a web application framework written in Python. It is developed by **Armin Ronacher**, who leads an international group of Python enthusiasts named Pocco. Flask is based on the Werkzeug WSGI toolkit and Jinja2 template engine. Let's see them in details:

## WSGI

Web Server Gateway Interface (WSGI) has been adopted as a standard for Python web application development. WSGI is a specification for a universal interface between the web server and the web applications.

## Werkzeug

It is a WSGI toolkit, which implements requests, response objects, and other utility functions. This enables building a web framework on top of it. The Flask framework uses Werkzeug as one of its bases.

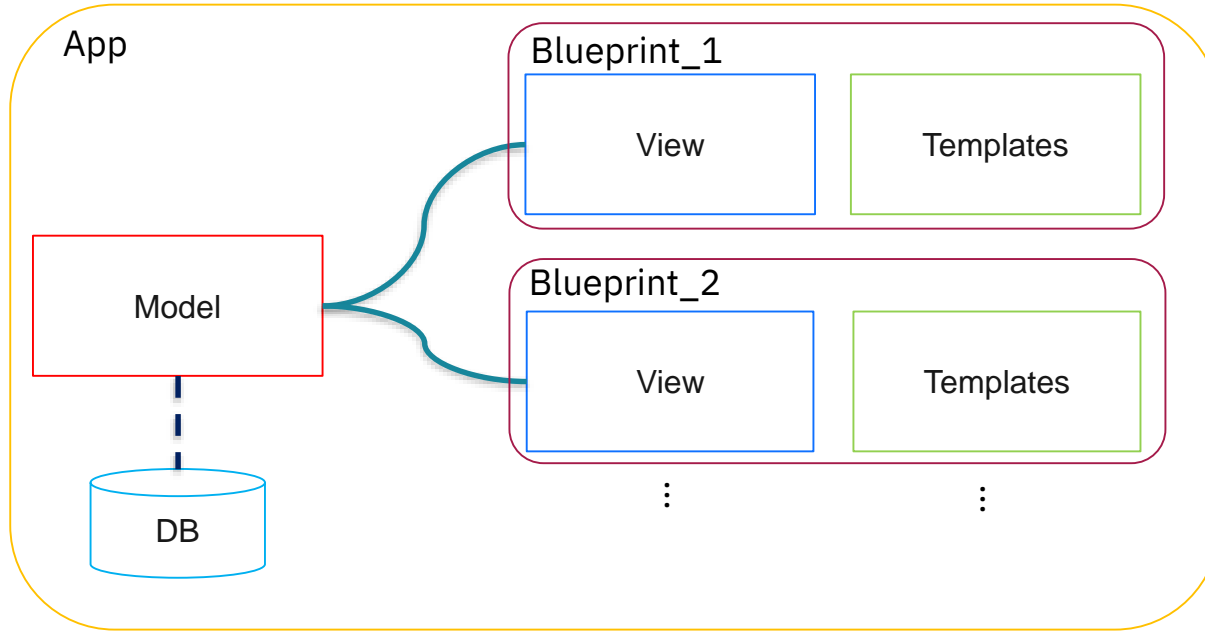
## jinja2

jinja2 is a popular templating engine for Python. A web templating system combines a template with a certain data source to render dynamic web pages.

Documentation: <http://flask.pocoo.org/>

# Flask App Schema

A flask web app is based on a core (a Flask object), that initialize the app.



The core interacts with three main concepts: View, Templates and Model.

- **View:** views are functions that take care of the backend, routes and to render the templates of each page of your app
- **Templates:** templates are the dynamic structure of each page
- **Model\*:** is the schema of the database where views can recover data

A view and a templates create a Blueprint, that can be registered in the core and call on each request from users' browsers.

# Watson Visual Recognition

One of the Watson API is Watson Visual Recognition, an IBM Cloud Service to classify images and create custom classifier

- **Api Reference:**

<https://www.ibm.com/watson/developercloud/visual-recognition/api/v3/curl.html?curl>

- **Documentation:**

<https://console.bluemix.net/docs/services/visual-recognition/getting-started.html#getting-started-tutorial>





# **Tutorial**

# **Flask App on**

# **IBM Cloud**

Today we will create a simple  
**Flask app** to classify the  
images through the **Watson  
Visual Recognition Service**

# Prerequisites

1. Python Version  $\geq$  3.6
2. IBM Cloud Account (<https://console.bluemix.net/registration/>)
3. CLI Command Line  
([https://console.bluemix.net/docs/cli/reference/bluemix\\_cli/get\\_started.html#getting-started](https://console.bluemix.net/docs/cli/reference/bluemix_cli/get_started.html#getting-started))

# Instructions: Flask Image Recognition

1. Create Watson Visual Recognition and Watson Studio Services in IBM Cloud
2. Create a local Flask App
3. Integrates Visual recognition in the app through the python sdk
4. Deploy the app on IBM Cloud

# IBM Cloud Dashboard

The screenshot displays the IBM Cloud Dashboard interface. At the top, there is a navigation bar with the IBM Cloud logo and menu items: Catalogo, Documenti, Supporto, Gestisci, and a user profile icon. Below the navigation bar, the dashboard title 'Dashboard' is followed by several filters: 'VERSIONI IN USO' (default), 'REGIONE' (Germany), 'ORGANIZZAZIONE' (cloud-paas@DE.ibm.com), and 'SPAZIO CLOUD FOUNDRY' (Europe). A search bar for resources and a 'Crea nuovo cluster' button are also present.

The main content area is divided into three sections:

- Applicazioni Cloud Foundry:** 1000 DEU GB utilizzate. This section contains a table with columns: Nome, Rete, Memoria (MB), and Stato. It lists three applications: CooTech, Scuola, and UniBresciaLesson, all in 'In Esecuzione (0%)' state.
- Cluster:** This section contains a table with columns: Nome, Ubicazione, Nodi, Versione Kube, and Stato. It lists one cluster named 'Hacloster' located in 'eu-de-par01' with 1 node and Kube version '1.7.4\_1003', currently in 'Pronto' state.
- Servizi:** 6/10 utilizzati. This section contains a table with columns: Nome, Offerta di servizi, and Piano. It lists seven services: Conversation-v1, CooTech-cloudantNoSQLDB, Personality Insights-c0, Scuola-cloudantNoSQLDB, Tone Analyzer-03, and UniBresciaLesson-cloudantNoSQLDB, each with its respective service offering and plan.

# Create Visual Recognition Service(1/2)

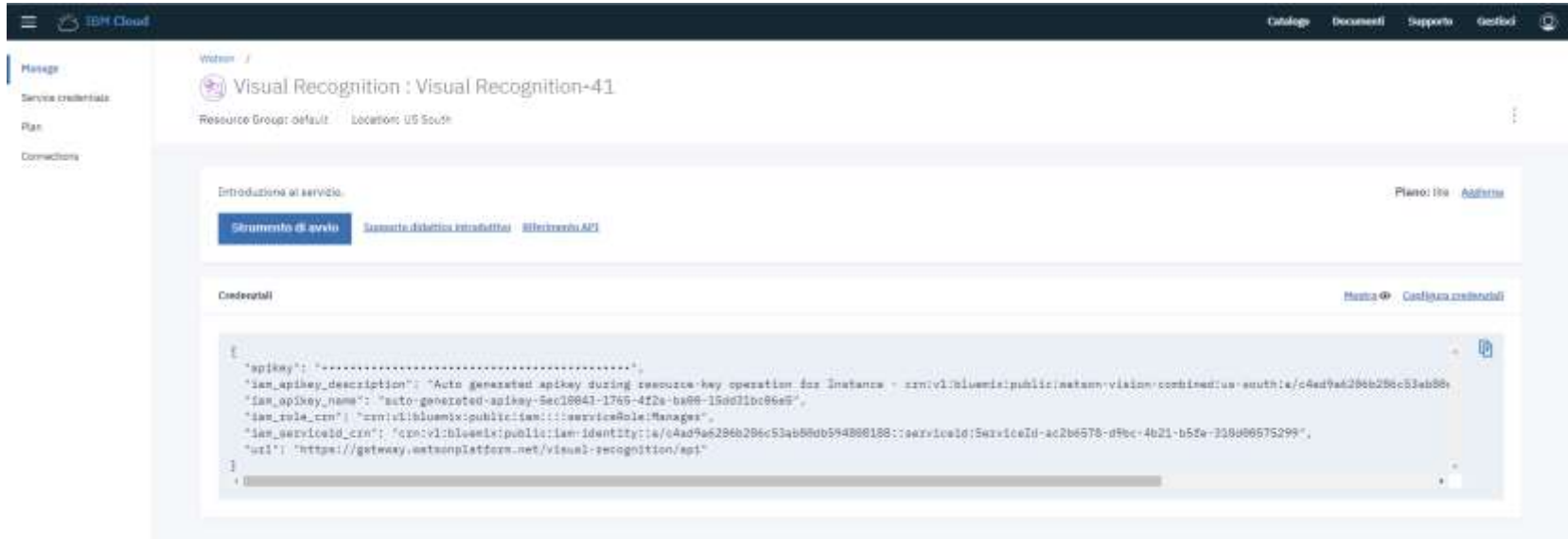
Login in the IBM Cloud, go to **catalog** and select in the Watson API the **Visual Recognition Service**

The image shows a grid of 12 service cards from the IBM Watson API catalog. Each card includes an icon, a title, a brief description, and 'Like' and 'IBM' buttons. The 'Visual Recognition' card is circled in red.

| Service Name                   | Description   | Like | IBM |
|--------------------------------|---|------|-----|
| Watson Assistant               | Aggiungere un'interfaccia di linguaggio naturale alle applicazioni per automatizzare le interazioni con i clienti.  | Like | IBM |
| Knowledge Studio               | Crea modelli personalizzati per insegnare a Watson la lingua del proprio dominio.   | Like | IBM |
| Natural Language Classifier    | Natural Language Classifier effettua la classificazione del linguaggio naturale nel testo delle domande. <a href="#">Ulteriori informazioni</a>                       | Like | IBM |
| Speech to Text                 | Trascrizione in streaming a bassa latenza.  | Like | IBM |
| Visual Recognition             | Trova il significato di un contenuto visivo. Analizza le immagini alla ricerca di scene, oggetti, volti e altro. <a href="#">Ulteriori informazioni</a>               | Like | IBM |
| Discovery                      | Aggiunge un motore di analisi del contenuto e di ricerca cognitiva alle applicazioni.   | Like | IBM |
| Language Translator            | Tradurre il testo da una lingua ad un'altra per specifici domini.   | Like | IBM |
| Natural Language Understanding | Analizza il testo per estrarre i metadati dal contenuto come ad esempio concetti, entità, sentimenti, relazioni. <a href="#">Ulteriori informazioni</a>               | Like | IBM |
| Text to Speech                 | Sintetizza del testo in parlato dal vostro network.   | Like | IBM |
| Watson Studio                  | Embed AI and machine learning into your business. Create custom models using your own data.   | Like | IBM |
| Knowledge Catalog              | Discover, catalog, and securely share enterprise data.  | Like | IBM |
| Machine Learning               | IBM Watson Machine Learning - prendi decisioni più intelligenti, risolvi problemi difficili e migliori i processi. <a href="#">Ulteriori informazioni</a>             | Like | IBM |
| Personality Insights           | Watson Personality Insights desume le informazioni dai dati dei media sociali e professionali per identificare la personalità. <a href="#">Ulteriori informazioni</a> | Like | IBM |
| Tone Analyzer                  | Tone Analyzer utilizza l'analisi linguistica per rilevare le toni di toni delle comunicazioni, emozioni, sentiment. <a href="#">Ulteriori informazioni</a>            | Like | IBM |

# Create Visual Recognition Service(2/2)

Then click on create: if everything goes well you'll be redirected on the page of your service



The screenshot shows the IBM Cloud console for a Visual Recognition service instance named 'Visual Recognition-41'. The page is in Italian. It features a navigation sidebar on the left with options like 'Manage', 'Service credentials', 'Plan', and 'Connections'. The main content area has a header with the service name and location (US South). Below this, there's a section titled 'Introduzione al servizio' (Introduction to the service) with a 'Piano: lite' (Plan: lite) button and an 'Addizionali' (Add-ons) link. A 'Strumenti di avvio' (Startup tools) button is also present. The 'Credenziali' (Credentials) section is expanded, showing a JSON object with the following fields:

```
{
  "apikey": ".....",
  "iam_apikey_description": "Auto generated apikey during resource-key operation for Instance - crn:v1:bluemix:public:atson-visual-recognition:us-south:4/c4ed9a6286b28c53ab8b",
  "iam_apikey_name": "auto-generated-apikey-Sec18841-1765-4f2a-ba98-15d81bc86e2",
  "iam_role_crn": "crn:v1:bluemix:public:iam:::serviceRole:Manages",
  "iam_serviceid_crn": "crn:v1:bluemix:public:iam-identity::4/c4ad9a6286b28c53ab8b0b594888188::serviceid:ServiceId-ac2b6578-dfbc-4b21-b50e-238d866575299",
  "url": "https://gatewayatsonplatform.net/visual-recognition/api"
}
```

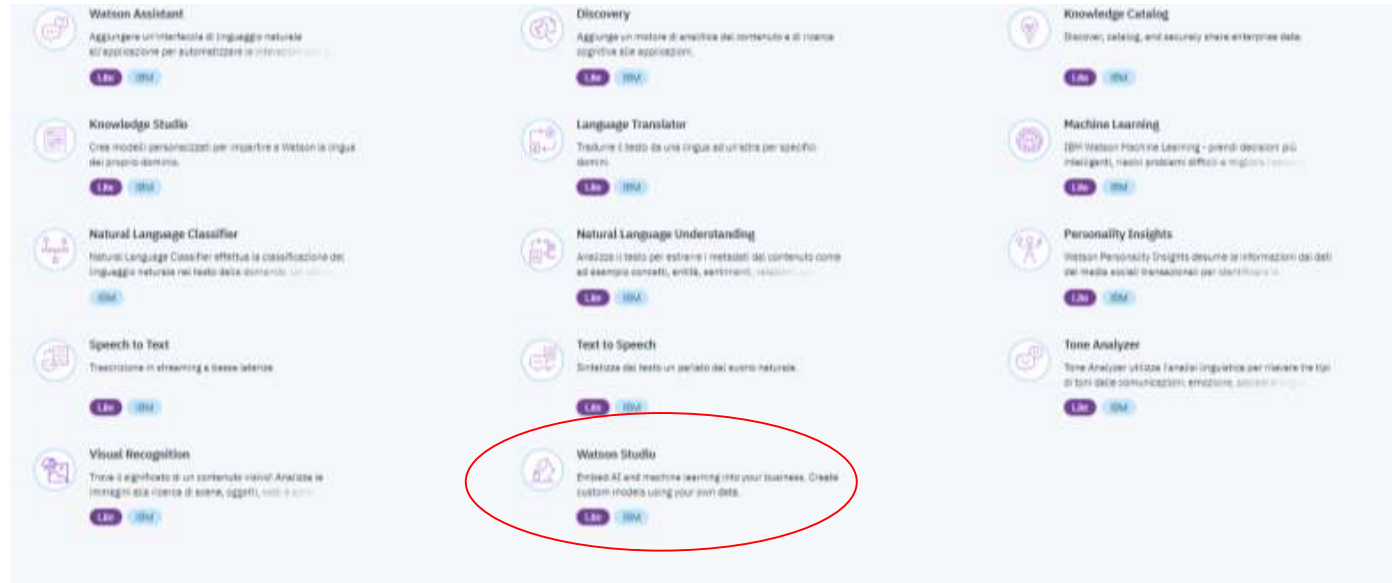
Here you can see the service credentials (in particular the API\_KEY), used to call the service from the python SDK

# Create Watson Studio Service

Login in the IBM Cloud, go to **catalog** and select in the Watson section the **Watson Studio**

## Watson Studio

provides a suite of tools and a collaborative environment for data scientists, developers and domain experts. Trough this service, you will be able to use the user interface of Visual Recognition and create custom cluster.





# First Step

1. Create a Folder for our app (e.g. Flask\_App)
2. Open the command line (cmd) and navigate to the folder of your app (*cd path\_to\_your\_folder*)
3. Using the virtualenv library pre-installed in python >=3.0, create a virtual environment to work in, by using the following command in the cmd: *python -m venv venv*
4. Activate the virtual env with the following line: *venv\Scripts\activate* (to deactivate it you can use the command *deactivate*)

```
<venv> C:\Users\IBM_ADMIN\Desktop\Lessons\Python\Project_Flask_Image_Recognition>
```

5. Once the venv is on (you will see *venv* at the begging of the line), we can install the libraries that we will need with pip:
  - *Pip install Flask*
  - *Pip install watson\_developer\_cloud* (this is the python sdk to use Watson API)\*

\*In case you struggle with Watson Developer Cloud installation, download the right dependencies from here: <https://www.lfd.uci.edu/~gohlke/pythonlibs/> and install them manually with *pip install file.whl*

# Flask App

Now we can start creating a basic Flask app with a super simple homepage

1. Create a folder called *project*, that will contains all our pages, and inisde it create three folders called *home*, *static* and *templates*, and a file named `__init__.py`, this file initialize the app.
2. Inside the home folder, create a *templates* folder and two files, one called *views.py* and one `__init__.py` (this file remains empty and just state that python can go inside this folder to get function or variables). Here is the tree:

```
project/  
  static/  
  templates/  
  home/  
    templates/  
    views.py  
    __init__.py  
  __init__.py
```

This way we create our structure: *static* folder contains static files, like css or images to call in the templates, *templates* folder contains a base html, sort of standard format for all our pages, and *home* folder is our homepage.

# Initialize App

Inside the `__init__.py` write the following code:

```

    __init__.py
1  from flask import Flask
2  import os
3
4  #####
5  #### config ####
6  #####
7  app = Flask(__name__)
8  app.config.from_object('config.DevelopmentConfig')
9
10 from project.home.views import home_blueprint
11
12 # register our blueprints
13
14 app.register_blueprint(home_blueprint)
15
```

config.py

Import Flask library

Initialize the app class

Select the configuration for the app (see next slide)



Here we are registering the pages of our site: blueprint help us to too link Views and templates, and by registering them, we let the app know where and what they are. **Every time you create a new page, remember to register its blueprint**

# Config.py

In the slide before we initialize the app with a configuration that come from a file inside outside the project folder called *config.py*: this file contains the configuration of the app, global variable, static folder, etc....

```
__init__.py | config.py
1  import os
2
3
4  #default config
5
6
7  class BaseConfig(object):
8      DEBUG = False
9      SECRET_KEY=' \x8dBI\x9b\xbf\r\xd0t\xb5I\x86\x89]\xe4\x03~\x88\x8e\xac\xcb\xa5\xc7\xbfz'
10     UPLOADED_PHOTOS_DEST='./project/static/images'
11
12     class DevelopmentConfig(BaseConfig):
13         DEBUG = True
14
15     class ProductionConfig(BaseConfig):
16         DEBUG = False
17
```

Here we create a BaseConfig object class, with some global variables set, and two other config objects (Development Config and Production Config) that extend it.

Global Variables can be:

*Debug*: to set the debug on or off

*Secret Key*: used to secure your session

Many others can be found in Flask documentation

# Base.html and Bootstrap.css

Thanks to jinja2 we can use a sort of standard layout for all our pages and extend it when necessary. To do so we create a *base.html* file in the templates folder inside project

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Image Recognition App</title>
5     <meta name="viewport" content="width=device-width, initial-scale=1.0">
6     <link href="static/bootstrap.min.css" rel="stylesheet" media="screen">
7     <link href="static/mycss.css" rel="stylesheet" media="screen">
8   </head>
9   <body>
10
11     <div class="container">
12
13       <!-- child template -->
14       {% block content %}{% endblock %}
15
16       <!-- errors -->
17       {% if error %}
18         <p class="error"><strong>Error:</strong> {{ error }}</p>
19       {% endif %}
20
21       <!-- messages -->
22       {% for message in get_flashed_messages() %}
23         {{ message }}
24       {% endfor %}
25     </div>
26
27   </body>
28 </html>
```

This way we can set and use css or js script globally in all our templates, insert our child templates, catch and handle error or flash messages.

For example we can download the bootstrap css style: <https://getbootstrap.com/>

Put it inside the static folder and call it from the *base.html* file to set it globally

# Home - Views.py

Now we can start creating our first view for our homepage: inside the views.py file we define the blueprint to link view and templates, the route to access this view and the backend function for the view

```
__init__.py  config.py  views.py
1  #####
2  """ imports """
3  #####
4
5  from flask import render_template, Blueprint
6
7  #####
8  """ config """
9  #####
10
11 home_blueprint = Blueprint(
12     'home', __name__,
13     template_folder='templates'
14 ) #pragma: no cover
15
16 #####
17 """ routes """
18 #####
19
20 <
21 # use decorators to link the function to a url
22 @home_blueprint.route('/', methods=['GET', 'POST'])
23 def home():
24     error = None
25     return render_template(
26         'index.html', error=error)
27
28
```

As always we import the library, we define the Blueprint home, to connect the View with the templates folder to get the html files, and at the end we define the route.

The routes are the different URLs that the application implements. In Flask, handlers for the application routes are written as Python functions, called view functions.

View functions are mapped to one or more route URLs so that Flask knows what logic to execute when a client requests a given URL.

# Home - templates

Now that we have set our view, we can create our *index.html* in templates folder to render it and decide what to show users: we extend the *base.html* created before with just a title in the page

```
1 {% extends "base.html" %}
2 {% block content %}
3
4 <h1>Image Recognition App</h1>
5
6
7 {% endblock %}
8
```

# Run!

We are ready to run our app locally: to do so we need to create our *run.py* file to fire up the server. We import our app and we set the host and the port to run the app. Put this file outside the project folder.

```
run.py
1  from project import app
2  import os
3
4  port = os.getenv('PORT', '5000')
5  if __name__ == "__main__":
6      app.run(host='0.0.0.0', port=int(port))
7
```

Once done, inside your virtual env you can type *python run.py* and go to *localhost:5000* in your browser to see your app running!



# Integrates Watson Visual Recognition

To integrate Watson Visual Recognition, we create a function to classify images calling the service and modify the templates and the view inside home to let the users upload an image, select the classifier and classify it.

Let's start by creating the function to call the service and classify an image:

1. Create a new function folder *function* inside project
2. Inside create a file `image_recognition.py`

```
image_recognition.py
1 from watson_developer_cloud import VisualRecognitionV3
2 import json
3
4 visual_recognition = VisualRecognitionV3(
5     version='2018-03-19',
6     iam_api_key='apy_key'
7 )
8
9 def classify_image(path, classifier):
10     with open(path, 'rb') as images_file:
11         classes = visual_recognition.classify(
12             images_file,
13             parameters = json.dumps({
14                 'classifier_ids': [classifier]
15             })
16         )
17     return(json.dumps(classes, sort_keys = True, indent = 2, separators = (',', ': ')))
```

First we initialize the service: replace `api_key` with the one inside the credentials of the service on IBM Cloud.

Then we create a function that takes a path of an image and a classifier name, give them to the `visual_recognition` instance and return a json with the results

# Forms

To upload an image and select the classifier we will use a form: Flask handle forms with a specific library called *flask-wtf*. So first of all install the library inside our virtual env: *pip install flask-wtf*

Then inside our home folder create a new file called *forms.py* where we initialize field in our form: a File field to upload image and a SelectField to choose between a list of classifier:

```
forms.py
1  from flask_wtf import FlaskForm
2  from wtforms import SelectField
3  from wtforms.validators import DataRequired,ValidationError
4  from flask_wtf.file import FileField,FileAllowed, FileRequired
5
6  |
7
8  class PhotoForm(FlaskForm):
9      classifier = SelectField('classifier', validators=[
10         DataRequired(message="Select a classifier please")],
11         choices=[('default','default'),('explicit','explicit'),('food','food')])
12
13     photo = FileField('photo', validators=
14         [FileRequired(message='Please load a photo'),
15         FileAllowed(['jpg','png'], message='Select a jpg or a png photo')])
16
```

Import library, initialize our Form Class and create the two fields (one to select the classifier and one to upload file), with a series of validators to check that the user uploaded a file and the correct type of file.

Flask-Wtf contains a long list of possible field, allowing you to create every type of form

# Handle Forms

Once created the form, we need to adjust our view file to handle it and modify the html to render it.

In the view file we just import the form class on top: *from project.home.forms import PhotoForm*

Then inside the view function we initialize the form by declaring it and add it in the variable passed to the templates: *form=PhotoForm()* and then *return render\_template('index.html',form=form, error=error)*

Your view function should look like this:

```
21 @home_blueprint.route('/', methods=['GET', 'POST'])
22 def home():
23     error = None
24     form=PhotoForm()
25     return render_template(
26         'index.html',form=form, error=error)
27
```

# Form in the templates

Now we modify the *index.html* file to render the form: so we create a form tag and inside it we put the two fields of our form.

Thanks to jinja2 we can call the element sent from the view with `{{...}}` and handle errors coming from validation error.

The `csrf_token` is used by Wtfform to validate forms.

At the end of the form we place a button to submit the inserted value and file.

Once done you can run your app and see your new form. If you press the button Classify as you can see nothing happen, because we still have to handle the post request coming from the form and upload the image.

```
index.html
1  {% extends "base.html" %}
2  {% block content %}
3
4  <h1>Image Recognition App</h1>
5  <br>
6  <form class="photo-form" role="form" method="post" action="" enctype="multipart/form-data">
7
8      {{ form.csrf_token }}
9      <p>
10         Image : {{ form.photo(class_='form-control',placeholder='Select an image') }}
11         <span class="error">
12             {% if form.photo.errors %}
13                 {% for error in form.photo.errors %}
14                     {{ error }}
15                 {% endfor %}
16             {% endif %}
17         </span>
18     </p>
19     <p>
20         Classifier : {{ form.classifier(class_='form-control') }}
21         <span class="error">
22             {% if form.classifier.errors %}
23                 {% for error in form.classifier.errors %}
24                     {{ error }}
25                 {% endfor %}
26             {% endif %}
27         </span>
28     </p>
29     <button class="btn btn-lg btn-success" type="submit" name="submit">Classify</button>
30 </form>
31 <br>
32
33 {% endblock %}
```

# Post request and Upload Images

To handle post request in our view we need to insert a check if the methods used to call the page is POST. After that we check if the form is validated and, if so, we can upload the image server side, classify it and show to the user the result.

So we import our function to classify images views.py file and add *request* to Flask import :

```
from project.function.image_recognition import classify_image
```

Then we modify our view file like so:

```
22 @home_blueprint.route('/', methods=['GET', 'POST'])
23 def home():
24     error = None
25     form=PhotoForm()
26     if(request.method == 'POST'):
27         if(form.validate_on_submit()):
28             filename=form.photo.data.filename
29             form.photo.data.save('./project/static/images/'+ filename)
30             data=classify_image('./project/static/images/'+ filename,form.classifier.data)
31             return render_template('index.html',form=form,data=data,filename=filename,error=error)
32     return render_template(
33         'index.html',form=form, error=error)
```

We save our uploaded image in an images folder that we need to create in static.

Then we pass the path to the image and the classifier selected to our function to classify images.

At the end we sent to the template the photo name and the results from the classification

# Render images and results

Finally we can render the image uploaded by the users and the results by adding this line to the *index.html*:

```
33  {%if filename is defined%}
34  </img>
35
36
37  <div class="results">
38      <pre>{{ data }}</pre>
39  </div>
40  {%endif%}
```

As you can see the results is not formatted well in the page: we can remedy by adding a css file in the static folder and add it in the base.html. Css file are used to style html pages. Here is an example of a *mycss.css* file:

```
1  pre{
2    height: 26.5pc;
3    overflow-y: scroll;
4    margin-left: 52%
5  }
6
7  img{
8    float:left;
9    width:50%;
10
11 }
```

# Deployment: File Required

Prima di fare il deploy dell'applicazione serve aggiungere i seguenti file all'applicazione:

- Manifest.yml: generalità (ram, buildpack,etc...) e nome della nostra applicazione

*applications:*

*- path: .*

*buildpack: python\_buildpack*

*memory: 128M*

*instances: 1*

*Command: python run.py*

*name: Flask\_app*

*disk\_quota: 1024M*

- ProcFile: codice per far avviare la nostra applicazione quando sarà caricata sul cloud

*web: python run.py*

- Requirments.txt: un file testo contenente tutte le librerie python da installare per far funzionare l'applicazione. Inside the virtual env use: *pip freeze > requirements.txt* (check wether the file contains dot...)
- Runtime.txt: un file testo contenente la versione di python da usare per l'applicazione (consultare quelle possibili [https://console.bluemix.net/docs/runtimes/python/index.html#python\\_runtime](https://console.bluemix.net/docs/runtimes/python/index.html#python_runtime))

# Deployment: App Push

Now we are ready to deploy our app on IBM Cloud:

1. Navigate on your app folder `Flask_App` with the cmd
2. Use command `bluemix login` to login to IBM Cloud
3. Select the space and the region when prompt
4. Use `bluemix target --cf` to set the org and the space
5. Finally `bluemix app push name_of_the_app`

If the stage process goes well you will see your app on your dashboard: you can click on it and click on the link `Visit_URL`

**Your app is online!**



# Thank You

For any questions, email me: [davide.pisoni@it.ibm.com](mailto:davide.pisoni@it.ibm.com)

