IBM Cloud:

Create, Integrate & Deploy

 © 2017 IBM Corporation

Agenda

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

Davide Pisoni

IT Analyst

04/12/2017 © 2017 IBM Corporation

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.

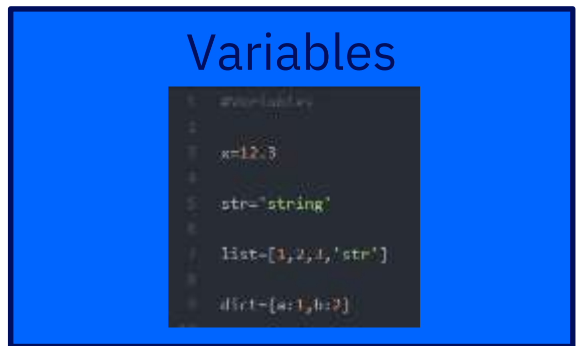
Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

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



Loops

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

Functions

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

Operations

```
#Operations

x=1+2

x=2/3

str_all='str'+'ing' #Results-->'string'

list[0] #Results-->1
```

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

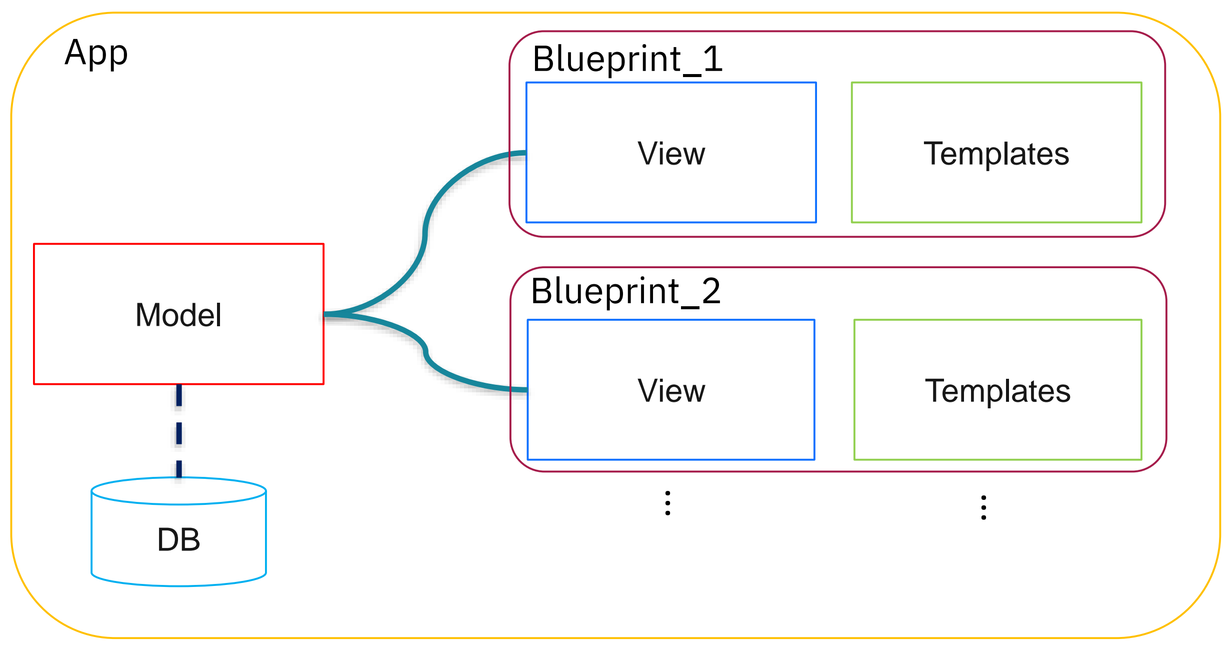
jinja2 is a popular templating engine for Python. A web templating system combines a template with a certain data source to render dynamic web pages.

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

 © 2017 IBM Corporation

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.

04/12/2017 © 2017 IBM Corporation

* During this course, model interaction and costruction won't be presented

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 crate a simple **Flask app** to classify the images trough the **Watson Visual Recognition** Service

 © 2017 IBM Corporation

Prerequisites

1. Python Version >= 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)

04/12/2017 © 2017 IBM Corporation

Istructions: Flask Image Recognition

1. Create Watson Visual Recognition and Watson Studio Services in IBM Cloud

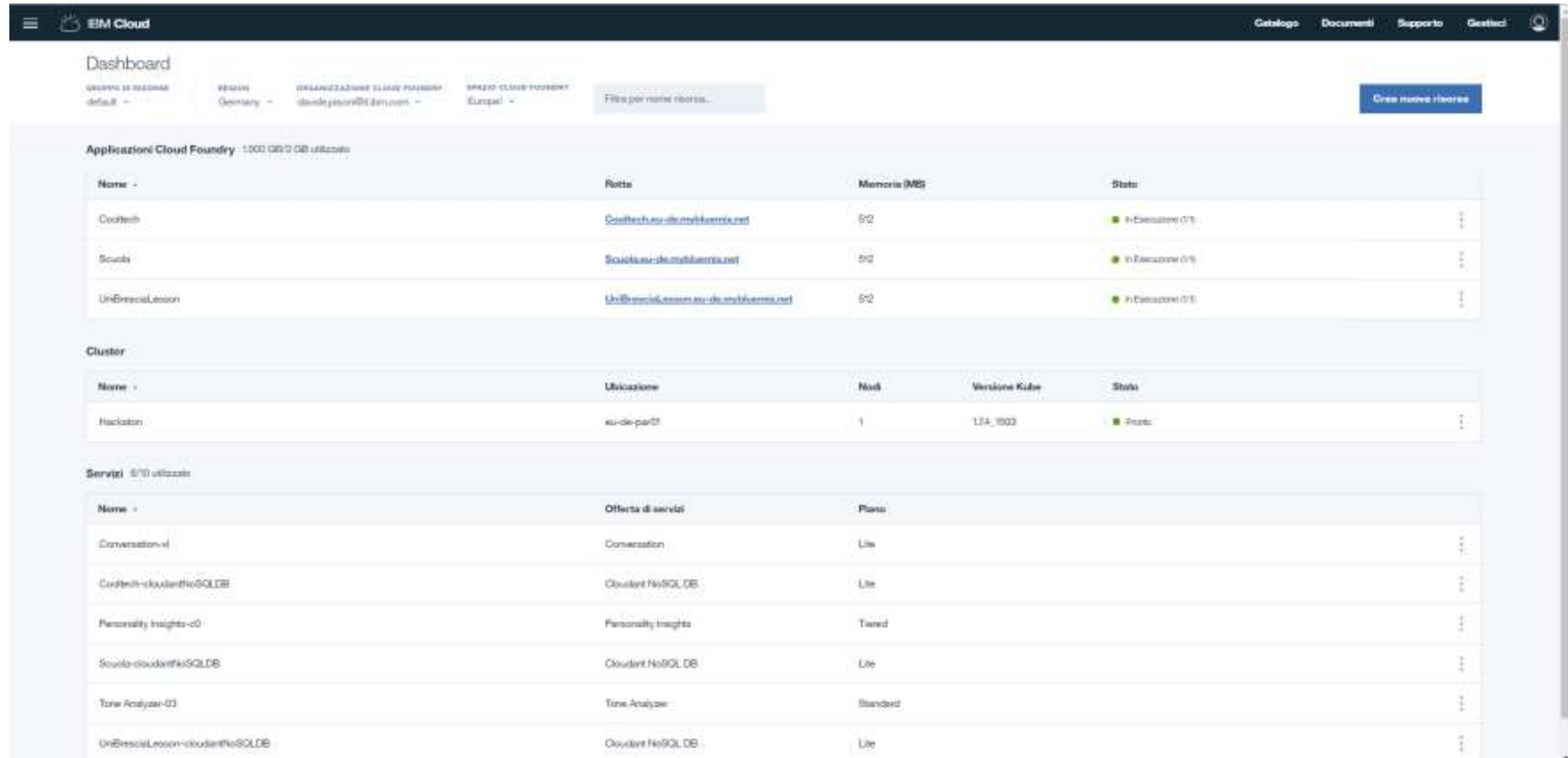
2. Create a local Flask App

3. Integrates Visual recognition in the app trough the python sdk

4. Deploy the app on IBM Cloud

 © 2017 IBM Corporation

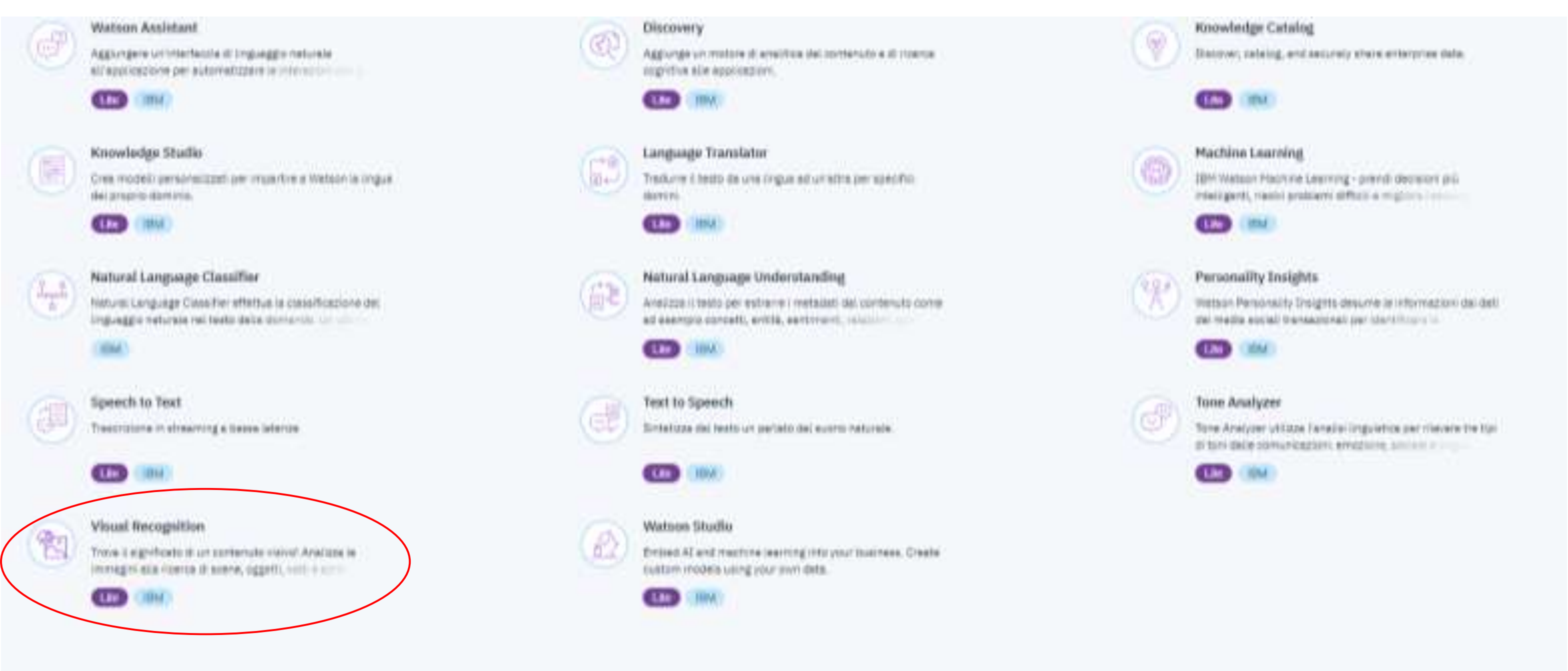
IBM Cloud Dashboard



 © 2017 IBM Corporation

Create Visual Recogition Service(1/2)

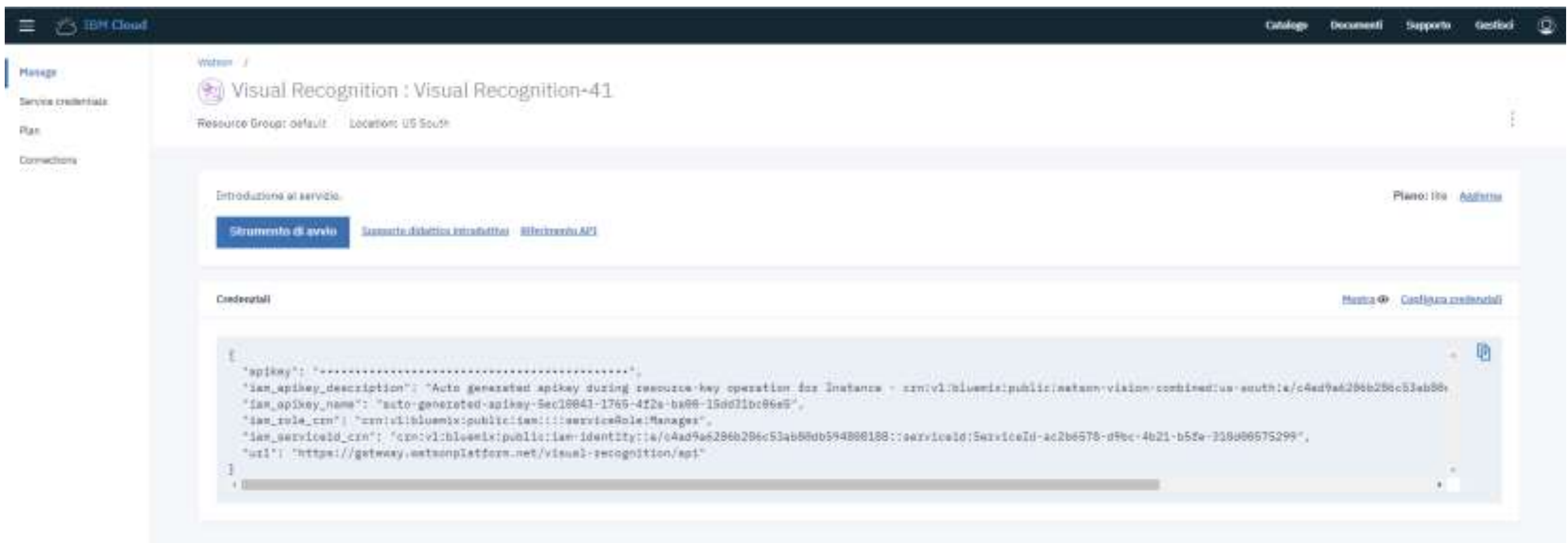
Login in the IBM Cloud, go to **catalog** and select in the Watson API the **Visual Recognition Service**



 © 2017 IBM Corporation

Create Visual Recognition Service(2/2)

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

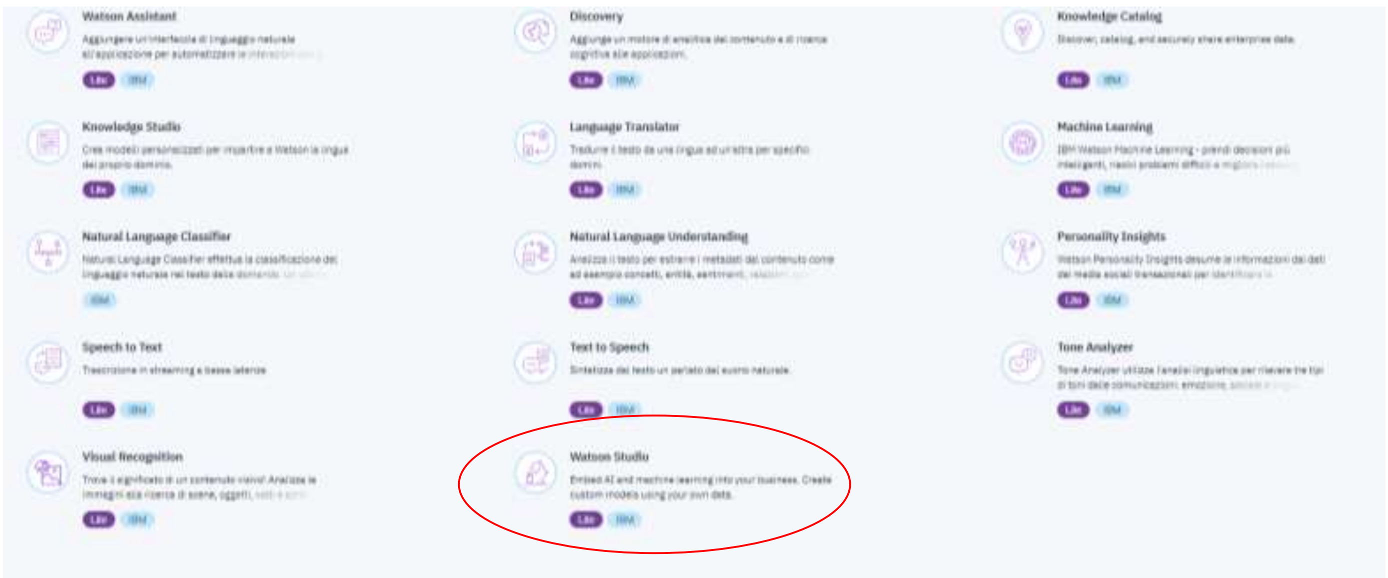


Here you can see the service credentials (in particular the API_KEY), used to call the service from the python SDK

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.



04/12/2017 © 2017 IBM Corporation

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 dependecies 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.

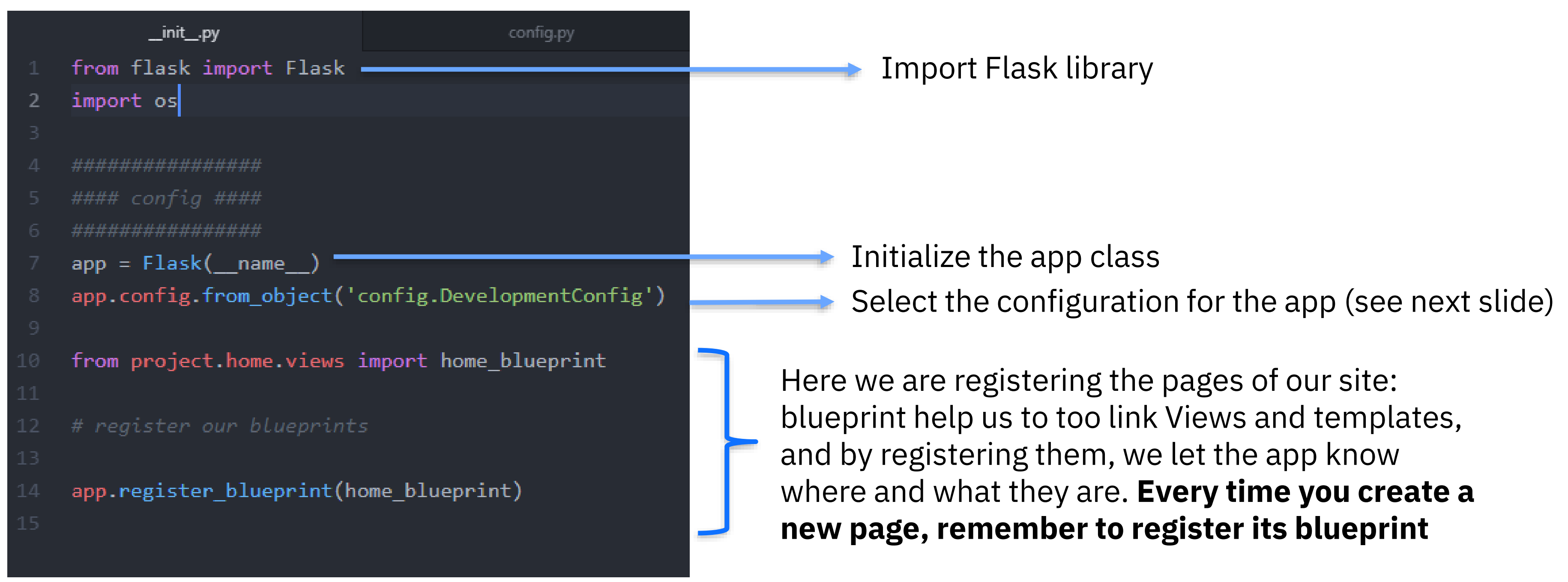
2. Inside the home folder, create a *templates* folder and two files, one called *views.py* and one __init__.py (this file remains empty and just state that python can go inside this folder to get function or variables). Here is the tree:

```
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:



Conifg.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 variabile, static folder, etc....

```
import os


#default config


class BaseConfig(object):
    DEBUG = False
    SECRET_KEY='\x8dBI\x9b\xbf\r\xd0t\xb5I\x86\x89]\xe4\x03~\x88\x8e\xac\xcb\xa5\xc7\xbfz'
    UPLOADED_PHOTOS_DEST='./project/static/images'

class DevelopmentConfig(BaseConfig):
    DEBUG = True

class ProductionConfig(BaseConfig):
    DEBUG = False
```

Here we create a BeseConfig object class, with some global varables 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

 © 2017 IBM Corporation

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

```
<!DOCTYPE html>
<html>
  <head>
    <title>Image Recognition App</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link href="static/bootstrap.min.css" rel="stylesheet" media="screen">
    <link href="static/mycss.css" rel="stylesheet" media="screen">
  </head>
  <body>

    <div class="container">

      <!-- child template -->
      {% block content %}{% endblock %}

      <!-- errors -->
      {% if error %}
        <p class="error"><strong>Error:</strong> {{ error }}</p>
      {% endif %}

      <!-- messages -->
      {% for message in get_flashed_messages() %}
        {{ message }}
      {% endfor %}
    </div>

  </body>
</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

```python
##################
#### imports ####
##################

from flask import render_template, Blueprint

################
#### config ####
################

home_blueprint = Blueprint(
    'home', __name__,
    template_folder='templates'
) #pragma: no cover

################
#### routes ####
################

# use decorators to link the function to a url
@home_blueprint.route('/', methods=['GET', 'POST'])
def home():
    error = None
    return render_template(
            'index.html', error=error)
```

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

```
{% extends "base.html" %}
{% block content %}

<h1>Image Recognition App</h1>


{% endblock %}

```

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

```python
from project import app
import os

port = os.getenv('PORT', '5000')
if __name__ == "__main__":
    app.run(host='0.0.0.0', port=int(port))
```

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

```
from watson_developer_cloud import VisualRecognitionV3
import json

visual_recognition = VisualRecognitionV3(
    version='2018-03-19',
    iam_api_key='apy_key'
)

def classify_image(path,classifier):
    with open(path, 'rb') as images_file:
        classes = visual_recognition.classify(
            images_file,
            parameters = json.dumps({
                'classifier_ids': [classifier]
            }))
    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 thate 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

```
from flask_wtf import FlaskForm
from wtforms import SelectField
from wtforms.validators import DataRequired,ValidationError
from flask_wtf.file import  FileField,FileAllowed, FileRequired


class PhotoForm(FlaskForm):
    classifier = SelectField('classifier', validators=[
    DataRequired(message="Select a classifier please")],
    choices=[('default','default'),('explicit','explicit'),('food','food')])

    photo = FileField('photo', validators=
    [FileRequired(message='Please load a photo'),
    FileAllowed(['jpg','png'], message='Select a jpg or a png photo')])

```

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 possibile 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

```
{% extends "base.html" %}
{% block content %}

<h1>Image Recognition App</h1>
<br>
<form class="photo-form" role="form" method="post" action="" enctype=multipart/form-data>

  {{ form.csrf_token }}
  <p>
    Image : {{ form.photo(class_='form-control',placeholder='Select an image') }}
    <span class="error">
      {% if form.photo.errors %}
        {% for error in form.photo.errors %}
          {{ error }}
        {% endfor %}
      {% endif %}
    </span>
  </p>
  <p>
    Classifier : {{ form.classifier(class_='form-control') }}
    <span class="error">
      {% if form.classifier.errors %}
        {% for error in form.classifier.errors %}
          {{ error }}
        {% endfor %}
      {% endif %}
    </span>
  </p>
  <button class="btn btn-lg btn-success" type="submit" name="submit">Classify</button>
</form>
<br>

{% 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 modifiy 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 src="static/images/{{filename}}"></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)

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 yuor dashboard: you can click on it and click on the link *Visit_URL*

Your app is online!

 © 2017 IBM Corporation

Thank You

For any questions, email me: davide.pisoni@it.ibm.com

 © 2017 IBM Corporation