TagMe

Rest API for Object Recgonition

Shawaiz Shah (s76940)

Contents

[Introduction 2](#_Toc713901)

[UML 3](#_Toc713902)

[1. Activity Diagram 3](#_Toc713903)

[2. Use Case Diagram 3](#_Toc713904)

[3. Deployment Diagram 4](#_Toc713905)

[Metrics 5](#_Toc713906)

[1. Complexity Metrics 5](#_Toc713907)

[2. Architectural Metrics 5](#_Toc713908)

[Clean Code Development 6](#_Toc713909)

[1. Consistent naming convention 6](#_Toc713910)

[2. Minimizing of Side-Effect using Context Management 6](#_Toc713911)

[3. Modularity 6](#_Toc713912)

[4. Exception Handling 6](#_Toc713913)

[5. Configurable Data at Higher Level 7](#_Toc713914)

[Continuous Delivery 8](#_Toc713915)

[Build Management (+ DSL) 9](#_Toc713916)

[Functional Programming 11](#_Toc713917)

[1. Final Data Structures Usage 11](#_Toc713918)

[2. Side Effect Free Functions 11](#_Toc713919)

[3. Use of higher order functions 11](#_Toc713920)

[4. Clojures/Anonymous Functions 11](#_Toc713921)

# Introduction

TagMe is an end-to-end pipeline for object recognition requests with a Rest API interface. It can be thought as a basic version of Google Vision Cloud API.

Following project was completed in two milestones;

* Milestone 1:

Development of an object labelling routine. **Tensor flow's object detection API** was used for this purpose

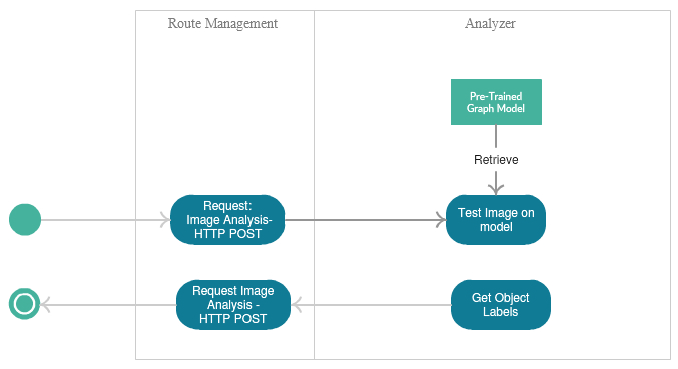
* Milestone 2:

A Restful API interface for handling HTTP requests. API endpoints were developed using. Flask Restful (an extension of project Flask)

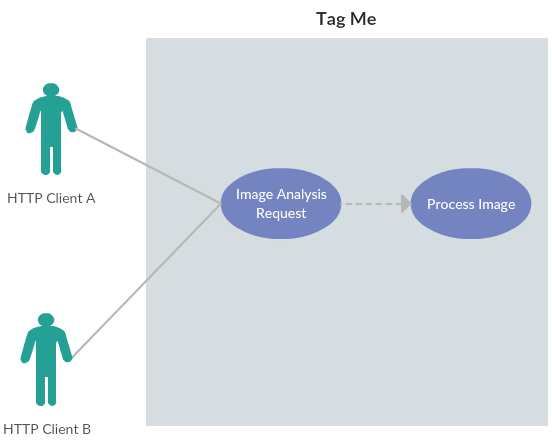
# UML

Following are the UML diagrams of project TagMe.

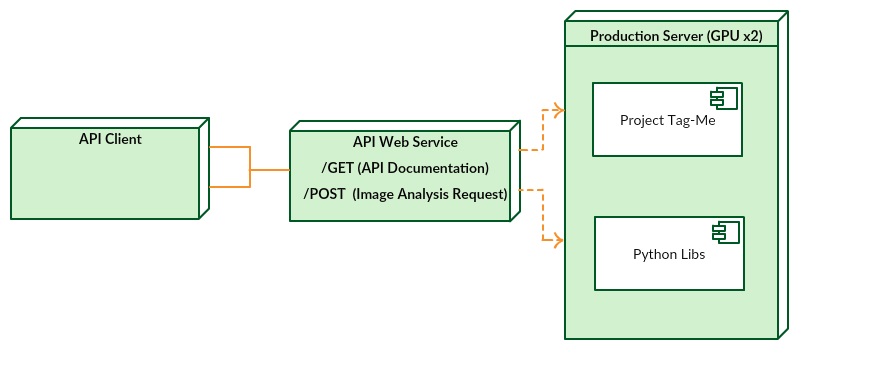
## Activity Diagram



## Use Case Diagram



## Deployment Diagram



# Metrics

For the code analysis and review, the industry standard tool, “SonarQube” was used. Initially code smell was significantly higher, which was gradually mitigated by applying refactoring techniques.

Please note that during code analysis, source codes from 3rd party libraries (e.g. object detection module from TensorFlow) were excluded.

For Example;

* In one case, a large code block was broken down into smaller code blocks, hence reducing code complexity.
* Function parameters were increased/decreased for making method calls more simpler and easier.
* Logically related pieces of code were merged together under same functions.

Software metrics that were monitor during course of development were;

## Complexity Metrics

In post-development code analysis, SonarQube indicated presence of Cyclomatic Complexity with a score of 17.

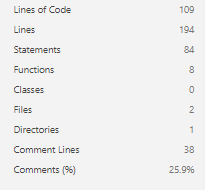


Complexity Score

Cyclomatic Complexity measures the minimum number of test cases required for full test coverage. Following measure indicates that my project requires 17 additional test cases for getting full test coverage.

## Architectural Metrics

Following are screenshots from metric measures.



Architectural Metric Measures

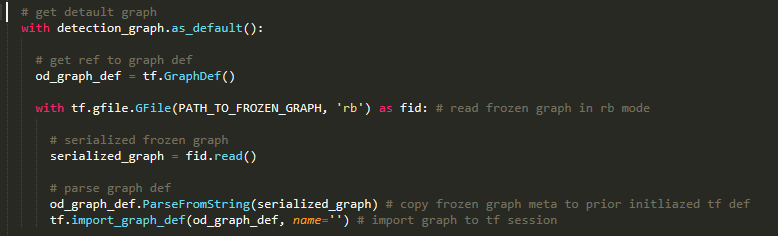
# Clean Code Development

## Consistent naming convention

Throughout coding, naming scheme of camel case is used for both variables and functions. Moreover, for immutable/final variables all-upper-casing scheme is used.

## Minimizing of Side-Effect using Context Management

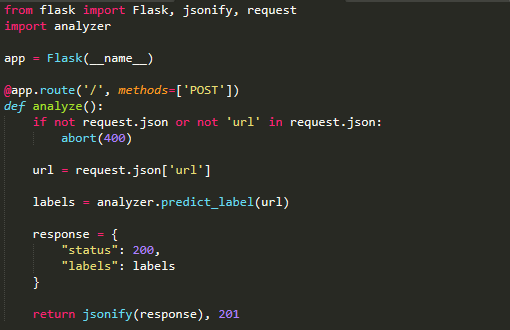
Wherever necessary, side-effects have been tried to minimized by isolating its effect at local scope by making use of python’s context management.



Context Management Usage

## Modularity

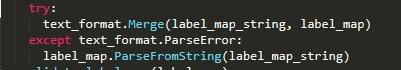
Source code has been divided into two modules. Module “Analyzer” is responsible for object recognition tasks, while module “Router” is responsible for routing of incoming and outgoing rest API requests.



Module Analyzer(analyzer.py) being imported in Module Router(app.py)

## Exception Handling

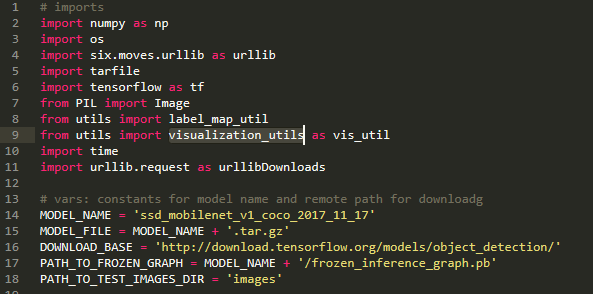
Wherever necessary exception handling blocks have been added to ensure that no runtime errors are thrown.



Exception Handling Usage

## Configurable Data at Higher Level

All configurable data (for example: Directory names, paths), have been placed at higher level, making tweaking of values easier for the purpose of debugging.



Configurable Data and Constants at higher level

# Continuous Delivery

Continuous Delivery and integration pipeline of this project is based on Git, Travis CI and Heroku. Travis CI is listening for modifications on Git repository. Any modifications in code base will trigger Travis CI, which will initiate a set of tests for validating successful integration of newly added code with already tested production-ready code in Git repository. Upon successful integration, Travis CI will deploy source code on our production server at Heroku.

Following diagram depicts code delivery flow;

Local Source Code Modifications

Git Repository

Integration Test Execution

Local Workstation

GIT

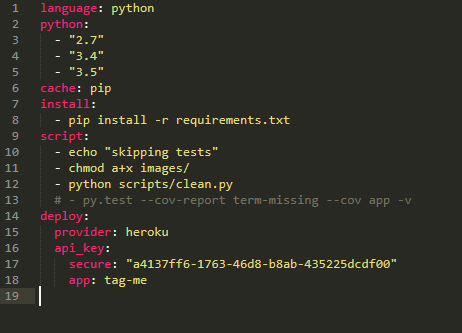
Travis-CI

Production Server

Heroku

# Build Management (+ DSL)

For the purposes of build management this project is dependent on Travis CI since builds are being managed at the time of project deployment. Consider following example;

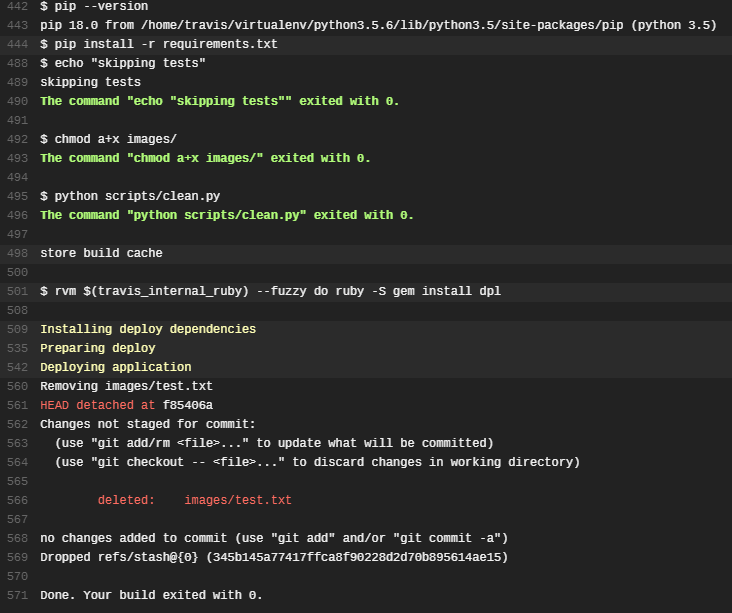


Build Script for Travis CI

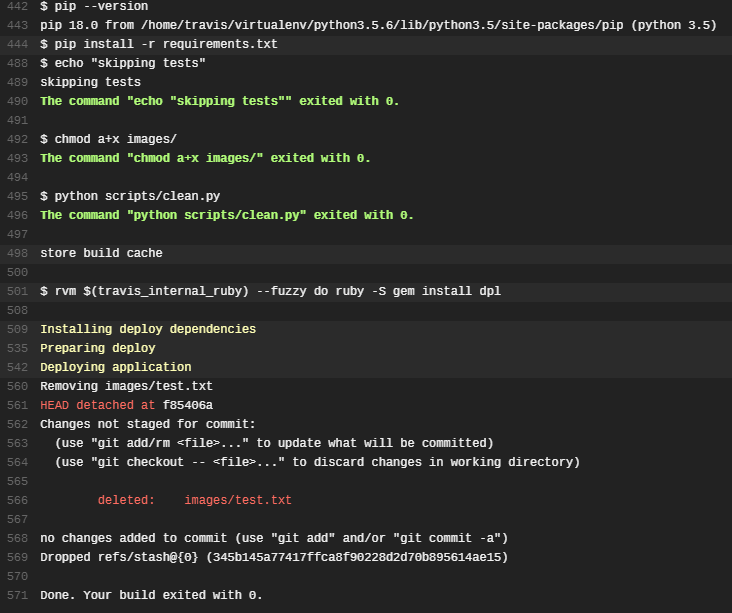
During build deployment, Travis has been instructed to;

1. Check for compatibility of source code with python version 2.7, 3.4 and 3.5
2. Install pre-requisite packages of project found in file requirments.txt
3. Delete contents of directory “images” where images that are to be processed will be stored
4. Deploy fresh build on Heroku instance for application “tag-me” (unique identifier on Heroku.com)

Following is the output of following script;



Build Script Output Log Part 1

 Build Script Output Log Part 1

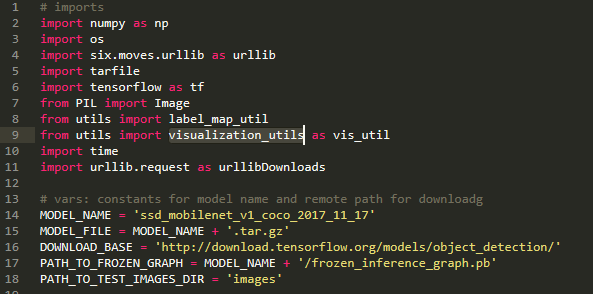
**P.S: Following script is also an example of Domain Specific Language. This script has been written in YAML (*.yml).***

# Functional Programming

Throughout this project, good practices for functional programming have been adopted. Following are few examples of such practices;

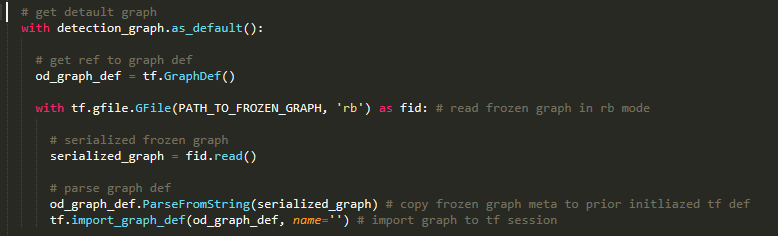
## Final Data Structures Usage

Few variables have been made made immutable in code. Hence using as constant.



## Side Effect Free Functions

As discussed earlier, wherever necessary context management has been used. So that its effect could remain locally. Hence using them would not result any side effects.



## Use of higher order functions

Higher order functions like map and filter have also been used.



## Clojures/Anonymous Functions

Usage of anonymous functions like “Lamda” have also been made, for getting rid from unnecessary function signature bodies where required.

