

# Technologies utilisées:

---

## Sommaire

---

- Introduction
- Choix du langage
- Choix des modules Python
- Choix des API
- Bade de données
- Unit test
- Deploiement
- Agile Dev
- Documentation
- Autres

## Introduction

---

Notre projet consiste en un BOT Messenger. Les raisons de ce choix sont expliquées dans le document 'Détail des différentes étapes'

Il nous fallait pouvoir:

- Récupérer les messages envoyé par l'utilisateur a notre page
- Analyser le message
- Chercher les évènements correspondant a sa recherche
- Leur trouver une image
- Les rendre a l'utilisateur en utilisant les fonctionnalités de Facebook
- Sauvegarder ses choix
- Analyser des tweets, extraire les informations pertinentes et les envoyer
- Executer des Unit Test
- Deployer
- S'organiser
- Utiliser du Code Review

Pour réaliser ce BOT Messenger, plusieurs options s'offraient a nous:

- Utiliser une solution web déjà faite consistant majoritairement a du glissé-déposé
- Le faire entièrement a la main

Afin d'avoir le plus de possibilités ouvertes et ne pas être limités, nous avons décidé de le faire entièrement à la main.

## Choix du langage

---

Après nous être documentés sur internet, nous avons décidé d'utiliser Python. En effet, la majorité de l'équipe connaissait déjà ce langage.

&

Les performances de ce langage convenaient parfaitement pour cette utilisation, nous ne cherchons pas à faire du code critique qui s'exécute en quelques millièmes de secondes.

Nous voulions de plus intégrer de nombreuses fonctionnalités rapidement. Ce langage haut niveau était donc parfait, nous aurions mis plusieurs mois avec un langage bas niveau comme le CPP

## Choix des modules Python

---

Python possède une communauté très active qui a ainsi développée de nombreux modules.

Nous avons décidé d'en utiliser certains:

- [Arrow](#) | Permet de mieux parser les dates

```
t = arrow.get('23-12-2017')
t.humanize()

[2 days before christmas]
```

Nous nous en servons ainsi pour parser les dates des événements et les rendre sous un format plus facilement lisible pour un humain / notre utilisateur

- [Flask](#) | Un framework Web

Flask nous permet de facilement récupérer les requêtes de facebook (reception d'un message).

C'est un Framework petit, léger et robuste, nous l'avons donc choisi car nous n'en demandions pas plus

- [FBMQ](#) | FBMQ Nous permet de formater les messages envoyés à Facebook

Messenger offre de grandes possibilités de customisation des messages (par exemple quick replies ou boutons).

Ce module nous permet de facilement demander cela a Messenger et nous évite d'écrire et formater de longues requêtes HTTP.

```
from fbmq import Page
page = Page('SECRET_KEY')

Exemple ici nous allons afficher du texte:

page.send(recipient_id, "hello world!")

et ici une image:

page.send(recipient_id, Attachment.Image(image_url))
```

## Choix des API

---

Il nous faut maintenant choisir des API pour remplir deux taches:

- Récupérer une liste d'événements
- Récupérer des images correspondants a ces événements

### Evenements

De nombreuses API étaient disponibles, Bandsintown, OpenAgenda, Facebook.

Malheureusement, la plupart d'entre elles ont leurs désavantages.

Trop cher pour certaines jusqu'à très limitées pour d'autres (Facebook)

Nous avons donc décidé d'utiliser EventFull. Leur api était assez complète et simple d'utilisation.

Gratuite, illimitée, et disposant de la plupart des événements actuels.

### Images

Une fois nos événements récupérés il nous fallait maintenant en récupérer des images pour illustrer tout cela.

Nous avons commencé par utiliser l'API de Google Images.

Malheureusement, nous nous sommes vite rendu compte que leur api était très lente. (de

l'ordre de 10-20 secondes pour une image). Ce qui était bien sur innacceptable.

Nous avons donc changé pour Qwant ! Le navigateur francais qui propose une api de recherche d'image extrêmement rapide.

## Base de données

---

Nous avons décidé d'utiliser MongoDB en tant que base de données. C'est la plus simple a mettre en place, et la plus adaptée en cas de montée en charge rapide.

Le format NoSQL permet aussi de restructurer le schéma a la volée ce qui était important car nous voulions pouvoir évoluer vite et changer ce dernier sans avoir a tout changer dans le code

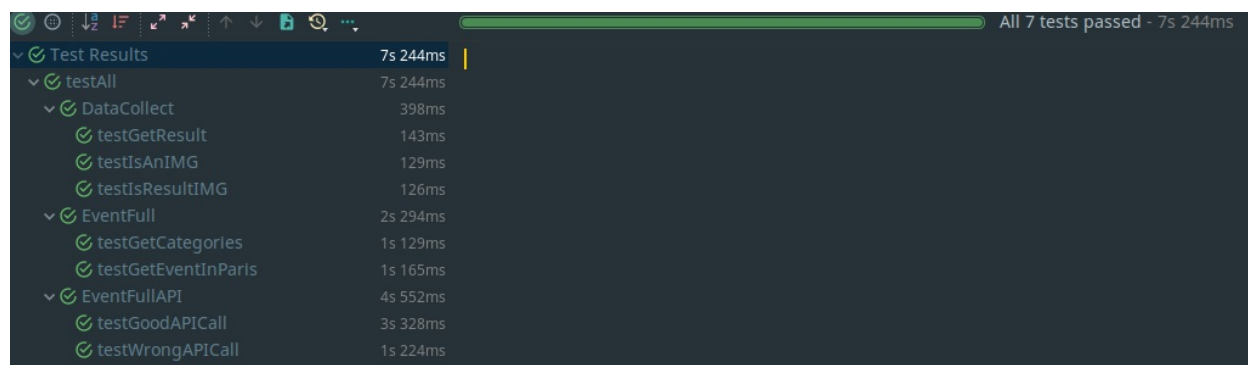
```
✓ [0]
  "_id" 5a23d622dee11389a3800dda
  "userId" TEST_USER
  > "taste" ["music"]
✓ [1]
  "_id" 5a23ec3ddee11389a3800ddb
  "userId" 909515002507104
  > "taste" ["business"]
  > "previsionTheme" ["music"]
  > "previsionCity" ["Nice"]
```

## Unit test

---

Les unit test sont l'unes des meilleures manières de s'assurer que notre code marche toujours après en avoir modifié une partie qui pourrait déteindre sur une autre.

Pour cela, nous les avons pris très au sérieux. Nous avons essayé d'avoir 100% de coverage et de tester absolument tout ce qui était possible.



```
✓ All 7 tests passed - 7s 244ms
✓ Test Results 7s 244ms
  ✓ testAll 7s 244ms
    ✓ DataCollect 398ms
      ✓ testGetResult 143ms
      ✓ testIsAnIMG 129ms
      ✓ testIsResultIMG 126ms
    ✓ EventFull 2s 294ms
      ✓ testGetCategories 1s 129ms
      ✓ testGetEventInParis 1s 165ms
    ✓ EventFullAPI 4s 552ms
      ✓ testGoodAPICall 3s 328ms
      ✓ testWrongAPICall 1s 224ms
```

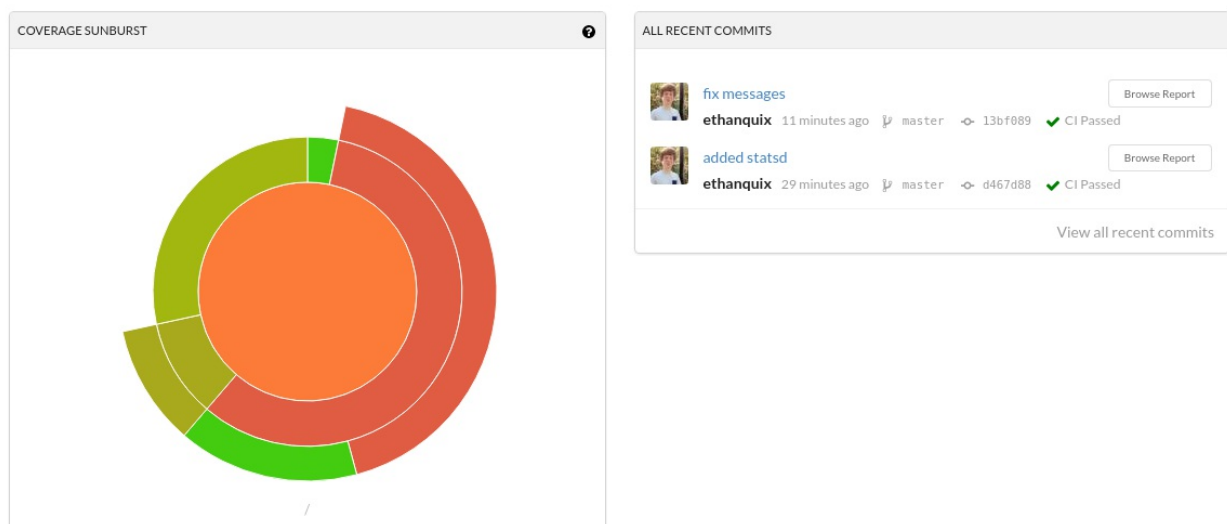
Mais nous avons aussi voulu aller plus loin.

Nous utilisons Travis afin de tester notre code.

Nous l'avons lié a notre compte github ainsi a chaque push, Travis exécute tous les tests unitaires présents.



Mais ce n'est pas tout. Nous avons aussi relié Travis a Codecov qui va ainsi nous générer un rendu du code coverage de notre code.



Cela nous a permis de nous rendre compte en temps réel des problèmes possibles et du code que l'on aurait pu avoir cassé

## Deploiement

### Les obligations

Pour déployer notre bot, Facebook demandait a ce qu'on l'héberge sur un serveur distant (non localhost), possédant un nom de domaine, et surtout sécurisé par HTTPS.

Nous avons donc acheté un nom de domaine (wispi.tk) et avons ensuite utilisé [Digital Ocean](#) pour héberger notre bot.

Nous avons choisi un "droplet" a 512mb de Ram et 1VCPU pour 5€/mois qui est amplement suffisant pour faire tourner notre bot

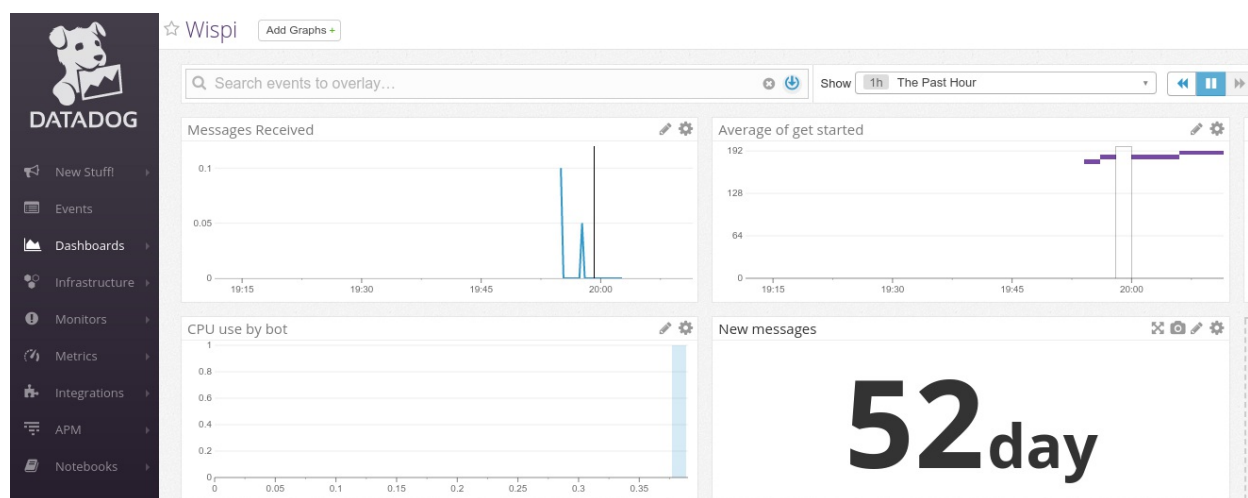
Pour générer le certificat https nous avons utilisé [Let's Encrypt](#) qui nous a permis de générer un certificat SSL gratuitement et facilement

```
$ letsencrypt nginx
```

Comme proxy inverse, nous utilisé [Nginx](#) pour sa facilité d'utilisation

Pour monitorer notre application ainsi que notre serveur, nous avons utilisé [Datadog](#) qui nous permet de voir en temps réel ce qui se passe mais aussi de définir des trigger dans le code afin d'obtenir différentes metrics

Il permet aussi de monitorer le serveur pour nous prévenir si ce dernier crash. Cela permet aussi de contrôler la consommation en ressources de notre programme



## Agile Dev

Entraînés par nos stage, et motivés par le gain de productivité que cela apportait, nous avons respecté les principes du développement Agile.

Ainsi, nous avons utilisé Youtrack afin de gérer les issues et le temps nécessaire, et upsource en tant qu'outil de code review.

A chaque push, il fallait 2 personnes qui approuvaient le code afin que ce dernier puisse être merge sur la branche Master.

## Documentation

La documentation a été faite en grande partie en Markdown.

Le markdown est un format simple, ayant un joli rendu. Surtout, contrairement a un document

Word, il n'y a aucun problème de compatibilité lors de l'édition de la documentation a plusieurs.

[Couscous.io](#) nous permet de générer cette documentation facilement et de la rendre sur github.

## Autre

Nous avons aussi utilisé heroku afin de facilement pouvoir déployer notre bot sans avoir besoin d'avoir son propre serveur.

Un Procfile est disponible a la racine, il suffit donc de déployer cette application avec heroku et elle sera instantanément prête a l'emploi

The screenshot shows the Heroku dashboard for an application named 'wispi'. The top navigation bar includes 'Personal', '>', 'wispi', a star icon, and buttons for 'Open app' and 'More'. Below this is a tabbed interface with 'Overview' selected, followed by 'Resources', 'Deploy', 'Metrics', 'Activity', 'Access', and 'Settings'.

The main content area is divided into three sections:

- Installed add-ons:** Shows a message: "There are no add-ons for this app. You can add add-ons to this app and they will show here. [Learn more](#)". The cost is listed as "\$0.00/month".
- Dyno formation:** Shows the app is using "free dynos". A table lists the formation: 

Type	Command	Status
web	gunicorn app:app --log-file=-	ON
- Collaborator activity:** Shows a list of collaborators. One collaborator, 'dimitriwyzlic@gmail.com', is listed with '9 deploys'.

On the right side, the 'Latest activity' section shows a list of recent events:

- Deployment by dimitriwyzlic@gmail.com (v13) with commit 3ea17190.
- Build succeeded by dimitriwyzlic@gmail.com (v13) with build log link.
- Deployment by dimitriwyzlic@gmail.com (v12) with commit 7637b603.
- Build succeeded by dimitriwyzlic@gmail.com (v12) with build log link.
- Deployment by dimitriwyzlic@gmail.com (v11) with commit bc93bb4b.
- Build succeeded by dimitriwyzlic@gmail.com (v11).