

CENTRE DE FORMATION PROFESSIONNELLE TECHNIQUE

**13 avril 2021**

**Travail de diplôme**

**Soccer Pronostic**

Rapport de stage

David Paulino

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Stage . . . . .	2
1.2	Proof Of Concept . . . . .	2
1.3	Environnement . . . . .	2
1.4	Organisation . . . . .	3
1.5	Livrable . . . . .	3
<b>2</b>	<b>Développement</b>	<b>3</b>
2.1	Description de l'architecture . . . . .	3
2.1.1	Architecture du projet . . . . .	3
2.1.2	Base de données . . . . .	3
2.2	Description des activités . . . . .	5
2.2.1	Façade pour la communication avec l'API . . . . .	5
2.2.2	Dotenv . . . . .	6
2.2.3	Logging . . . . .	6
2.2.4	Base de données . . . . .	7
<b>3</b>	<b>Bibliographie</b>	<b>7</b>

# 1 Introduction

Ce rapport a comme objectif de détailler toutes les étapes lors de mon stage effectué au CFPT. Comme les contrats de stages de techniciens ont été annulés par l'école suite à la pandémie du COVID-19, il nous a été demandé de faire un équivalent de stage pour pouvoir obtenir notre diplôme de technicien. Le travail du stage a pour but de produire un POC ("proof of concept") du travail de diplôme pour démontrer la faisabilité du projet et pour nous permettre d'avancer sur le projet qui nous permettra d'obtenir le titre de technicien ES.

## 1.1 Stage

Durant 6 jour de travail, soit 48 heures, l'objectif de mon stage est de réaliser un POC pour mon travail de diplôme. L'objectif de ce dernier est de réaliser des prédictions sur un match entre deux équipes de football sur une base de statistiques des matchs récents de ces deux équipes. Cela permet ensuite à des pronostiqueurs de se baser sur ce travail pour avoir une source supplémentaire pour choisir sur quelle équipe miser lors d'un paris sportif.

## 1.2 Proof Of Concept

Le POC a pour but de concevoir la classe qui me permet de récupérer les données de l'API que j'utilise pour mon travail de diplôme. J'y ajouterai évidemment de nouvelle chose à faire au fil des jours de stage, comme l'élaboration de l'architecture de l'application ou encore le logging des appels à l'API.

## 1.3 Environnement

- Un PC standard école avec Windows 10, 2 écrans
- Visual Studio Code
- Outil de versionnage de code (Git, avec dépôt distant sur Github / Bitbucket / GitLab)
- Navigateur web (Mozilla Firefox / Google Chrome)
- Outil bureautique à choix pour les documents
- Accès à une source de données pour avoir des statistiques sur des matchs de football

## 1.4 Organisation

Étudiant :

— David Paulino, **david.plnmr@eduge.ch**

Tuteur de stage :

— Antoine Schmid, **antoine.schmid@edu.ge.ch**

## 1.5 Livrable

Pour la fin du stage, le 15 avril 2021 :

Pour le tuteur de stage :

- Rapport de stage au format PDF
- Journal de bord au format PDF
- L'accès au répertoire distant pour pouvoir cloner le projet
- Une explication des pré-requis, pour pouvoir potentiellement exécuter le travail effectué par l'étudiant, dans un README à la racine du répertoire distant

# 2 Développement

## 2.1 Description de l'architecture

### 2.1.1 Architecture du projet

Après discussion avec mon tuteur de stage au sujet de l'architecture du projet, j'ai décidé de faire un répertoire pour chaque fonctionnalité. (voir fig. 1) J'avais envisagé de séparer les classes et les fichiers de tests mais on se retrouve mieux dans une arborescence comme celle là.

### 2.1.2 Base de données

Au niveau du schéma de la base de données, j'avais initialement prévu de faire deux tables, une table qui contiendrait les matchs et une table qui contiendrait les prédictions en lien avec ces matchs. Cependant, comme il faut uniquement stocker les prédictions et pas forcément les matchs, après discussion avec M. Schmid, on en a conclu qu'une seule table suffisait. (voir fig. 2)<sup>1</sup>

---

1. Le diagramme a été fait sur <https://dbdiagram.io/>

soccer\_pronostic

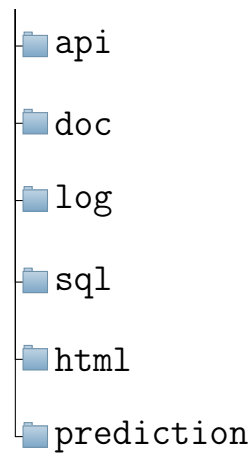


FIGURE 1 – Arborescence prévue pour le projet

prediction	
id	int
created_at	timestamp
prediction	varchar
api_match_id	int
hometeam_name	varchar
awayteam_name	varchar
off_score_hometeam	int
def_score_hometeam	int
off_score_awayteam	int
def_score_awayteam	int


 dbdiagram.io

FIGURE 2 – MLD de la BDD

## 2.2 Description des activités

### 2.2.1 Façade pour la communication avec l'API

Pour la communication avec l'API, il m'a été recommandé par mon tuteur de stage d'utiliser le design pattern Façade<sup>2</sup>.

Ce design pattern permet d'avoir une interface simple vers un système complexe. Par exemple, si nous utilisons une librairie pour faire du traitement d'image qui contient beaucoup de classes et que nous souhaitons convertir une image en SVG, nous allons créer une Façade qui va utiliser toutes les classes nécessaires pour faire cette conversion. (voir figure 3)

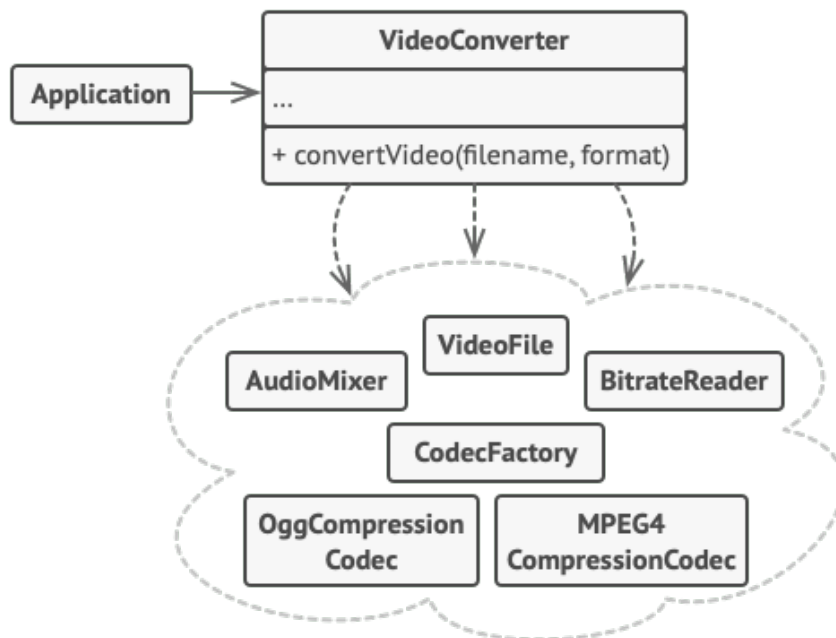


FIGURE 3 – Schéma pour montrer le fonctionnement de la façade

Dans notre cas, il permet de simplifier la communication entre l'API et l'application. Pour se faire, j'ai simplifier la manière de faire des requêtes à l'API en créant une méthode privée qui se nomme `getAction()`. Cette méthode est générale et elle prend en paramètres les paramètres nécessaire pour toutes les requêtes spécifiques faites à l'API.

```
1 def getAction(self, request_params):  
2     response = requests.get('https://apiv2.apifootball.com/?APIkey  
   ={{key}}&{{params}}'.format(key=self.api_key, params=request_params))  
3
```

2. <https://refactoring.guru/fr/design-patterns/facade>

```
4         if response.status_code == 200: # Code 200 = OK. Healthy
           connection
5         return response.content.decode('utf-8') # decode to get
           the content in string
6         # TODO : VERIFICATION IF THE CODE IS NOT 200
```

Listing 1 – `getAction.py`

Vous pouvez apercevoir sur le listing 1 que le code est très généraliste pour permettre d'être utilisé par n'importe quelle méthode. On peut aussi remarquer qu'à la ligne 5 du listing 1, on doit décoder le contenu récupéré de la requête car c'est un texte encodé en byte. (b').

Le listing 2 quant à lui utilise `getAction()` en passant en paramètre les données spécifiques à `getCountries()`.

```
1 def getCountries(self):
2     endpoint_action = "get_countries"
3     return self.getAction('action={action}'.format(action=
           endpoint_action))
```

Listing 2 – `getCountries.py`

### 2.2.2 Dotenv

La librairie Dotenv est très utile pour éviter de stocker des données sensibles, tels que des mots de passes de base de données ou des tokens d'API.

Je n'ai pas rencontré de problème dans l'ajout de la librairie au POC. J'ai donc créé un fichier nommé `.env` qui contient les variables d'environnement et lorsque j'ai besoin de ces variables-ci, je fais un `load_dotenv()` pour récupérer les variables stockés dans le fichier `.env`. Cela me permet de garder une sécurité sur ma clé d'API et sur les mots de passes utilisateurs de ma base de données.

### 2.2.3 Logging

Le logs sont une chose très important en programmation. Cela permet d'avoir une trace sur les actions faites et pour nous permettre de comprendre certaines erreurs qui peuvent apparaître, sans les afficher directement à l'utilisateur. Les logs sont stockés dans un fichier prévu à cet effet.

J'ai rencontré un soucis lors de l'implémentation des logs dans mon POC. En effet, les méthodes `info` et `debug` ne s'affichaient pas, mais les méthodes `warning` ou encore `error` s'affichaient. N'ayant pas trouvé de solution à ce soucis, j'en ai parlé brièvement à Monsieur Garcia qui m'a informé sur le fait que les librairies de logging utilise généralement un filtre sur l'affichage

des logs. Après cette information, nous avons vérifié ensemble sur la documentation de la librairie et on pouvait spécifier le filtre que l'on souhaite appliquer dans la configuration de base des loggings. Cela m'a permis de régler ce point bloquant.

### 2.2.4 Base de données

Pour l'application, une base de données va nous permettre de pouvoir stocker les prédictions faites de manière hypothétiques pour avoir un historique sur ces dernières. La classe `DbManager` permet de gérer la base de données (`SELECT`, `INSERT`, `DELETE`). J'y ai ajouté des `SELECT` statements pour récupérer une prédiction selon l'identifiant de l'API et selon les noms d'équipes.

Des logs sont évidemment écrit à chaque statement qui peut effectuer un changement sur la base de données.

```
1 def __query(self, your_query):
2     """
3     Make your own query in the specified table. Make sure to only
4     give a SELECT statement, otherwise your query won't work.
5     """
6     self.__cursor.execute(your_query)
7     return self.__cursor.fetchall()
```

Listing 3 – queryDb.py

Sur le listing 3, j'ai essayé de faire une méthode minimale qui sera appelé par tous les `SELECT` statements. Cette dernière est **private**.

```
1 def getAll(self):
2     """
3     Get all the rows in the table
4     """
5     return self.__query("SELECT * FROM prediction")
```

Listing 4 – getAll.py

Le code du listing 4 appelle uniquement le code du listing 3 avec la requête qu'il faut. Ensuite, il retourne le resultat de la requête sous forme de dictionnaire.

## 3 Bibliographie

- <https://refactoring.guru/fr/design-patterns/facade>
- <https://pypi.org/project/python-dotenv/>
- <https://docs.python.org/3/library/logging.html>



— <https://www.javatpoint.com/how-to-connect-database-in-python>