Travail de semestre

Thomas Fujise

Thomas Fujise

CFPT 2022

Table des matières

1. Cahier des charges

1.1 Sujet

Créer une application WEB regroupant des clients et des coachs de sport pouvant récupérer des données d'entrainements directement depuis un appareil connecté Polar

1.2 But du projet

Le but du projet est de créer une plateforme WEB regroupant pratiquants et coachs afin de gérer une salle de sport et le suivi des pratiquants. Les données d'entrainements suivantes sont récupérées depuis une smartwatch Polar:

- Pulsation cardiaque (Repos/Actif/Après effort)
- Le type d'exercice effectué
- Date et durée de l'entrainement
- Nombre de total de calories brulées
- Nombre de calories active (Optionnel)
- Les données relatives au sommeil
  - Pulsation cardiaque
  - Cycles de sommeil
  - Interruption
  - Hypnogramme

Il y a également un système de badging pour savoir quand le pratiquant commence et mets fin à son entrainement.

1.3 Utilisation

La plateforme Web permet de s'enregistrer soit en tant que pratiquant, soit en tant que coach.

En tant que clients : L'ID utilisateur Polar doit être renseigné lors de l'inscription. Une fois le compte créé, on dispose d'une page profil avec plusieurs données d'entrainement, quelques graphiques pour les illustrer permettant d'observer la progression au fur et à mesure du temps. Lorsqu'un compte client est créé, une carte RFID lui est attribuée afin de badger le début et la fin d'un entrainement. Une fois la fin de l'entrainement badgé, l'API Accesslink Polar permettra de récupérer les données de la séance achevée. Le profil du client sera mis à jour automatiquement. 2 onglets sont également disponibles "Entrainements" et "Diète" qui comporteront les programmes d'entrainements et de nutritions du client.

En tant que coach: Une fois connecté, on dispose d'une page avec la liste de tous les pratiquants dont on a la charge. On peut ajouter un pratiquant en le recherchant par son nom, en cliquant sur le bouton "ajouter" un mail de confirmation sera envoyé au client concerné pour valider la prise en charge. Le coach peut uploader un fichier (.csv/.xls) pour les programmes et la diète sur le profil de ses pratiquants pour leur transmettre directement depuis l'application, lors de la mise à jour du programme d'un pratiquant, celui-ci reçoit un mail pour le prévenir. Un fichier "Template" pour conserver un format commun entre coachs sera disponible sur la page d'accueil, si le format mit à disposition n'est pas respecté le fichier ne sera pas pris en compte.

1.4 Spécifications

- Les données d'entrainements sont récupérées avec l'API Accesslink de Polar en Python sous forme d'objet, elles sont vérifiées et stockées directement dans la base de données
- Le système de badge fonctionne à l'aide de cartes RFID et un lecteur NFC (ACS ACR 122u)
- Les graphiques liés aux stats sont effectués avec la librairie javascript Chart.js

- Les programmes d'entrainements et de nutritions doivent respecter le format proposé (Lors de l'upload une vérification est effectuée)
- Un système de notification est mis en place pour avertir les utilisateurs des différents événements

1.5 Restrictions

- Pour l'instant, il est uniquement possible d'utiliser des appareils Polar pour les données d'entrainement.
- Le système de badge est utilisé au début et à la fin et non pendant l'entrainement.
- L'API accepte au maximum 500 requête pour 20 utilisateurs différents en 15 min et 5000 pour 100 utilisateurs en 24h
- Les montres utilisées lors des entrainements sont celles du coach (montres déjà enregistrer sur le client qui accède à l'API)

1.6 Environnement

Technologies utilisées :

- HTML5
- CSS3
- Javascript
- SQL
- Python
- Flask (Micro Framework Python)
- Lecteur NFC (ACS ACR122U)
- API AccessLink Polar
- Chart.js

Système d'exploitation :

- Développement sur Windows

Versionning / Documentation :

- GitHub
- MarkDown (Mkdocs)

1.7 Livrable

- Fichier zip contenant le projet + bdd
- Documentation technique et journal de bord au format PDF

2. Documentation technique

2.1 Rappel du cahier des charges

2.1.1 Introduction

Dans le cadre du 2ème travail de semestre, j'ai décidé d'effectuer un projet qui me sera très utile pour le travail de diplôme. L'idée serait de faire une application en python qui récupère des données d'entrainement avec l'API Polar en fonction de la carte RFID qui est détecté.

But du projet

Le but de ce projet est de me préparer au maximum pour le travail de diplôme. C'est pour cela que j'ai décidé de mettre ensemble 2 technologies que je n'ai pas encore beaucoup utilisé et qui sont essentielles pour mon travail de diplôme.

Pour ce projet, uniquement les scripts python permettant de traiter les données voulues seront effectués. La partie interface de l'application finale est réalisée dans le cadre d'un autre cours.

Contraintes techniques

Je dois utiliser un lecteur de carte à puce et l'API Polar Accesslink pour récupérer les données d'entrainements.

2.1.2 Analyse

Technologies

- Python
- Markdown
- Github (versionning, task manager)
- Ordinateur de type PC, 2 écrans
- Visual Studio Code
- Mkdocs (Documentation)
- NFC Reader ACR122U
- Polar Accesslink API

NFC READER ACR122U

Le lecteur NFC ACR122U est un appareil permettant de lire et d'écrire sur des cartes sans contact. Il est basé sur la technologie Mifare 13,56 MHz (RFID) et suit les standards de la norme ISO 18092. L'ACR122U est développé et vendu par ACS ltd .

Pour pouvoir utiliser ce lecteur, j'ai utilisé la librairie pyscard qui me permet de récupérer les infos des cartes à puces que le lecteur lit

RFID

Qu'est-ce que la technologie RFID ??

RFID (Radio Frequency Identification), est une technologie qui permet d'enregistrer des données sur un support et de les récupérer à distance. Elle est apparue dans les années 1940 et était utilisée uniquement par l'armée pour l'identification des avions de guerre qui entrait dans l'espace aérien du Royaume-Uni. Elle s'est ensuite répandue dans différents secteurs industriels à partir des années 1980.

Comment fonctionne la RFID ?

Les étiquettes RFID, sont composées d'une puce RFID et d'une antenne et sont collées sur un produit. Elles enregistrent les données et avec un lecteur électromagnétique on peut ensuite lire les ondes radio présentes sur la puce RFID grâce à l'antenne.

Pourquoi utiliser RFID

RFID est un système de traçabilité. A l'aide d'une seule puce, il est possible de tracer les produits pendant tout le processus de production, de transport et de distribution, voire même jusqu'à leur fin de vie.

Dans le cadre de mon projet, RFID est une solution adéquate pour gérer les entrées/sorties des clients dans la salle d'entrainement.

Différences entre RFID et NFC

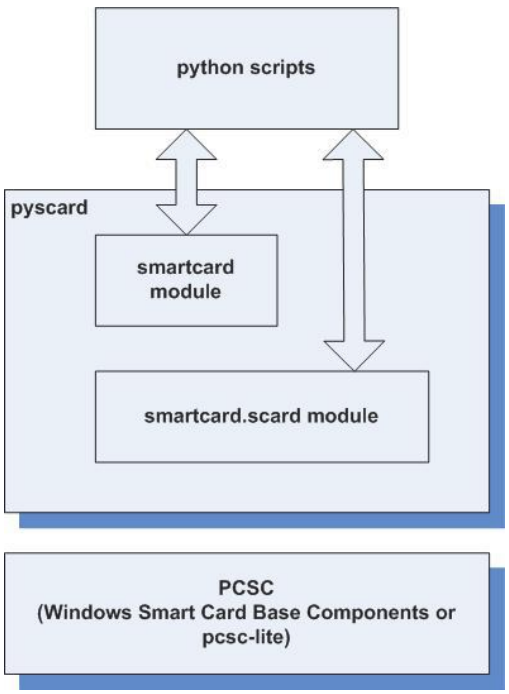
Dans le cadre de mon projet, j'utilise parfois le terme NFC. La technologie NFC (Near Field Communication) est un dérivé de la RFID qui a commencé à être utilisé dès 2011. Le NFC repose sur le même concept que la RFID. C'est une technologie qui fonctionne avec une puce permettant d'échanger des données entre un lecteur et n'importe quel terminal avec un simple rapprochement ou contact entre les deux objets.

La communication sans fil ne fonctionne qu'à courte portée et haute fréquence, une distance d'environ 10cm. La technologie NFC se retrouve dans la plupart des smartphones, consoles de jeux ou cartes bancaires. Le lecteur de carte que j'utilise fonctionne à l'aide de cette technologie également.

Les principales différences entre RFID et NFC résident dans la portée plus courte et sécurisée pour NFC (10cm) contre jusqu'à 10m pour RFID. La technologie NFC peut transmettre toute sorte de données contre RFID qui ne transmet que l'ID. La lecture fait, quant à elle, défaut à la technologie NFC qui ne peut lire qu'une puce à la fois ce qui peut limiter ses cas d'utilisation.

PYSCARD - LIBRAIRIE PYTHON SMART CARD

Pyscard est un module python qui permet d'utiliser les cartes à puce (PC/SC) avec python. Il donne accès à plusieurs classes et fonctions donnant accès aux cartes et aux lecteurs.



Architecture pyscard :

- smartcard.scard est un module d'extension enveloppant l'API WinSCard (Les composants de base smartcard) aussi connue sous le nom PC/SC (Personal computer / Smart Card)
- smartcard est un framework Python construit à partir de l'API PC/SC

Installation

Pour installer la librairie **pyscard** sur Windows 10, il faut au préalable installer SWIG et l'ajouter directement au PATH. Il faut ensuite installer Visual C++ version 14.0 ou plus récente (directement installable depuis le Visual Studio Installer).

SWIG

SWIG est un outil logiciel open source qui permet de connecter des logiciels ou librairies écrites en C/C++ avec des langages de scripts tels que : Perl, Python, Ruby, PHP ou d'autres langages de programmation comme Java ou C#.

MYSQL CONNECTOR/PYTHON

MySQL Connector permet aux programmes Python d'accéder aux bases de données MySQL. Je l'ai utilisé dans mon projet, pour la vérification des cartes de membres.

Installation

MySQL Connector vient directement avec l'installation de Python et peut être importé dans n'importe quel script avec :

```
import mysql.connector
```

OAUTH 2.0

OAuth est un protocole libre qui permet d'autoriser un site web, logiciel ou une application à utiliser l'API sécurisée d'un autre site web. L'API Polar utilise notamment ce protocole pour gérer les authentifications.

POLAR ACCESSLINK API

Accesslink est une API qui donne accès aux données d'entrainement et d'activitée journalière enregistrés par les appareils Polar. Pour pouvoir l'utiliser il est nécessaire de posséder un compte Polar Flow afin de créer un client sur admin.polaraccesslink.com qui nous donnera accès à l'API.

Accesslink utilise OAuth2 comme protocole d'authentification. Les utilisateurs enregistrés ont besoin de s'authentifier pour pouvoir avoir accès aux données.

Fonctionnalités de base d'Accesslink :

Fonctionnalité	Description
Utilisateurs	Permets d'enregistrer, supprimer et récupérer les informations de base de l'utilisateur
Pull Notification	Permets de vérifier si l'utilisateur à des données disponible à récupérer
Donnée d'entrainement	Permets d'accéder aux données d'entrainements de l'utilisateur
Activité journalière	Permets d'accéder aux données de l'activité journalière de l'utilisateur
Info physique	Permets d'accéder aux informations physique de l'utilisateur (Ex: Taille/Poids)
Modèle de données	Décrit tous les objets qui transportent les données entre le serveur et le client
Annexes	Contient des exemples et des détails sur l'interface de l'application

Environnement de développement

Pour l'environnement de développement, j'ai utilisé Visual Studio Code.

2.2 Analyse fonctionnelle

2.2.1 Interface

La partie réalisée lors du travail de semestre ne possède pas d'interface. Le programme récupère et sauve certaines données mais n'affiche rien. La partie "Affichage" de mon projet a été réalisé dans le projet du cours PDA.

2.2.2 Fonctionnalités

Inscription

Lors de l'inscription d'un nouveau client, il remplit un formulaire, puis une fois le formulaire remplit le coach doit scanner une carte RFID. Le lecteur va lire la carte et l'ID de la carte sera attribué au compte client du nouvel inscrit.

Authentification

L'application permet de s'authentifier à l'aide du lecteur et d'une carte RFID. L'ID de la carte est enregistré dans la base de données. Si l'authentification est réussite, l'utilisateur peut utiliser une montre Polar pour son entrainement. A la fin de l'entrainement, le client scanne sa carte RFID avant de partir. Si la carte est reconnue, l'application va chercher les dernières données d'entrainement disponible à l'aide de l'API Polar AccessLink.

Enregistrement

Lorsque des données d'entrainement sont récupérés suite à la séance d'un client, elles sont automatiquement enregistrées dans la base de données et relié directement au client concerné.

Cas d'utilisation

A l'arrivée d'un nouveau client, il doit s'inscrire via un formulaire. Une fois le formulaire renseigné, le coach doit scanner une nouvelle carte RFID à l'aide du lecteur. La carte sera enregistrée avec le compte du nouveau client en base de données.

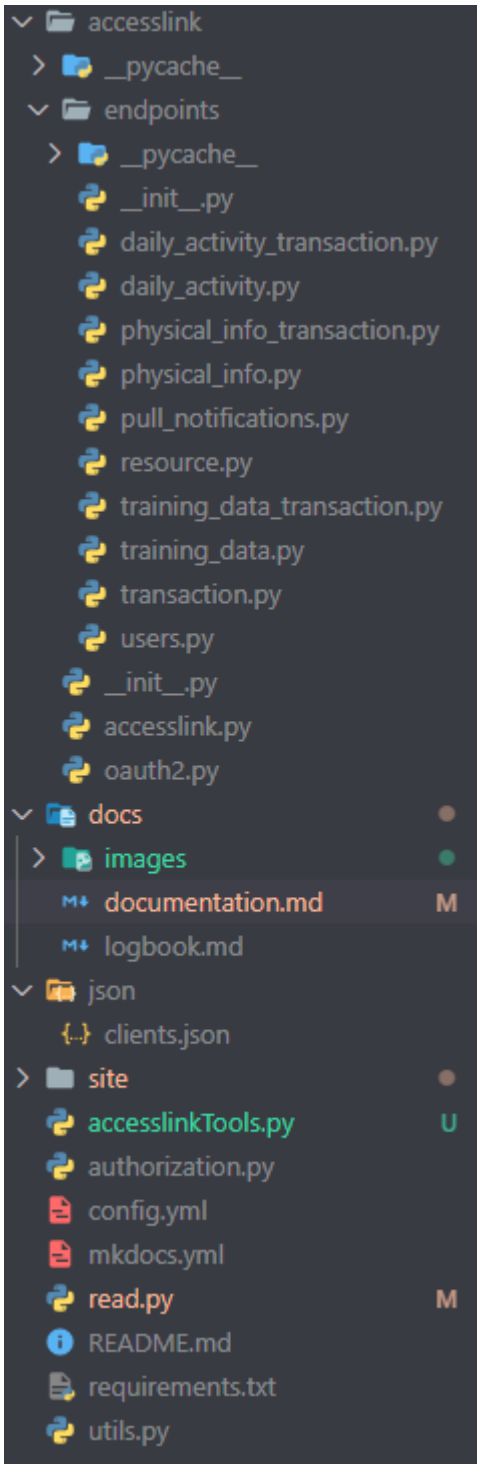
Lorsqu'un client vient s'entrainer il doit s'authentifier à l'entrée à l'aide de sa "carte de membre" (la carte RFID remise lors de l'inscription). Une fois entré, le client peut se munir d'une des montres Polar misent à disposition pour son entrainement.

A la fin de son entrainement, juste avant de partir, le client badge à la sortie avec sa carte. Si la carte est reconnue, les dernières données d'entrainements sont récupérées à l'aide de l'API Polar. Toutes les données voulues sont enregistrées dans la base de données et sont reliés directement au client concerné.

2.3 Analyse organique

2.3.1 Architecture

Arborescence de fichier



DOCUMENTATION.MD

Fichier Markdown contenant la documentation technique du projet.

LOGBOOK.MD

Fichier Markdown contenant le résumé journalier du travail effectué.

AUTHORIZATION.PY

Script python qui permet d'authentifier un compte Polar pour avoir accès à l'API

```
config["user_id"] = token_response["x_user_id"]
config["access_token"] = token_response["access_token"]
save_config(config, CONFIG_FILENAME)

# Register the user as a user of the application.
# This must be done before the user's data can be accessed through AccessLink.
try:
    accesslink.users.register(access_token=config["access_token"])
except requests.exceptions.HTTPError as err:
    # Error 409 means that the user has already been registered for this clien
    if err.response.status_code != 409:
        raise err

shutdown()
return "Client authorized! You can now close this page."
```

Token généré avec l'id de l'utilisateur, on essaye ensuite de l'enregistrer avec l'API accesslink pour avoir accès aux données

ACESSLINK.PY

Objet python qui enveloppe toutes les fonctionnalitées de l'API Polar

```
class AccessLink(object):
    """Wrapper class for Polar Open AccessLink API v3"""

    def __init__(self, client_id, client_secret, redirect_url=None):
        if not client_id or not client_secret:
            raise ValueError("Client id and secret must be provided.")

        self.oauth = OAuth2Client(url=ACCESSLINK_URL,
                                  authorization_url=AUTHORIZATION_URL,
                                  access_token_url=ACCESS_TOKEN_URL,
                                  redirect_url=redirect_url,
                                  client_id=client_id,
                                  client_secret=client_secret)

        self.users = endpoints.Users(oauth=self.oauth)
        self.pull_notifications = endpoints.PullNotifications(oauth=self.oauth)
        self.training_data = endpoints.TrainingData(oauth=self.oauth)
        self.physical_info = endpoints.PhysicalInfo(oauth=self.oauth)
        self.daily_activity = endpoints.DailyActivity(oauth=self.oauth)
```

Pour chaque demande effectuable avec l'API (pour le retour de différente donnée) un objet a été créé. L'objet Accesslink récupère les données avec tous les endpoints qui permettent les transactions avec l'API. Il gére également l'authentification pour avoir accès à l'API.

ENDPOINTS/DAILY_ACTIVITY.PY

Objet qui permet de récupérer toutes les données concernant l'activitée journalière, il utilise l'objet DailyActivityTransaction pour effectuer la transaction avec l'API

```
class DailyActivity(Resource):

    def create_transaction(self, user_id, access_token):
        """Initiate daily activity transaction

        Check for new daily activity and create a new transaction if data is available.

        :param user_id: id of the user
        :param access_token: access token of the user
        """
        response = self._post(endpoint="/users/{}/activity-transactions".format(user_id),
                              access_token=access_token)
        if not response:
            return None

        return DailyActivityTransaction(oauth=self.oauth,
                                         transaction_url=response["resource-uri"],
                                         user_id=user_id,
                                         access_token=access_token)
```

ENDPOINTS/DAILY_ACTIVITY_TRANSACTION.PY

Objet pour l'activitée journalière qui effectue les demandes à l'API pour récupérer les données souhaitées.

```
class DailyActivityTransaction(Transaction):

    def list_activities(self):
        """Get a list of activity resource urls in the transaction"""
        return self._get(endpoint=None, url=self.transaction_url,
                         access_token=self.access_token)

    def get_activity_summary(self, url):
        """Get user's activity summary from the transaction

        :param url: url of the activity entity
        """
        return self._get(endpoint=None, url=url,
                         access_token=self.access_token)

    def get_step_samples(self, url):
        """Get activity step samples

        :param url: url of the activity entity
        """
        return self._get(endpoint=None, url=url+"/step-samples",
                         access_token=self.access_token)
```

ENDPOINTS/PHYSICAL_INFO.PY

Objet qui permet de récupérer toutes les données concernant les informations physiques, il utilise l'objet PhysicalInfoTransaction pour effectuer la transaction avec l'API

```python
class PhysicalInfo(Resource):

    def create_transaction(self, user_id, access_token):
        """Initiate physical info transaction

        Check for new physical info and create a new transaction if data is available.

        :param user_id: id of the user
        :param access_token: access token of the user
        """
        response = self._post(endpoint="/users/{}/physical-information-transactions".format(user_id),
                              access_token=access_token)
        if not response:
            return None

        return PhysicalInfoTransaction(oauth=self.oauth,
                                       transaction_url=response["resource-uri"],
                                       user_id=user_id,
                                       access_token=access_token)
```

ENDPOINTS/PHYSICAL_INFO_TRANSACTION.PY

Objet pour les informations physiques qui effectue les demandes à l'API pour récupérer les données souhaitées.

```python
class PhysicalInfoTransaction(Transaction):

    def list_physical_infos(self):
        """Get a list of physical info resource urls in the transaction"""
        return self._get(endpoint=None, url=self.transaction_url,
                         access_token=self.access_token)

    def get_physical_info(self, url):
        """Get user's physical information from the transaction

        :param url: url of the physical info entity
        """
        return self._get(endpoint=None, url=url,
                         access_token=self.access_token)
```

ENDPOINTS/TRAINING_DATA.PY

Objet qui permet de récupérer toutes les données concernant les données d'entrainements, il utilise l'objet TrainingDataTransaction pour effectuer la transaction avec l'API.

```python
class TrainingData(Resource):

    def create_transaction(self, user_id, access_token):
        """Initiate exercise transaction

        Check for new training data and create a new transaction if data is available.

        :param user_id: id of the user
        :param access_token: access token of the user
        """
        response = self._post(endpoint="/users/{}/exercise-transactions".format(user_id),
                              access_token=access_token)
        if not response:
            return None

        return TrainingDataTransaction(oauth=self.oauth,
                                       transaction_url=response["resource-uri"],
                                       user_id=user_id,
                                       access_token=access_token)
```

ENDPOINTS/TRAINING_DATA_TRANSACTION.PY

Objet pour les données d'entrainements qui effectue les demandes à l'API pour récupérer les données souhaitées. Il peut récupérer :

- La liste des exercices effectués
- Un résumé des séances
- Les différentes zones de rythme cardiaque
- Un résumé des séances en format .gpx ou .tcx

```python
class TrainingDataTransaction(Transaction):

    def list_exercises(self):
        """Retrieve list of urls to available exercises

        After successfully initiating a transaction, training sessions included
        within it can be retrieved with the provided transactionId.
        """
        return self._get(endpoint=None, url=self.transaction_url,
                         access_token=self.access_token)

    def get_exercise_summary(self, url):
        """Retrieve training session summary data

        :param url: url of the exercise entity
        """
        return self._get(endpoint=None, url=url,
                         access_token=self.access_token)

    def get_heart_rate_zones(self, url):
        """Retrieve heart rate zones in training session

        :param url: url of the exercise entity
        """
        return self._get(endpoint=None, url=url+"/heart-rate-zones",
                         access_token=self.access_token)
```

ENDPOINTS/PULL_NOTIFICATIONS

Objet qui permet de récupérer la liste des données disponibles.

```python
class PullNotifications(Resource):

    def list(self):
        """List available data

        Get list of available exercises and activities for users
        """
        return self._get(endpoint="/notifications")
```

ENDPOINTS/TRANSACTIONS.PY

Objet parent pour toutes les transactions qui seront effectuées avec l'API Polar.

```python
class Transaction(Resource):

    def __init__(self, oauth, transaction_url, user_id, access_token):
        super(Transaction, self).__init__(oauth)
        self.transaction_url = transaction_url
        self.user_id = user_id
        self.access_token = access_token

    def commit(self):
        """Commit the transaction

        This should be done after retrieving data from the transaction.
        """
        return self._put(endpoint=None, url=self.transaction_url,
                         access_token=self.access_token)
```

ENDPOINTS/RESOURCE.PY

Objet parent qui permet de passer premièrement par l'authentification OAuth avant d'effectuer des actions avec l'API.

```python
class Resource(object):

    def __init__(self, oauth):
        self.oauth = oauth

    def _get(self, *args, **kwargs):
        return self.oauth.get(*args, **kwargs)

    def _post(self, *args, **kwargs):
        return self.oauth.post(*args, **kwargs)

    def _put(self, *args, **kwargs):
        return self.oauth.put(*args, **kwargs)

    def _delete(self, *args, **kwargs):
        return self.oauth.delete(*args, **kwargs)
```

ACCESSLINKTOOLS.PY

Objet AccesslinkTools qui regroupe toutes les fonctionnalités liées à l'API. Il utilise les endpoints créé (décrit ci-dessus) pour effectuer les transactions et récupérer les données.

```python
class PolarAccessLink(object):
    """All functionalities for Polar Open AccessLink v3."""

    def __init__(self):
        self.config = load_config(CONFIG_FILENAME)

        if "access_token" not in self.config:
            print("Authorization is required. Run authorization.py first.")
            return

        self.accesslink = AccessLink(client_id=self.config["client_id"],
                                     client_secret=self.config["client_secret"])

        self.running = True
```

READ.PY

Script python qui permet de lire les données sur une carte RFID. Il compare l'ID de la carte scannée avec celles enregistrés en base, si la carte est reconnue l'utilisateur à accès aux fonctionnalitées de l'API Polar.

Attente d'une carte, une fois détéctée, la connexion est établie :

```
request = CardRequest(timeout=None, cardType=card_type)
while True:
    time.sleep(0.1)
    # listen for the card
    service = None
    try:
        service = request.waitforcard()
    except CardRequestTimeoutException:
        print("ERROR: No card detected")
        exit(-1)

    # when a card is attached, open a connection
    conn = service.connection
    conn.connect()
```

Requête SQL pour vérifier si l'ID de la carte détéctée existe dans la base de données, si oui l'utilisateur à accès aux fonctionnalitées Polar :

```
sql = "SELECT cardID FROM CLIENTS WHERE cardID = %s"
adr = (uid,)
mycursor.execute(sql,adr)

if mycursor.rowcount==1 :
    print("Success")
    PolarAccessLink()
else:
    print("Refused")
```

CONFIG.YML

Fichier YAML contenant la configuration du client pour avoir accès aux données avec l'API Polar. Il Contient :

- L'id de l'utilisateur
- L'id du client (Pour l'accès)
- Le secret du client
- Le token d'accès (Il est généré avec le fichier authorization.py)

REQUIREMENTS.TXT

Fichier contenant toutes les librairies/modules nécessaires pour le lancement de l'application. Pour l'utilisation :

```
pip install -r requirements.txt
```

MKDOCS.YML

Fichier YAML contenant la configuration pour Mkdocs.

2.4 Conclusion

2.4.1 Améliorations possibles

Dans le cadre du travail de semestre, j'ai totalement dissocié la partie donnée avec l'utilisation de l'API Polar et du lecteur RFID de l'interface graphique. L'interface graphique permet de relier tous les différents scripts que j'ai fait pour en faire une application fonctionnelle. La partie interface a été faite dans le cadre du cours PDA, il faudrait donc mettre ces 2 parties ensembles.

Une autre amélioration à explorer serait d'ouvrir l'application à tous les types de montres connectés. Pour le moment uniquement les montres Polar sont acceptées (à cause de l'utilisation de l'API Polar).

Il faudrait également regarder pour avoir la possibilité d'utiliser la montre du client et pas une montre mise à disposition par le coach comme c'est le cas pour l'instant.

2.4.2 Bilan personnel

Ce projet a été très bénéfique pour moi, je n'avais encore jamais utilisé l'API Polar ou encore un lecteur NFC et j'ai maintenant à disposition pas mal d'objet Python que je peux directement réutiliser. J'ai rencontré pas mal de problème et ça m'a permis de préparer pas mal de point pour mon travail de diplôme. J'ai trouvé très intéressant de travailler sur ces points car ils sont essentiels pour mon travail de diplôme et j'espère vraiment pouvoir avoir quelque chose de fonctionnel à la fin.

3. Logbook

3.1 Jeudi 09 Décembre 2021

Je décide de tester un exemple d'utilisation de l'API Accesslink de Polar en python, le script n'arrive pas a se lancer avec une erreur dans le loader du fichier de config yaml. Il y avait juste un problème avec la version de Pyyaml, certaine installation change sa version et elle n'était plus bonne pour le script, j'ai réinstallé pyyaml avec la version 5.4

```
pip install pyyaml==5.4.1
```

Après une discussion avec M.Aigroz sur mon sujet, nous avons parler de l'import des fichiers excel (Les programmes que les coachs pourront upload), les données seront converties en json pour être plus facilement traitable ensuite.

Excel to json

3.2 Jeudi 16 Décembre 2021

L'API Polar étant un des éléments les plus importants pour mon application, j'aimerai essayer de faire en sorte d'avoir une application en python capable d'afficher les infos de l'utilisateur Polar lorsque le lecteur RFID scan la carte de l'utilisateur.

J'ai amené une montre Polar M400 de chez moi pour pouvoir avoir plusieurs montre pour tester l'API. Je vais prendre les 2 montres et enregistrer des données d'entrainement pendant les vacances.

L'idée pour mon travail de diplôme est d'avoir 1 seul compte Polar (Le compte de l'admin de la salle de sport) qui serait connecté à plusieurs montres dans la salle. Lorsqu'un client arrive et badge sa carte de membre (carte RFID), une montre lui est assigné pour la session d'entrainement (elle prend fin lorsque le client badge sa sortie).

3.3 Jeudi 23 Décembre 2021

J'ai repris le projet "CrowMagic" auquel j'ai participé l'année passée dans le cadre des ateliers décloisonnés. Le projet utilisait les cartes RFID et je voulais reprendre un peu l'utilisation des cartes pour mon projet.

J'ai installé VNC Viewer pour utiliser le CrowPi directement sur mon PC.

Chaque carte RFID possède un id, dans un json je définis les valeurs dont j'ai besoin qui sont associées à l'id de la carte. Problème avec l'import de RPi.GPIO, je vais regarder l'install ou si c'est un problème de version.

3.4 Jeudi 13 Janvier 2021

La malette CrowPi ne fonctionne plus, je pense que jeudi dernier avant les vacances quelqu'un a du debranché la malette. J'ai prit la deuxième malette et j'ai reconfiguré VNC pour pouvoir avoir accès. Je créer un Trello pour gérer la roadmap du projet. J'ai commencé à remplir les backlogs et j'ai défini un niveau d'importance pour les tâches. Bug avec vscode sur le CrowPi, vscode n'arrive pas à se lancer comme il faut (Affichage). J'ai commencé à créer un fichier json contenant les données des cartes RFID et le programme pour lire les infos des cartes. Lorsque le problème avec vscode sera réglé je pourrai tester le programme. L'après-midi M.Garcia m'a prêté un lecteur NFC, je vais essayer de le tester d'ici jeudi prochain pour essayer de remplacer la malette CrowPi.

3.5 Jeudi 20 Janvier 2022

Je me suis documenté sur le lecteur NFC ACR122U afin de l'utiliser avec les cartes RFID, j'ai découvert la librairie python pyscard qui permet d'utiliser les lecteurs NFC. A l'aide de la documentation j'ai pu faire un petit programme qui lit la carte scannée et affiche son UID. Malheureusement, pour l'utilisation de cette libraire fonctionne sur le fait qu'une connexion doit être établie avec une carte donc si aucune carte n'est placée à l'avance sur le lecteur le programme crashera car il n'aura aucune carte pour créer la connexion. Une fonctionnalité permet de mettre un timeout et d'attendre qu'une carte soit scannée mais un nombre de seconde doit être défini avant que le programme ne se ferme.

Sachant qu'un nombre de seconde d'attente doit être défini, je ne pense pas que cette approche conviendrait pour mon projet. Je décide donc de repartir sur la malette CrowPi et de réinstaller une version plus récente de raspberryOS.

3.6 Jeudi 27 Janvier 2022

J'ai discuter un peu avec M.Aigroz par rapport à mon problème avec le lecteur NFC et la connexion avec la carte. Je revois donc toute la documentation et je finis par trouver un moyen de faire fonctionner le lecteur. J'ai réussi à avoir un fichier json contenant les ids des cartes RFID et j'utilise le lecteur pour comparé les ids des cartes scannées avec les ids stockées dans le json. Il faudrait maintenant que j'arrive à intégrer un appel à l'API Polar lorsque je scanne la carte. Depuis les vacances, j'ai eu le temps d'ajouter des données d'entrainements sur les montres Polar que j'avais à disposition. Maintenant le problème avec le lecteur NFC réglé je vais commencer la partie sur l'API Polar et la récupération des données.

3.7 Jeudi 03 Février 2022

J'ai avancé sur la documentation, j'ai ajouter la description du lecteur, la librairie pyscard ainsi que l'API Polar. J'ai commencé à regarder le fonctionnement de l'API pour essayer de récupérer les données d'entrainements quand la carte RFID est scannée. Il me semble que lorsqu'on récupère les données d'entrainements avec l'API on récupère toutes les données du compte, il faut que je regarde si il est pas possible de récupérer seulement les dernières données.

Rappel : L'authentification pour l'API Polar n'est pas pareil que l'authentification avec les cartes RFID. Il y a 1 compte Polar (celui du coach) qui est relié à plusieurs montres et l'authentification des cartes RFID (client) en est une totalement séparée.

3.8 Jeudi 10 Février 2022

Nous avons réalisé une présentation intermédiaire pour montrer l'avancement de nos projets. Suite à cette présentation, je réalise que jusque là je n'ai fait que de la recherche ou quelques essais mais je n'ai pas encore de POC concrèt.

3.9 Jeudi 24 Février 2022

En m'aidant de la doc Polar, je vais créer un objet qui me permettra de récupérer les datas. J'ajouterai ensuite l'objet au programme qui me permet de vérifier les cartes RFID pour récupérer les données lorsqu'on s'identifie avec la carte RFID. Je vais également créer une base de données pour enregistrer quelques données.

3.10 Jeudi 3 Mars 2022

Malade (Covid-19)

3.11 Jeudi 10 Mars 2022

Le rendu du projet est fixé à la semaine prochaine. J'ai fait l'update du firmware de la montre Polar. En essayant de lancer le programme chez moi, je me rends compte que je n'avais pas prit en compte quelques "requirements" pour pouvoir utiliser la librairie **pyscard**. Pour pouvoir installer pyscard, il faut préalablement installer Swig et l'ajouter dans le PATH (Variables d'envirronements). Swig étant un outil qui permet de connecter des librairies écrites en C ou en C++ à d'autres langages comme le Python

3.12 Jeudi 17 Mars 2022

Dernier jour pour ce deuxième travail de semestre, j'effectue les derniers ajustements sur la documentation technique. Ce travail de semestre a été bien plus utile que le premier car j'ai réussi à préparer plusieurs classes Python que je pourrai réutiliser lors de mon travail de diplôme. En travaillant là-dessus, je suis parvenu a éclaircir la plupart des points d'inquiétude que j'avais en vue du travail de diplôme.