Prosopapp : création d'une application de requêtage d'une base prosopographique

Compte-rendu des étapes du projet

Le projet Prosopapp

- A propos

L'ambition de Prosopapp est de proposer un outil permettant d'interroger et de consulter une base de donnée prosopographique, ayant pour objet les enseignants chercheurs et les professeurs de l'Ecole Nationale des Chartes.

Ce projet est né de l'envie de produire une interface similaire à la base prosopographique en ligne de l'EPHE (https://prosopo.ephe.fr/). Dans l'idéal, le travail effectué pourrait être réutilisé pour servir de base à la création d'un dictionnaire prosopographique sur le site internet de l'Ecole des Chartes. En effet, s'il existe déjà un annuaire qui recense professeurs, élèves et étudiants, une interface comme celle que nous proposons permettrait au public d'accéder à des données beaucoup plus précises sur les individus : le titre des thèses, leurs domaines d'étude, leurs distinctions. De même, il pourrait être intéressant d'avoir une base qui recenserait également les anciens chartistes, comme nous l'avons proposé dans notre base de données test.

- Comment fonctionne notre application ?

Notre projet est développé avec Python 3. Il utilise également Bootstrap pour la mise en forme graphique. Pour l'installer en local, il est nécessaire d'ajouter les librairies présentes dans le fichier requirements.txt via la commande : pip install -r requirements.txt.

Pour la tester :

1. Installer python3
2. Créer un environnement virtuel dédié à notre application
3. Cloner, ou Forker puis cloner, notre repository
4. Installer les librairies
5. Vous rendre dans le dossier `prosopapp` qui contient l'application puis la lancer depuis le terminal sur votre serveur local, via la commande `flask run`.

- Crédits

Cette application est réalisé dans le cadre d'un module dispensé par M. Thibault Clérice en Master 2 TNAH à l'Ecole Nationale des Chartes (Paris, Promotion 2019). Il s'agit d'un projet de groupe de fin de semestre, réalisé par quatre étudiantes (Camille BESSE, Camille CARETTE, Louise DUTERTRE, Lucie RONDEAU DU NOYER).

Les étapes de réalisation du projet

Etape 1 : La création d'une base de donnée test

Pour réaliser ce projet d'application, il nous fallait nécessairement une base de donnée contenant un nombre suffisant de données que nous pourrions requêter par la suite. La création de cette base de donnée n'étant pas au coeur de notre projet, nous avons opté pour certains choix plus aisés à mettre en place vu le délai de développement qui nous était imparti.

1. L'obtention des données

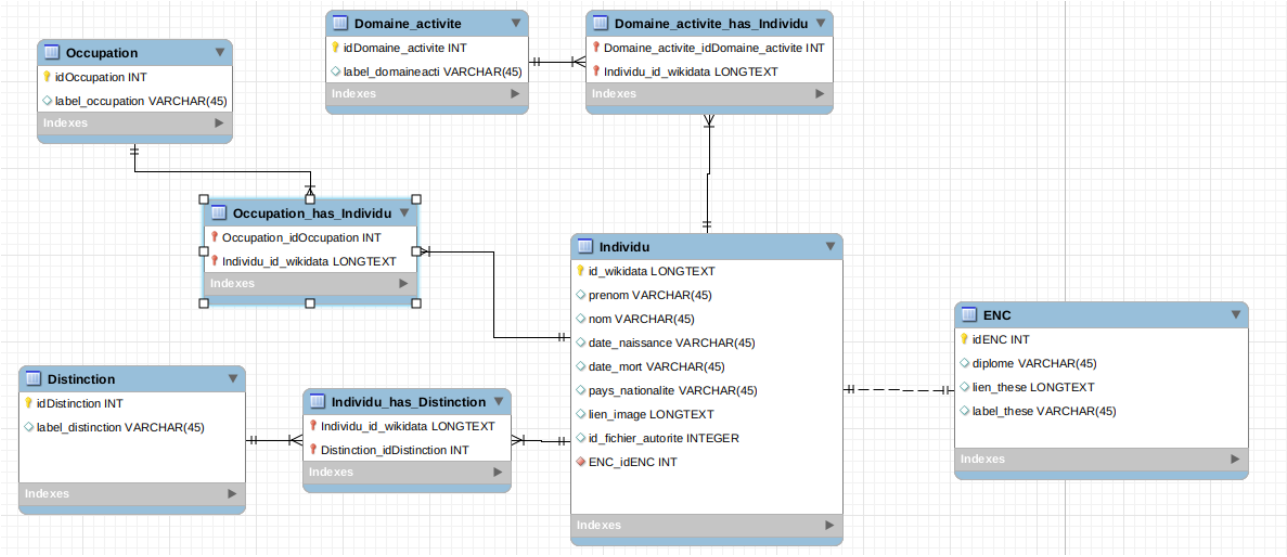
Les données tests utilisées pour cette application ont été recueillies par le biais de requêtes SparQL sur Wikidata. Ce travail a permis de collecter des informations sur une quarantaine d'individus, toutes périodes confondues. Les fichiers csv issus de ces requêtes ont été mergés et nettoyés grâce à Dataiku en un fichier de sortie final, contenant les champs suivants :

- Prénom (Q202444)
- Nom (Q1084)
- Nom complet
- Pays de Nationalité (P27)
- Date de naissance (P569)
- Date de mort (P570)
- Identifiant fichier d'authorité
- Occupation (P106)
- Domaine d'activité (P101)
- Diplôme universitaire (P512)
- Distinction reçue (P166)
- Thèse (Q1266946)
- Année de thèse
- Photo (P18)

2. Création et modifications de la base de donnée

Pour les problématiques qui concernent les points ci-dessous, nous avons fait appel à l'expertise de Vincent Jolivet, responsable de la mission projets numériques à l'Ecole des Chartes

Nous avions dans un premier temps envisagé de convertir directement notre fichier csv pour en faire une base de données SQLite, mais nous nous sommes heurtées à un problème de compatibilité du logiciel DB Browser avec notre fichier. Après avoir considéré la possibilité d'écrire un script python pour charger les données de fichier dans une base de données, nous avons finalement opté pour la solution avec laquelle nous étions la plus familière : écrire les commandes directement en SQL pour créer les tables, les relations, et remplir notre base. A noter qu'au préalable, un modèle conceptuel a été réalisé au moyen du logiciel MySQL Workbench, afin de définir clairement la manière dont nous voulions construire notre base de données.



Le premier modèle relationnel de notre BDD, réalisé à l'aide du logiciel MySQL Workbench.

Par souci de commodité, et dans la mesure où la constitution de cette base de donnée n'était pas le coeur de notre travail, nous avions décidé au départ de construire uniquement des relations de type 1-n, à savoir qu'un chercheur ne pouvait avoir par exemple qu'une seule occupation ou distinction. Par la suite, nous avons revu notre modèle à la demande du superviseur de notre projet afin d'y intégrer une relation de type n-m pour le champ occupation, ce qui a nécessité de modifier quelque peu la structure de la base de données.

Attention : Certaines commandes SQL comme `DROP COLUMN` ne sont pas prises en compte par SQLite. Pour modifier la base de données, il fallait donc :

- Utiliser la commande `BEGIN TRANSACTION` ;
- Recréer une nouvelle table individu identique à l'ancienne en supprimant la clé étrangère `occupation_id` (puisque que ce n'est plus une relation 1-n) ;
- Repeupler la nouvelle table avec les informations de l'ancienne via la commande `INSERT INTO` ;
- Faire un `DROP TABLE` pour supprimer l'ancienne table ;
- Créer une table d'association pour signifier la nouvelle relation n-m.

Etape 2 : Définir la structure du site

Il était important de définir la structure du site avant de commencer à travailler sur le code, notamment dans le but de déterminer les pages html contenues dans le dossier template de notre repository.

Architecture du site

- Header commun à toutes les pages avec champ de recherche plein texte.
- Niveau supérieur :

  - Accueil : courte description du projet.
  - Galerie des chercheurs : index illustré renvoyant vers les notices détaillées.
  - Formulaire de recherche avancée.

- Second niveau :

- Pages résultats (recherche plein texte et recherche avancée).
- Notices détaillées.

Le formulaire de recherche

Le formulaire de recherche avancée étant au coeur de notre application, nous avons réfléchi précisément aux champs que nous voulions voir figurer dans cette fonctionnalité.

- Recherche simple
- Nationalité : menu déroulant
- Domaine d'activité : menu déroulant
- Occupation : menu déroulant
- Distinction : menu déroulant
- Diplôme : menu déroulant
- Thèse : plein texte
- Id autorité : plein texte
- Mort : année exacte / avant / après (plein texte x 3)
- Naissance : année exacte / avant / après (plein texte x 3)
- Thèse : année exacte / avant / après (plein texte x 3)

A noter que pour permettre la recherche plein texte en ce qui concerne les dates, il a fallu modifier leur format car elles étaient normalisées.

Etape 3 : Construire une arborescence de fichiers pour rendre l'application fonctionnelle

Pour l'arborescence des fichiers de notre application, nous nous sommes basées sur les recommandations délivrées par Thibault Clérice dans la cadre du module Python, complétées par celles du 'Flask Mega-Tutorial' de Miguel Grinberg (accessible en ligne).

```
prosopochartes
├── base_csv
│   ├── Compilation.csv
│   └── Les codes qui ont permis la création des différents fichiers csv.pdf
├── base_sqlite
│   └── prosopochartes.sqlite
├── Compte_rendu_final_projet_python.md
├── comptes_rendus_seances
│   ├── 2019_03_04.md
│   ├── 2019-03-05.md
│   ├── 2019_03_06.md
│   ├── 2019_03_07.md
│   ├── 2019_03_08.md
│   └── 2019_14_03.md
├── LICENSE
├── pas_a_pas_github.md
├── prosopapp
```

```
|   ├── app
|   |   ├── app.py
|   |   ├── constantes.py
|   |   ├── errorhandler.py
|   |   ├── __init__.py
|   |   ├── modeles
|   |   |   ├── donnees.py
|   |   |   ├── __init__.py
|   |   |   └── __pycache__
|   |   |       ├── donnees.cpython-36.pyc
|   |   |       └── __init__.cpython-36.pyc
|   |   ├── __pycache__
|   |   |   ├── app.cpython-36.pyc
|   |   |   ├── constantes.cpython-36.pyc
|   |   |   ├── errorhandler.cpython-36.pyc
|   |   |   ├── __init__.cpython-36.pyc
|   |   |   └── routes.cpython-36.pyc
|   |   ├── routes.py
|   |   ├── statics
|   |   |   ├── bootstrap
|   |   |   |   ├── css
|   |   |   |   |   ├── bootstrap.css
|   |   |   |   |   ├── bootstrap.css.map
|   |   |   |   |   ├── bootstrap-grid.css
|   |   |   |   |   ├── bootstrap-grid.css.map
|   |   |   |   |   ├── bootstrap-grid.min.css
|   |   |   |   |   ├── bootstrap-grid.min.css.map
|   |   |   |   |   ├── bootstrap.min.css.map
|   |   |   |   |   ├── bootstrap-reboot.css
|   |   |   |   |   ├── bootstrap-reboot.css.map
|   |   |   |   |   ├── bootstrap-reboot.min.css
|   |   |   |   |   └── bootstrap-reboot.min.css.map
|   |   |   |   └── js
|   |   |   |       ├── bootstrap.bundle.js
|   |   |   |       ├── bootstrap.bundle.js.map
|   |   |   |       ├── bootstrap.bundle.min.js
|   |   |   |       ├── bootstrap.bundle.min.js.map
|   |   |   |       ├── bootstrap.js
|   |   |   |       ├── bootstrap.js.map
|   |   |   |       ├── bootstrap.min.js
```

```
│   │   │   │       └── bootstrap.min.js.map
│   │   │   └── Images
│   │   │       ├── anonyme2.jpg
│   │   │       ├── anonyme.jpg
│   │   │       ├── carousel1.png
│   │   │       ├── carousel2.png
│   │   │       ├── carousel3.png
│   │   │       ├── EcoleChartes.png
│   │   │       ├── ENC.jpg
│   │   │       ├── ENC_Logo.png
│   │   │       └── logochartes.jpg
│   │   └── templates
│   │       ├── headerfooter.html
│   │       ├── pages
│   │       │   ├── accueil.html
│   │       │   ├── chercheurs.html
│   │       │   ├── erreurs
│   │       │   │   ├── 404.html
│   │       │   │   ├── 410.html
│   │       │   │   └── 500.html
│   │       │   ├── noticechercheur.html
│   │       │   ├── recherche.html
│   │       │   ├── resultats_avances.html
│   │       │   ├── resultats.html
│   │       │   └── telechargement.html
│   │       └── partials
│   │           └── metadata.html
│   ├── prosopochartes.sqlite
│   ├── __pycache__
│   │   └── run.cpython-36.pyc
│   └── run.py
├── README.md
└── requirements.txt
```

Schéma de l'architecture de l'application.

A noter que pour obtenir une architecture fonctionnelle, et éviter les bugs, tous les fichiers `__init__.py` ont été laissés vierges.

Etape 4 : Ecriture du fichier donnees.py

Pour interagir avec des bases de données SQL, nous avons utilisé **SQLAlchemy** ainsi que son adaptation pour Flask (flask sqlalchemy), qui permettait de simplifier l'écriture des requêtes dans notre application. SQLAlchemy est un toolkit open source SQL et un mapping objet-relationnel (ORM) écrit en Python.

- Rappel

Qu'est-ce qu'un ORM ? Un ORM est un système qui permet d'indiquer que tel objet est une table de la base de données. A noter qu'avec ce fonctionnement, une table est appelée une `classe`, et chaque objet est une ligne de notre table. **Pourquoi utiliser un ORM ?**

- Possibilité de changer plus aisément le moteur de base de données : les fonctions seront traduites dans la syntaxe choisie.
- L'écriture des requêtes est plus facile. **Cependant** :
- C'est une couche qui alourdit la communication entre notre application et notre base de données.
- L'ORM est conçu pour communiquer avec un langage commun et ne prend pas en charge les fonctionnalités particulières de chaque langage.

- Déclaration des modèles

L'écriture du fichier donnees.py était essentiel pour que nous puissions requêter notre base de données grâce à SQL Alchemy. La déclaration des modèles a été réalisée suivant la documentation de SQL Alchemy, d'abord pour ce qui concernait les relations one to one (1-n), puis les relations many to many (n-m) dans un deuxième temps.

Même si la documentation sur ce point semblait assez claire, il fut nécessaire de remodifier un certain nombre de fois la version initiale de ce fichier, car des erreurs de syntaxes empêchaient la bonne exécution des requêtes. Les messages d'erreur affichés par le terminal nous ont heureusement permis de nous débuguer à chaque fois.

Etape 5 : Création des fonctions dans le fichier routes.py

L'écriture des fonctions étant au coeur de la conception de notre application, il s'agit logiquement de la partie qui nous a occupé le plus de temps.

Le fichier routes.py contient ainsi 6 fonctions :

- `accueil()` : Renvoie vers la paghe d'accueil de notre application.
- `chercheurs()` : Cette fonction permet d'afficher certains champs sur la page qui recense les notices de tous les chercheurs. C'est une version en quelque sorte résumée de la notice complète du chercheur.
- `recherche()` : Permet de créer le formulaire de recherche avancée.
- `resultats()` : Permet d'effectuer une recherche plein-texte dans toutes les classes de la base de données.
- `resultats_avances()` : Permet d'effectuer une recherche avancée sur la base de données en requêtant des champs spécifiques.
- `noticechercheur()` : Permet d'afficher une notice détaillée pour chaque chercheur.

En dehors des fonctions qui permettent simplement de rediriger vers les pages html de notre application (ou d'en générer une automatiquement comme pour `noticechercheur()`), celles qui ont nécessité le plus de temps à concevoir sont les deux fonctions `resultats()` et `resultats_avances()`, qui nous ont permis de mettre en place le requêtage de notre base de données.

Un premier obstacle important s'est présenté à nous pour l'écriture de la fonction `resultats()` : comment interroger différentes classes de notre base de données au sein d'une même requête ? Après de nombreux tests de syntaxes différentes, et le constat que la méthode `.join()` ne permettait pas de produire des résultats satisfaisants, il nous a été suggéré d'utiliser `.outerjoin()` pour lier les différentes classes entres elles.

Exemple : `Individu.query.outerjoin(Diplome).outerjoin(Distinction)`

A noter que la documentation sql alchemy en ligne sur cette méthode est très succinte, voire inexistante.

Si cette solution a très bien fonctionné dans un premier temps, nous avons rapidement été confrontées à un autre problème lorsqu'il nous a fallu modifier notre base de données en y ajoutant une relation many to many (n-m) pour la table `occupations`. Ayant constaté qu'il n'était plus possible dans ces circonstances d'utiliser `.outerjoin()`, nous avons réalisé différents tests, et c'est finalement les méthodes `.has()` et `.any()` qui nous ont permis de faire fonctionner correctement la fonction `resultats()` (`.has()` : critère == True ? / `.any()`: est-ce qu'au moins un des critères == True ?).

Exemples :
```
Individu.diplome.has((Diplome.diplome_label).like("%{}%".format(motclef)))
Individu.occupations.any((Occupation.occupation_label).like("%{}%".format(motclef)))
```

L'écriture de la fonction `resultats_avances()` a posé des problèmes beaucoup plus délicats du fait de la complexité même de la requête à mettre en place.

Il a fallu d'abord définir une valeur d'attribut `name` pour chaque champs du formulaire de recherche, afin de pouvoir récupérer les valeurs entrées dans le formulaire de recherche (il s'agit du même principe que pour `motclef` dans la fonction `resultats()`).

Pour requêter chaque champ indépendamment les uns des autres, nous avons utilisé des 'if' et non pas 'elif' : en effet, en utilisant 'elif', les conditions ci-desous n'auraient été prises en compte que si la condition précédente n'avait pas été remplie. Or, nous ne souhaitons pas une fonction qui agisse ansi ("si pas de données dans ce champ, voir s'il y en a dans le suivant"), mais au contraire une fonction qui prenne en compte tous les paramètres entrés dans chaque champ du formulaire de recherche avancée, en ajoutant à chaque fois un nouveau filtre à l'état précédent de la variable `requete`.

Dans les conditions qui font référence à une date (type int dans notre base sqlite), nous avons choisi de retyper le texte entré dans le champ du formulaire par l'utilisateur (dans le cas où ce texte est composé de caractères numériques) en integer. En effet, même si les opérateurs >=, == et <= fonctionnent sur des caractères numériques même s'ils sont de type str, les retyper en integer nous

permet par la suite de générer un message d'erreur lorsqu'un caractère qui n'est pas un chiffre est tapé dans ces champs.

A noter que pour cette fonction, nous avons été confrontées aux mêmes changements que pour la précédente du fait de l'ajout d'une relation n-m dans notre base de données.

Pour cette partie qui concerne la construction des fonctions, nous remercions Julien Pilla, développeur à l'ENC, de nous avoir conseillé sur un certain nombre de points relatifs à la syntaxe SQL Alchemy. Merci à lui de sa patience !

Etape 6 : Structurer les pages templates à l'aide de Bootstrap

Pour le design de notre site, nous nous sommes servis des templates Bootstrap, qui nous ont permis de mettre en place rapidement une interface utilisateur minimale. Cette partie n'était pas le coeur de notre projet, c'est pourquoi l'utilisation de ces modèles était recommandée afin de ne pas passer trop de temps sur la conception du design.

Toutes nos pages html se trouvent dans le dossier templates/pages de notre application. A noter que notre fichier `headerfooter.html` qui nous a permis d'intégrer le mêmes header sur chaque page html, se trouve à l'extérieur de l'application.

La difficulté principale a résidé dans le fait de relier les fonctions aux pages pour que notre application soit fonctionnelle. Pour ce faire, nous avons notamment utilisé la méthode `url_for()`, spécifique à flask, qui génère une URL vers un point donné.

Un autre point sur lequel nous avons du être attentives est la valeur de l'attribut `name` de l'élément `<input>`, qui permet de définir une variable, que nous avons réutilisé ensuite dans nos fonctions `resultats()` et `resultats_avances()`. Là aussi, des erreurs de syntaxe ou des oublis ont donné lieu à de nombreux bugs lorsque les fonctions ou les templates ont été modifiés.

Nous n'avons pas rencontré de difficulté majeure pour cette partie, mais la mise en place des templates pour qu'ils fonctionnent correctement avec l'application a nécessité que nous y passions un certain temps.

Quelques difficultés rencontrées durant la réalisation de ce travail

- Utilisation de Github

L'ensemble de l'équipe s'est heurté à des problèmes de conflit de branche à cause de la génération automatique d'un fichier `.idea` par le logiciel Pycharm. L'ajout au `.gitignore` a résolu ce problème pour certains postes, mais pas pour d'autres (macintosh), il a donc été nécessaire de supprimer systématiquement ce fichier avant chaque push. Avec le recul, il semble qu'il aurait été plus simple d'utiliser directement l'interface graphique prévue par Pycharm pour interagir avec notre repository git en local et nos branche distantes sur github. Toutefois, nous n'avions pas connaissance de cette fonctionnalité au début de notre projet. Globalement, il est à noter que la prise en main de github pour un travail de groupe comme celui-ci a nécessité un temps d'adaptation de la part de tous les membres. Si nous étions formées à utiliser github pour de petits devoirs, la gestion d'un projet de groupe plus complexe s'est avérée beaucoup moins aisée que nous ne le pensions au départ.

- La documentation SQL Alchemy

Sur bien des points, la documentation SQL Alchemy en ligne nous a paru trop succinte pour que nous puissions comprendre véritablement comment contourner nos problèmes. Sur certaines questions précises, nous avons du faire appel à une expertise extérieure pour pallier aux manques de la documentation. Il s'est avéré souvent frustrant de simplement ne pas pouvoir réaliser certaines choses par manque de documentation, d'autant que nous ne manquions pas de volonté pour essayer différentes méthodes.

- Gestion du temps

Une grande partie du temps du projet a été consacrée à faire fonctionner notre application. Bien que préparées, nous nous sommes rendues compte que chaque étape nous demandait un temps très considérable pour être mises en oeuvre, entre le moment où nous écrivions une première version de la fonction et celui où elle était finalement aboutie, pouvaient s'écouler plusieurs journées de travail.

- Déploiement sur Heroku

Malgré de nombreux efforts, le déploiement sur Heroku s'est révélé compliqué, car de nombreux obstacles se sont posés que la documentation n'a pas toujours permis de résoudre. Le tutoriel suivi pour le déploiement peut être consulté à l'adresse suivante : https://realpython.com/flask-by-example-part-1-project-setup/ . En le suivant à la lettre, nous avons pu déployer notre application sur deux URL différentes (une pour continuer à la développer (prosopochartes-pro), et une pour la vitrine (prosopochartes-stage). A noter que pour finaliser le "déploiement", il nous a fallu créer une clé ssh, qui était inexistante. Toutefois ni l'une ni l'autre ne fonctionnent (elles peuvent être consultées pour vérification à l'une ou l'autre des deux URL : prosopochartes-pro.herokuapp.com, ou prosopochartes-stage.herokuapp.com).

Nous avons passé beaucoup de temps à essayer de résoudre les bugs qui pourraient empêcher le déploiement des applications : lorsqu'on entre la commande `heroku logs --tail` dans le terminal, des erreurs plus précises apparaissent. L'une d'entre elle venait de la présence de deux applications liées au même repository, et a nécessité de définir laquelle d'entre les deux URLs devait être runnée par défaut pour l'exécution d'une commande. Une fois cette erreur résolue, l'une des erreur du terminal indiquait la mention suivante : `No web processes running`. D'après la documentation, cette erreur pourrait venir de la valeur des dynos, que j'ai constaté en effet être inexistante grâce à la commande `heroku ps`. Toutefois, la commande `heroku ps:scale web=1`, qui doit permettre de fixer la valeur de 1, renvoie elle-même une erreur (`couldn't find that process type`), que nous n'avons pu résoudre (la documentation indique que cela pourrait venir d'une mauvaise écriture des buildpacks, mais après vérification, l'erreur n'est pas ici de notre côté).

Le déploiement sur Heroku reste donc un échec, même si nous avons espoir de parvenir à réaliser le déploiement en dehors du cadre de ce devoir. Il est utile de souligner à ce sujet que la documentation présente sur le site d'Heroku est réellement insuffisante pour réaliser un déploiement en bonne et due forme. En ce sens, le tutoriel présent sur Flask s'est avéré beaucoup plus complet.

Bilan global

Malgré les difficultés rencontrées, le bilan de ce projet est tout à fait positif en ce qui concerne nos acquis : l'ensemble du groupe a eu le sentiment d'avoir beaucoup progressé dans l'apprentissage de python et du développement web en se confrontant à la "complexité" d'un projet comme celui-ci. Il

faut prendre en compte bien sûr que notre formation ne vise pas à faire de nous des développeuses accomplies, mais qu'il s'agit de maîtriser certains outils pour ne pas être démunies ensuite si nous sommes amenées à les rencontrer dans notre avenir professionnel. En ce sens, ce projet a rempli nos attentes, et nous a permis de renforcer les acquis de ce semestre.