

AMARGER Corentin - AUBRY Nicolas - BRILLET Clément
GAVALDA Jason - JANNOT Pierre - PINAUD Lorenzo

ESIEEbot



Chatbot pour portail des admissions ESIEE

Rapport de Projet E3 ESIEEbot année 2019-2020

Sommaire

Sommaire	1
Introduction	2
Problématique	2
Cahier des charges	2
Conception du bot de chat (IA)	3
La “base de données”:	3
Reconnaissance des questions :	4
Le machine learning :	5
Fonctionnement sur le site :	6
Traitement des données :	7
Conception du site	12
Utilisation de Flask et fonctionnement global du framework :	15
Implémentation du robot sur le site :	17
Réalisation de la partie CSS sans framework :	19
Contenu de la partie article :	21
Réalisation de la partie CSS avec Bootstrap :	21
Maintenance du chatbot (partie web) :	24
Fonctionnement de la maintenance du chatbot :	25
Gestion du projet	30
Git :	30
Jira :	32
Problèmes rencontrés	36
Améliorations possibles	37
Bugs connus	39
Conclusion	42
Remerciements	43
ANNEXES	44

Introduction

Tous les ans à ESIEE Paris, les admissions sont sources de multiples questions. Cependant, malgré l'accessibilité de l'information sur le site de l'école, beaucoup de futurs étudiants appellent les services administratifs qui se retrouvent alors saturés.

De plus, les questions sont souvent répétitives et trouvent leurs réponses facilement sur le site, ce qui provoque une baisse de réactivité face aux questions pertinentes.

Problématique

Suite à de nombreuses recherches non fructueuses d'un chatbot correspondant à leurs critères, ESIEE Paris a proposé ce projet de site hébergeant un chatbot qui permettrait de répondre aux questions lors des admissions et ainsi soulager les services administratifs.

L'objectif est alors de créer un site sur lequel est implémenté un chatbot qui répondra aux questions et dirigera l'utilisateur vers les réponses présentes sur le site de l'école ou le rediriger vers le service d'administration si aucune réponse n'est disponible.

Cahier des charges

Le commanditaire du projet demandait que le chatbot soit hébergé sur un mini-site comportant 4-5 onglets et respectant la nouvelle charte graphique de l'ESIEE, qui est censée être mise en place à la rentrée 2020. Le chatbot doit être capable de répondre à tout un ensemble de questions qu'un élève de lycée désirant entrer à ESIEE Paris en E1, ou des élèves de CPGE, Licences, DUT ou BTS désirant intégrer ESIEE Paris en E3 pourraient se poser. À noter que ce chatbot ne sera accessible qu'à des élèves étant en procédure d'admission avec l'ESIEE, ce chatbot n'est donc pas accessible au grand public. Le site doit également contenir des liens vers le site principal de l'ESIEE Paris ainsi que vers le site de la nouvelle Université Gustave Eiffel. Le chatbot doit être capable de rediriger les utilisateurs vers des liens en rapport avec leurs questions sur le site de l'ESIEE. De plus, il a été demandé de mettre à disposition un moyen

graphique pour mettre facilement à jour les données du chatbot pour l'administration.

Conception du bot de chat (IA)

Les responsables de la partie IA et chatbot ont été Lorenzo PINAUD et Clément BRILLET. De manière similaire aux autres membres du groupe, nous nous sommes renseignés sur différentes possibilités d'implémentation pendant la première semaine, car nous n'avions alors aucune expérience sur le fonctionnement et les principes sous-jacents de l'utilisation d'un chatbot. Nous avons opté pour un bot fait sur python, pour des raisons de praticité, et de mise en place sur une page web. Nous avons trouvé plusieurs possibilités de mise en place d'un chatbot, mais elles reposaient à chaque fois sur le même principe : faire comprendre au bot la phrase de l'utilisateur, pour ensuite y répondre de manière appropriée. Suite au suivi d'une élective sur l'interprétation des phrases en langages naturels (utilisés par les humains, par opposition au langage informatique) de Lorenzo PINAUD, et à l'utilisation de bibliothèques telles que NLTK et tflearn (cette dernière utilisant du machine learning), nous avons choisi d'utiliser le langage Python pour réaliser cette IA, car il réunissait tous les composants dont nous avons besoin, et ce avec des bibliothèques faites pour les tâches que nous souhaitons réaliser.

La "base de données":

Notre premier bot fonctionnait grâce à un fichier .json, ce dernier nous servant de base de données, structurée comme un dictionnaire, et permettant donc un accès rapide aux données (car compatible avec python) directement depuis le programme. De plus, comme nous avons pu le voir suite à la suggestion de M. BERCHER de regarder l'assistant WATSON d'IBM, il existe des chatbots dans le monde professionnel qui utilisent ce même genre de fichier pour stocker leurs données, nous confortant dans notre choix.

La structure du fichier sera ensuite toujours la même : il y a un ensemble d'intentions, qui vont constituer les sujets pour lesquels le bot sera susceptible d'être questionné. Chacune possède 4 éléments (La balise, la catégorie, les modèles, et la réponse.) :

- La balise consiste en un ou quelques mots clés définissant le sujet ;
- La catégorie est utilisée pour savoir quel type d'utilisateur cette intention concerne (dans notre cas, sa provenance, de lycée ou de prépa par exemple).
- Les modèles correspondent à différentes questions, se rapportant au sujet, susceptibles d'être posées par l'utilisateur.
- La réponse sera ce qui est renvoyé à l'utilisateur si l'intention est reconnue, en utilisant la question donnée, et ce grâce aux modèles.

Exemple :

```
{  
  "intentions": [  
    {  
      "balise": "bonjour",  
      "categorie": "general",  
      "modeles": [  
        "Bonjour",  
        "Comment vas tu",  
        "Il y a quelqu'un?",  
        "Bonsoir",  
        "Comment ca va"  
      ],  
      "reponses": [  
        "Bonjour!"  
      ]  
    },  
  ],  
}
```

Reconnaissance des questions :

Pour cela, nous avons décidé d'utiliser NLTK, cette librairie permettant de faire du traitement de langage naturel. Que ce soit pour la question entrée par l'utilisateur ou les modèles dans le fichier .json, les phrases sont d'abord parsées, c'est à dire que chaque mot est séparé. Les mots inutiles au sens de la phrase sont ensuite retirés (tels que les articles par exemple), afin de ne retenir que les mots clés. Ces mots sont ensuite racinisés, cela veut dire que l'on en extrait la racine, premier traitement permettant deux choses : tout d'abord, il corrige certaines fautes car les terminaisons sont enlevées (plus de fautes d'accord ou de conjugaison). Ensuite, il permet de plus facilement extraire le sens d'une phrase, en regroupant plusieurs mots d'une famille sous une même

racine. Par exemple, les mots restaurant et restauration seront identiques pour le bot.

```
import nltk
from nltk.stem.snowball import FrenchStemmer
stemmer = FrenchStemmer()
```

```
#bibliothèque qui permet le traitement de langage
#fonction de racinisation en français
```

Ceci est primordial car le bot veut comprendre le sens de la question pour l'associer à une balise, et n'a donc que faire des règles de grammaire, qu'il ne connaît pas dans tous les cas. Il arrivera donc à comprendre de la même manière "Comment tu t'appelles" et "comment t'appelle-tu ?" (il ne voit pas les fautes précédentes, car il reconnaît la racine "appel").

Pour le fonctionnement du bot on utilise le one hot encoding, pour cela on a besoin de créer un dictionnaire de mots. Le dictionnaire est un tableau où chaque mot différent qui apparaît dans les modèles est défini par un index. Le programme va ensuite faire du one hot encoding, consistant à prendre la liste de mots créée à partir de la question donnée, et de remplir le dictionnaire en fonction (mettre des un ou des zéros qui définissent par leur position la présence ou non d'un certain mot). En faisant ceci le bot peut voir quels mots sont présents dans la phrase qu'on lui donne.

```
def sac_de_mots(p, mots):
    sac = [0 for _ in range(len(mots))]

    p_mots = nltk.word_tokenize(p)
    p_mots = [stemmer.stem(mot.lower()) for mot in p_mots]

    for ph in p_mots:
        for i, m in enumerate(mots):
            if m == ph:
                sac[i] = 1

    return numpy.array(sac)
```

Le machine learning :

Pour la partie entraînement du bot, on crée une liste de tableaux sur la base du one hot encoding, en utilisant les modèles. À cela on associe les balises pour que le bot puisse comprendre ce qu'il est censé trouver en sortie. C'est ces deux listes qu'on appelle "entraînement" et "sortie", qui serviront pour l'apprentissage du bot. C'est à ce moment que vient le machine learning, comme expliqué plus tôt, on utilise tflearn. Le but de cette partie est que le programme crée des modèles prédéfinis qui vont ensuite être reconnus lors de son utilisation sur chacune des pages pour avoir la réponse la plus précise possible. Le bot se rend compte que certains mots ont un poids plus important

que d'autres pour définir une intention. C'est ces mots là qu'il va donc considérer importants et donc dans le modèle mettre un poids plus important. C'est pour cela qu'on définit plusieurs catégories pour une balise unique, comme c'est le cas pour stages, car il ne pourrait pas reconnaître précisément de quels stages la personne veut parler avec juste une question "Comment fonctionnent les stages?", car ils sont différents pour les E1 et les E3. Pour récupérer ces balises, lors de l'entraînement, le programme va récupérer tous les modèles dont il a besoin, car certaines intentions existent sous plusieurs formes, en fonction de où se trouve l'utilisateur sur la page web (home, Post Bac, CPGE, Licence/DUT/BTS), la catégorie de l'intention permettant de faire les distinctions entre chaque :

- "general" pour les balises devant être utilisées pour tous les onglets (et dont la réponse ne change pas d'un onglet à l'autre)
- "home", pour les balises donnant une réponse généraliste, mais pouvant être affinée selon la provenance
- "Post_bac" pour les Post Bac
- "CPGE" pour les CPGE
- "DUT" pour les Licences/DUT/BTS

Nous avons donc quatre modèle différents qui sont créés, un par page du site.

Fonctionnement sur le site :

Sur le site, le bot fonctionne en quatre fonctions (cas où l'amélioration est désactivée) :

- *chat(ent,modele,amelioration)* qui permet d'appeler *traitements_reponse(ent,modele)* qui va traiter l'entrée (la fonction chat permet de dissocier les cas quand l'amélioration est activée, cela sera expliqué dans la partie dédiée).
- *traitement_reponse(ent,modele)* appelle la fonction *choix_modele(ent,modele)*. Cette fonction permet de différencier les modèles et donc de répondre différemment par rapport à la provenance de l'utilisateur. En différenciant les modèles, il initialise le tableau des différentes intentions différemment. Elle permet aussi de prédire le pourcentage d'appartenance aux intentions pour la phrase entrée en paramètre. Mais pour cela il lui faut un tableau type one hot encoding, c'est pour cela qu'elle appelle la fonction *sac_de_mots(p,mots)*

- *sac_de_mots(p,mots)* qui s'occupe de traiter la phrases envoyées en paramètre en parsant les mots puis en racinisant les mots autres que les mots-vides, et de créer un tableau sur le principe du one hot encoding en utilisant les dictionnaires du modèle définit en paramètre (le paramètre mots).
- Finalement, la fonction *traitement_reponse(ent,modele)* récupère l'indice de l'intention la plus probable en utilisant la fonction *argmax()* et renvoie la réponse associée. La chaine de caractère est ensuite marquée avec la fonction *Markup()* de la librairie Flask, pour que lors de la traduction en HTML par Flask les caractères spéciaux comme les balises HTML restent des balises et ne soient pas affichées telles quelles sur le site. C'est avec cela que nous arrivons à afficher des hyperliens ou des radios dans les messages du bot.

Finalement lorsque le bot a trouvé la balise, il renvoie la réponse associée dans le fichier .json au site pour que celui ci l'affiche.

Après quelques semaines, un de nos professeurs nous a indiqué l'existence du bot Watson assistant, cité précédemment. Ce produit permet de faire un chatbot de manière graphique sans besoin de coder. Le fonctionnement de ce bot étant similaire au nôtre, il nous a rassuré et convaincu que nous étions bien sur la bonne voie.

Nous n'avons malgré tout pas choisi de l'utiliser car son implémentation graphique ne nous plaisait pas et aussi parce qu'il est payant, étant un service d'IBM. De plus, l'implémentation graphique du bot est un module se plaçant en bas à droite d'une page, et ne correspondait donc pas au cahier des charges du projet. Pour toutes ces raisons, nous avons choisi de rester sur notre bot codé par nos soins de zéro.

Traitement des données :

Afin d'obtenir une base de données plus conséquente pour le bot, nous avons décidé de réaliser une enquête auprès de tout le personnel de l'ESIEE, notamment les élèves. Cette enquête a pris la forme d'un Google Form que nous avons envoyé par mail en diffusion sur l'ensemble des personnes de l'ESIEE, ainsi l'administration pouvait également y répondre. Nous l'avons commencé comme ceci : "Quelle(s) question(s) avez-vous posée(s) ou auriez aimé poser à l'école au moment des admissions ?". Nous avons alors regroupé les possibles questions par rubrique sur le form :

- Les questions relatives à la scolarité (horaires, cours, évaluations, etc...)
- Les questions relatives à la vie associative (clubs, sports, etc...)
- Les questions relatives à l'aspect économique (frais de scolarité, bourses, logements, restauration, etc...)
- Les questions relatives aux échanges scolaires (stages, départ à l'étranger)
- Les questions relatives aux inscriptions (dossier FAR, Webaurion)
- Toute autre question ne rentrant pas dans ces catégories.

Lors de la rédaction de ce rapport, 105 personnes avaient répondu à cette enquête, formant une base de près de 700 questions.

Il a donc fallu traiter chacune de ces questions en regroupant les doublons et les questions similaires, ce qui a été fait avec le renfort de Nicolas AUBRY et Jason GAVALDA en plus de l'équipe d'IA de Clément BRILLET et Lorenzo PINAUD.

Une fois les regroupements faits, nous avons pu les ajouter dans le fichier .json (donnees.json regroupant notre base de données de questions). Le bot contenant alors une base de données plus solide.

Néanmoins, un défaut subsistait, puisque le bot donnait toujours les mêmes réponses, peu importe l'onglet dans lequel l'utilisateur se trouvait.

C'est à ce moment là que nous est venue l'idée de séparer le chatbot dans les différents onglets, et de l'entraîner spécifiquement pour chaque onglet avec les catégories correspondantes. Ainsi, il était capable de donner une réponse personnalisée à chaque personne selon l'onglet choisi.

Amélioration et validation des réponses :

Une autre suggestion de nos tuteurs était de permettre l'amélioration du bot par l'intermédiaire des questions qui lui étaient posées : ainsi, le bot devait être capable de demander à l'utilisateur si sa réponse lui convenait, et dans le cas contraire, de faire remonter cette nouvelle question à l'administration, afin de contrôler sa pertinence. Ainsi, l'administration n'aurait qu'à valider cet ajout dans le fichier .json en y ajoutant la réponse appropriée.

L'équipe web a donc dû concevoir de nouveaux onglets permettant à l'administration de le faire (voir la partie conception du site pour plus de

détails). Mais suite à cela, il fallait bien que la partie web puisse avoir accès aux données du fichier .json, et cela a été fait par l'intermédiaire d'un panel de fonctions dans le fichier savamment nommée "fonction.py". Celui-ci contient aussi les fonctions qui permettent à l'administration de faire la maintenance de la base de données par des pages web ("Recherche", "Suggestion", "Ajout", "Modification", "Suppression").

Un certain nombre de ces fonctions a été conçu tôt dans le projet, et certaines sont donc devenues obsolètes suite à la mise en place de Flask, mais elles ont été laissées car elles peuvent toujours être utiles. Voici donc une liste de celles utilisées et un bref descriptif de leur utilité :

- ajout_dans_JSON(nomJSON, donnees), qui permet de créer une nouvelle intention qui aurait pu être oubliée, à partir du nom du fichier json et d'une intention déjà construite; utilisée dans "Ajout"
- suppr_dans_JSON(nomJSON, intention), qui permet de supprimer une ancienne balise qui aura perdu son utilité (COVID par exemple dans quelques années), à partir du nom du fichier json et du nom de l'intention à supprimer (donc la balise); utilisée dans "Suppression"
- saveToJSON(nomFichier, donnees, chemin='./'), qui permet de sauvegarder le fichier .json après une modification, à partir du nom du fichier json; utilisée dans "Suggestion", "Ajout", "Modification"
- recupere_balises(nomFichier), qui permet de récupérer toutes les balises pour les montrer à l'utilisateur, pour choisir ce qu'il veut voir, à partir du nom du fichier json et d'une variable contenant les données + le chemin accessoirement; utilisée dans "Recherche", "Modification", "Suppression"
- objet_dans_JSON(nomFichier, balise, objet), qui permet de récupérer les modèles ou la réponse d'une balise, à partir du nom du fichier json et la balise correspondante en plus de l'objet à récupérer ("modeles" ou "reponses"); utilisée dans "Recherche", "Modification"
- format_intentions(balise, modeles, categorie, reponses=[]), qui permet de créer une intention à remettre dans le .json, avec toute les données nécessaire pour faire une balise; utilisée dans "Ajout", "Modification"
- remplace_objet(nomFichier, balise, objet, objets), qui permet de modifier une réponse ou un modèle du .json, à partir du nom du fichier json, la balise concernée, le nom de l'objet en question ("modeles" ou "reponses") et l'objet par lequel on le remplace; utilisée dans "Suggestion", "Modification"

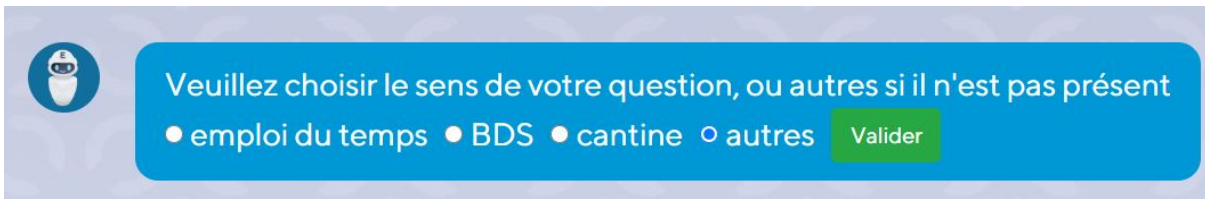
- `stocke(nomTexte, balise, objet)`, qui permet de stocker les suggestions de l'utilisateur dans le fichier texte "`amelioration.txt`", à partir du nom du fichier json, la balise désignée et l'objet (tout le temps une phrase modèle); utilisée dans le chatbot si la variable "`amelioration`" vaut `True`
- `cleanString(string)`, qui est utilisée dans `stocke()` pour une question de sécurité, vu que l'on utilise une phrase de l'utilisateur pour l'ajouter dans notre base de données, utilisée dans le chatbot si la variable "`amelioration`" vaut `True`
- `retirer_suggestion(nomTexte)`, qui permet de retirer la première suggestion de la liste, après traitement par l'utilisateur, à partir du nom du fichier texte; utilisée dans "`Suggestion`"
- `ajout_suggestion(nomFichier, nomTexte, balise, modele)`, qui permet de rajouter la première suggestion de la liste dans la balise correspondante, et après la supprime, à partir du nom du fichier texte, de la balise correspondante et du modèle à ajouter; utilisée dans "`Suggestion`"

Pour que le bot puisse s'améliorer de lui-même, nous avons pensé à rajouter un système de validation, après chaque réponse du bot l'utilisateur pouvait décider si oui ou non la réponse lui allait.

Cette fonctionnalité a été très compliquée à mettre en place car le fonctionnement du site faisait que cette distinction devait se faire au sein de la fonction chat.

C'est pour cela que nous avons modifié la fonction `chat(ent,modele,amelioration)` pour qu'elle différencie les entrées qui correspondent à des questions, et les entrées qui ont pour but de dire si oui ou non la réponse précédente était la bonne. Pour cela on utilise des `if`, pour savoir quand l'utilisateur entre "`non`" ou "`oui`", c'est la seule manière que l'on a trouvée pour différencier une question d'une réaction.

L'idée que nous avons eue est la suivante : si la réponse ne plait pas à l'utilisateur, on lui envoie sous la forme d'un form composé de radios (fait en HTML) les intentions les plus proches trouvées par le modèle, après celle qui vient d'être utilisée comme réponse.



Ainsi l'utilisateur sélectionne une intention, ce qui fait que premièrement le bot lui renvoie la réponse associée à l'intention sélectionnée. De plus, cela écrit dans un fichier nommé amélioration.txt l'intention et la question associée. Ce fichier est utilisé dans la page ajout pour pouvoir ajouter des questions au fichier json, avec toujours le même format qui est :

- la balise
- la phrase suggérée
- une ligne vide de séparation

Exemple :

```
COVID
rentre pour le COVID

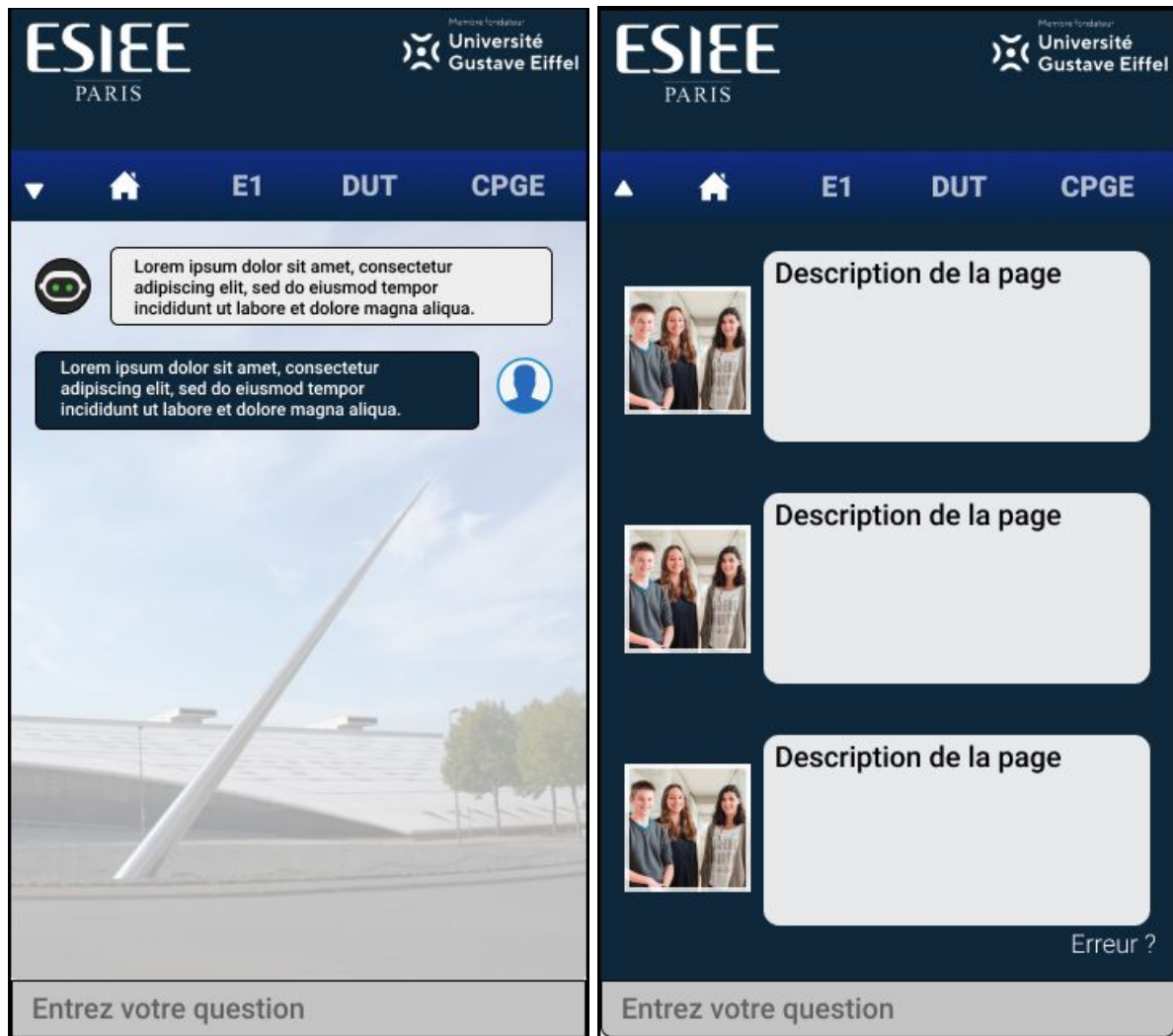
autres
Comment avoir un logement

Echanges internationaux
les departs
```

Conception du site

Pour développer notre site, nous avons dû utiliser le langage HTML et le langage CSS. Étant, pour la plupart, de parfaits débutants dans ce domaine, il a été nécessaire pour nous de se former à leur utilisation au début du projet. Une fois cela fait, nous avons également réalisé une première maquette du site pour nous donner une idée globale de ce à quoi il pourrait ressembler. La modélisation suivante a été réalisée grâce au site Figma :





Dans sa conception, nous avons réalisé que nous pouvions effectuer le découpage des onglets en fonction de l'origine des élèves en procédure d'admission. Ainsi, notre site se découpe en 4 onglets :

- Un onglet "home" qui sert de manière générale à n'importe quel élève,
- un onglet "Post-bac", pour les élèves issus de lycées et désirant intégrer l'ESIEE en première année de Premier Cycle (E1),
- un onglet "CPGE" pour les élèves issus de classes préparatoires et ayant passé avec succès le concours Puissance Alpha nécessaire pour l'admission à ESIEE Paris en première année de Cycle Ingénieur (E3),
- un onglet "Licence/DUT/BTS" pour les élèves issus de ces formations désirant entrer à l'école en première année de Cycle Ingénieur (E3).

Nous avons également décidé que les logos de l'ESIEE et de l'Université Gustave Eiffel (UGE) imposés par la nouvelle charte graphique contiendraient les liens vers les deux sites.

Comme on peut le remarquer sur la maquette ci-dessus, nous avons conceptualisé le découpage de chaque page comme suit :

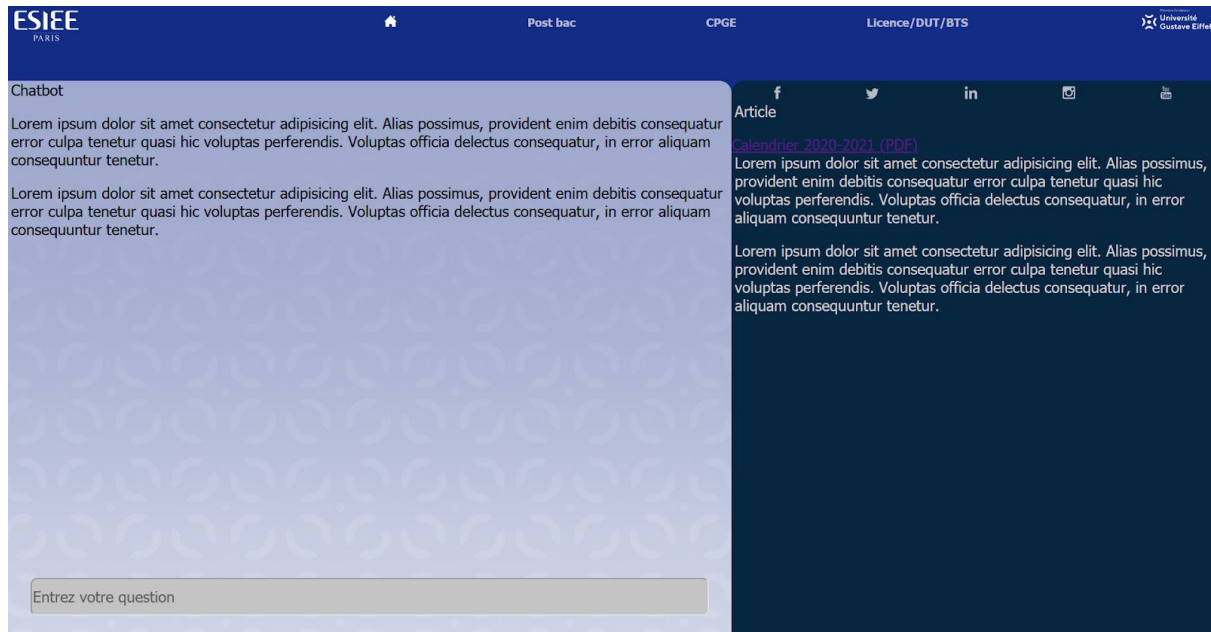
Le header du site doit contenir les logos de l'ESIEE et de l'UGE l'un à gauche, l'autre à droite, ainsi qu'une navbar contenant les liens vers les différents onglets du site.

Le body, lui, se décompose de la façon suivante :

- Le chat avec le bot doit occuper la partie gauche du site : c'est-à dire que l'historique du chat ainsi que le champ où l'utilisateur interagit avec le bot s'y situent.
- La partie droite quant à elle contient une partie nommée "article", qui doit contenir différents liens faciles d'accès pour les réponses des questions les plus posées.

Nous avons commencé le projet avec une équipe "site web" de 4 personnes elle-même séparée en 2 sous-équipes : une équipe centrée sur le HTML et la mise en place globale du site, composée de Nicolas AUBRY et de Jason GAVALDA; et une autre équipe centrée sur la partie graphique du site, et donc le CSS, composée de Corentin AMARGER et de Pierre JANNOT.

Au bout d'une semaine, nous avons réalisé la première maquette du site en HTML et CSS, en y ajoutant les liens vers les réseaux sociaux de l'école. Néanmoins, nous avons remarqué que la navbar était trop surchargée si elle devait contenir à la fois les onglets et les liens vers les réseaux sociaux. Nous avons donc décidé de déplacer ces derniers dans la partie article. Dans le même temps, nous avons réalisé à quel point nous étions peu efficaces en effectuant une telle séparation dans notre groupe "site web" et avons décidé de fusionner les deux sous-groupes.



Nous avons montré cette maquette lors de notre réunion avec nos tuteurs Denis BUREAU et Corinne BERLAND, et ceux-ci l'ont validée.

Dans le même temps est venue la problématique d'implémenter le chatbot sur le site. Celui-ci étant codé en Python, il nous fallait trouver un moyen de l'implémenter sur le site. Suite à plusieurs recherches, nous avons découvert Flask, un framework open-source de développement web en Python, qui nous permet justement d'injecter du code Python sur notre page web, et donc notre chatbot. Nous n'avons donc pas besoin de passer par du JavaScript pour faire fonctionner notre chatbot.

Utilisation de Flask et fonctionnement global du framework :

Nous avons fait le choix de développer le site en utilisant le framework Python Flask plutôt que d'utiliser les langages web classiques (HTML/CSS/PHP/JavaScript) afin de faciliter l'implémentation de l'IA sur le site.

La construction du site s'effectue donc dans un script Python dans lequel nous définissons les "emplacements", sur le site, des différentes pages (ou "routes").

Pour chaque page il existe deux fonctions, une qui sert à afficher la page dans un premier temps, une autre qui sert à afficher la page et d'effectuer des

requêtes 'POST' et 'GET' nécessaires au fonctionnement du chatbot. Chaque fonction renvoie au minimum la page HTML à afficher, et éventuellement des variables à injecter dans le code HTML de la page.

La framework Flask utilise la librairie Jinja2 permettant de programmer dans une page HTML, en effet, nous pouvons utiliser des boucles, des tests, des injections de variables, directement dans le code de la page ce qui nous permet d'afficher certaines parties du site lorsqu'une condition est vérifiée comme l'alerte des cookies par exemple, d'afficher le contenu d'une liste comme l'historique de chat, ou encore changer le titre de la page rapidement.

```
@app.route('/post_bac') # Voir explications au dessus
def postBac():
    # La variable "title" sert juste à afficher le titre de la page
    return render_template('postBac.html', title='Post Bac')

@app.route('/post_bac', methods=['POST', 'GET']) # Voir explications au dessus
def postBac_post():

    quest, rep = page('questions_pb', 'reponses_pb',
                     'chatzone_pb', 'PB', "True")

    return render_template('postBac.html', chat_q=quest, chat_r=rep, cookie_session=request.cookies.get('session'))
```

Exemple de définition de "route" et d'injection de variables

```
<!-- Bloc for permettant d'afficher tout l'historique de chat-->
{% for i in range(chat_r|length) %}
<div class="row justify-content-end pr-2">
    <div class="mw-92">
        <!-- Alerte de couleur "secondaire" (gris) affichant la question de l'utilisateur dans le chat -->
        <div class="question alert alert-secondary" role="alert">
            {{chat_q[i]}}
        </div>
    </div>
</div>
<div class="row justify-content-start pl-2">
    <div class="col-sm-2 col-md-1 col-12">
        <!-- Image du bot ajoutée à chaque réponse de celui-ci dans le chat-->
        
    </div>
    <!-- Alerte de couleur bleu (défini dans style2.css via réponse) affichant la réponse du bot dans le chat-->
    <div class="mw-92">
        <div class="reponse alert" role="alert">
            {{chat_r[i]}}
        </div>
    </div>
</div>
{% endfor %}
</div>
```

Exemple de boucle for dans le code HTML

Par ailleurs, Flask possède aussi un système de block, permettant d'injecter du code HTML dans un autre code HTML. En effet, chaque page étant

globalement similaire il aurait été redondant de copier le code en commun sur chaque fichier.

Ainsi, chaque page du site se base sur un fichier qui s'appelle "base.html" contenant les informations communes aux pages (le style graphique, la structure de la barre de navigation, les différentes "zones" du site...), et sur un autre fichier HTML contenant les différences à injecter dans le fichier de base.

```
<div class="col-8 bg-gradient-light">
  <nav class="navbar navbar-expand-lg navbar-dark">
    <!-- Bouton déroulant les onglets en cas d'utilisation sur une petite fenêtre-->
    <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarNav"
      aria-controls="navbarNav" aria-expanded="false" aria-label="Toggle navigation">
      <span class="navbar-toggler-icon"></span>
    </button>
    <!-- Navbar contenant les liens vers les différents onglets-->
    <div class="collapse navbar-collapse justify-content-around" id="navbarNav">
      <div class="navbar-nav">
        {% block navbar %}
        {%- endblock navbar %}
      </div>
    </div>
  </nav>
</div>
```

Code de la navbar dans base.html avec un block "navbar"

```
{% extends "base.html" %}

<!-- Navbar contenant les liens vers les différents onglets du site-->
{% block navbar %}

<a class="nav-text nav-item nav-link" href="/">Home</a>
<a class="nav-text nav-item nav-link active" href="/post_bac">Post Bac</a>
<a class="nav-text nav-item nav-link" href="/cpge">CPGE</a>
<a class="nav-text nav-item nav-link" href="/licence_dut_bts">Licence/DUT/BTS</a>

{% endblock %}
```

Import du fichier de base dans le code de "postBac.html", exemple d'utilisation de block

Implémentation du robot sur le site :

L'implémentation du robot sur le site s'est effectuée en plusieurs étapes.
D'un point de vue "HTML" brut, une zone de texte est créée, nous récupérons

ensuite ce que l'utilisateur écrit par la méthode 'POST', puis nous l'envoyons au robot via la fonction "chat" du fichier "chat_utilisation_web.py" renommée en "bot" dans le fichier "website.py". Les paramètres de cette fonction sont détaillés dans la sous-partie "Fonctionnement sur le site" de la partie "IA".

Afin de conserver l'historique de chat et de le dissocier entre chaque utilisateur, nous avons mis en place un système de cookies de session en utilisant les fonctions Flask associées.

Ce cookie de session comporte plusieurs champs dont voici la liste :

- questions_pb : champ stockant les questions de l'onglet post bac
- reponses_pb : champ stockant les réponses de l'onglet post bac
- questions_g : champ stockant les questions de l'onglet home (général)
- reponses_g : champ stockant les réponses de l'onglet home
- questions_c : champ stockant les questions de l'onglet cpge
- reponses_c : champ stockant les réponses de l'onglet cpge
- questions_ldb : champ stockant les questions de l'onglet licence dut bts
- reponses_ldb : champ stockant les réponses de l'onglet licence dut bts
- modif : champ stockant le token de modification (expliqué dans la partie concernant la maintenance de la base de données)
- feedback : champ stockant le token de feedback (expliqué dans la partie concernant la maintenance de la base de données)

Ces différents champs sont initialisés par la fonction initCookie() prenant en paramètre le token de modification (initialisé à "False" par défaut).

Le stockage de l'information dans le cookie de session s'effectue comme suit :

1. Initialisation du cookie s'il est inexistant
2. Récupération des données de celui-ci
3. Modification des données
4. Stockage des données dans le cookie

Cette routine s'effectue à chaque requête 'POST' reçue, donc à chaque formulaire HTML envoyé par l'utilisateur au site.

Cette routine se déclenche lors de l'appel de la fonction page() prenant en argument champ_q (le champ de questions dans lequel stocker l'information, différent selon l'onglet), champ_r (le champ de réponses dans lequel stocker l'information différent selon l'onglet), chatzone (le nom de la zone de chat à

utiliser, différente selon l'onglet), botname (le nom du jeu de réponses à utiliser, différent selon l'onglet), feedback (par défaut à "False", servant pour le système d'amélioration des réponses et de suggestions).

Cette fonction page permet la gestion de la sauvegarde des données de chat, mais gère aussi le système de feedback en vérifiant que les conditions pour envoyer le formulaire d'amélioration sont réunies (token de feedback sur True, dernier message enregistré dans le champ de questions valant "non") afin de recueillir les suggestions de l'utilisateur.

Voir l'annexe pour plus d'explications.

Le système de suggestion est décrit plus tard dans la partie "Maintenance du chatbot (partie web)".

Un bouton "Reset" a été ajouté au chatbot afin de supprimer l'historique de chat. Lorsque qu'un appui sur ce bouton est effectué, l'utilisateur est renvoyé sur la route "/clear" du site, qui exécute la fonction clear() sur le cookie de session, le supprimant tout simplement. Cette route "/clear" redirige ensuite l'utilisateur sur la route "/" qui est la route par défaut.

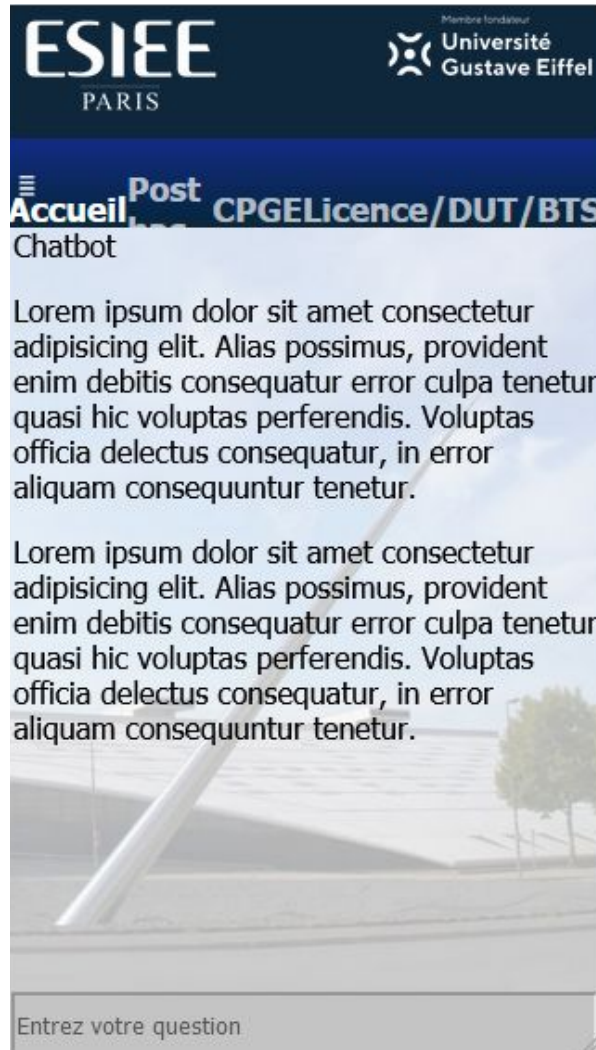
Le système d'acceptation des cookies fonctionne plus ou moins de la même façon : la page vérifie s'il existe un cookie de session, si non alors on affiche le code d'acceptation des cookies et lorsque le bouton "Accepter" est cliqué l'utilisateur est redirigé sur la route "/cookiegen" qui génère le cookie de session et redirige ensuite sur la route "/". Cette fonctionnalité est cependant buggée à l'heure actuelle.

Réalisation de la partie CSS sans framework :

Une autre grande problématique pour le site était de le rendre responsive, c'est-à-dire qu'il soit capable tout seul d'adapter l'affichage peu importe le support utilisé : un ordinateur ou un téléphone portable.

Tout d'abord, afin de faire un site responsive, nous avons décidé d'utiliser des flexbox afin de pouvoir placer les éléments selon notre besoin ainsi qu'une taille des éléments en pourcentage de la taille de l'écran. Il nous fallait cependant trouver un moyen de différencier l'affichage des éléments

entre la version ordinateur et téléphone puisque certaines parties telles que la barre de navigation ne fonctionnaient pas sur un petit écran :



Plusieurs options étaient disponibles pour créer un site responsive. Nous avons d'abord pensé à faire une page différente entre la version ordinateur et la version téléphone en observant le système d'exploitation utilisé par le visiteur, mais cela voulait aussi dire que le même code serait dupliqué en deux fichiers HTML afin de leur donner un fichier CSS différent.

Nous avons finalement décidé d'utiliser la ligne de code "*@media screen and (max-width: x pixels)*" qui se traduit par une condition de largeur d'écran maximale de x. Ainsi, en observant les tailles où le site pouvait poser problème, il nous suffisait de changer quelques parties du CSS telles que les emplacements de certains éléments (par exemple la partie article appartenant ou non à un menu déroulant) afin de rendre le site lisible. C'est cette solution qui nous a permis par la suite de produire un affichage ordinateur/téléphone proche de la maquette proposée. Cependant, cette solution ne permettait pas

de traiter tous les problèmes amenés par les tailles d'écran, puisque nous ne pouvions corriger que les problèmes que nous trouvions. De plus, cela nous obligeait à traiter tous les cas possibles (ce qui n'est techniquement pas réalisable) et cela alourdissait le code CSS.

Contenu de la partie article :

Enfin la dernière grande problématique pour le site web était de décider quel contenu mettre dans la partie article, sachant qu'à ce moment-là notre base de données n'était pas importante.

Nous avons donc décidé d'y intégrer le calendrier de l'année 2020-2021, ainsi que les différents programmes de chaque année. Seulement, lors de la réunion suivante avec nos tuteurs, ceux-ci nous ont indiqué qu'étant donné la provenance de ces ressources (issues de l'intranet de l'ESIEE), il valait mieux ne pas les mettre sur le site, afin de respecter un certain degré de confidentialité (l'intranet n'étant logiquement accessible que par le personnel de l'ESIEE, élèves ou encadrants). Nous avons donc décidé d'y intégrer plutôt des liens vers différentes rubriques utiles du site de l'ESIEE, en fonction des questions qui nous seront le plus demandées dans notre enquête (voir partie développement de l'IA).

Lors de cette même réunion, nos tuteurs nous ont également encouragé à utiliser un autre framework pour le développement du site, afin d'avoir un site totalement adaptatif plutôt que de traiter ce problème avec un système de conditions sûrement incomplet.

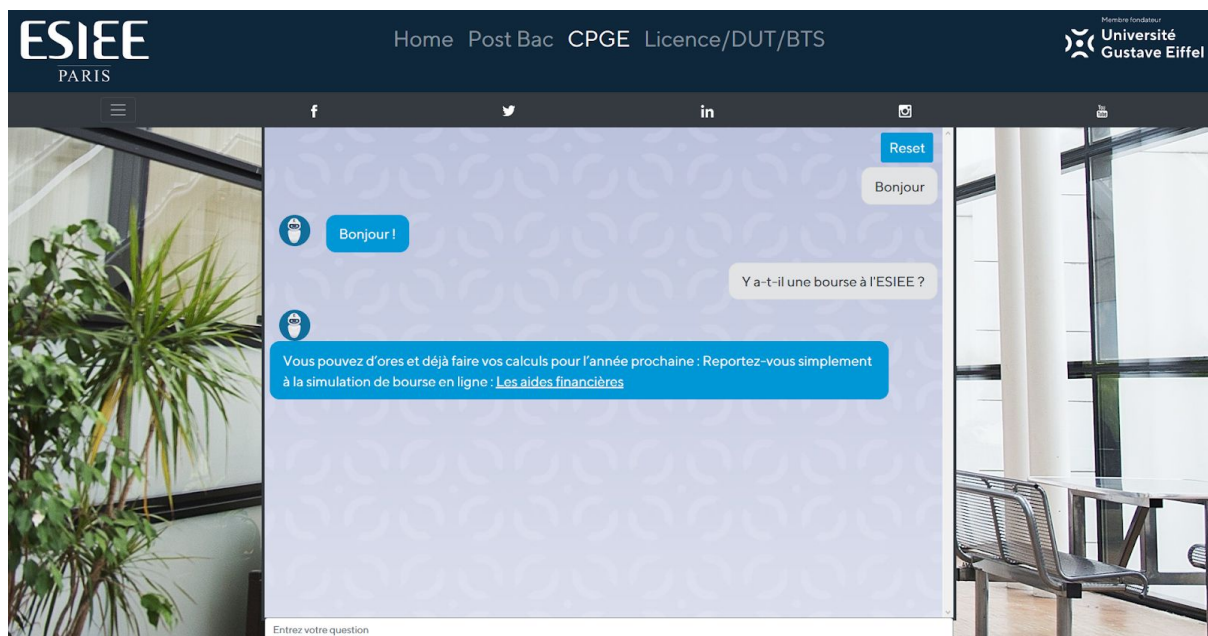
Réalisation de la partie CSS avec Bootstrap :

Nous avons donc choisi d'utiliser le framework Bootstrap pour sa capacité à créer des éléments qui changent de taille ou de disposition automatiquement en fonction de la taille de l'écran. Utilisant Flask pour la partie équivalente au PHP, nous avons utilisé Flask-Bootstrap4 pour utiliser Bootstrap. Celui-ci possède en effet de nombreuses classes donnant directement un certain aspect aux éléments HTML et le site de Bootstrap permet de récupérer le code HTML correspondant à des éléments particuliers que l'on peut adapter (boutons, pages modales, alertes, etc). Ne connaissant pas ce framework, il nous a fallu quelques jours afin de le découvrir et d'apprendre à utiliser la plupart de ses fonctionnalités.

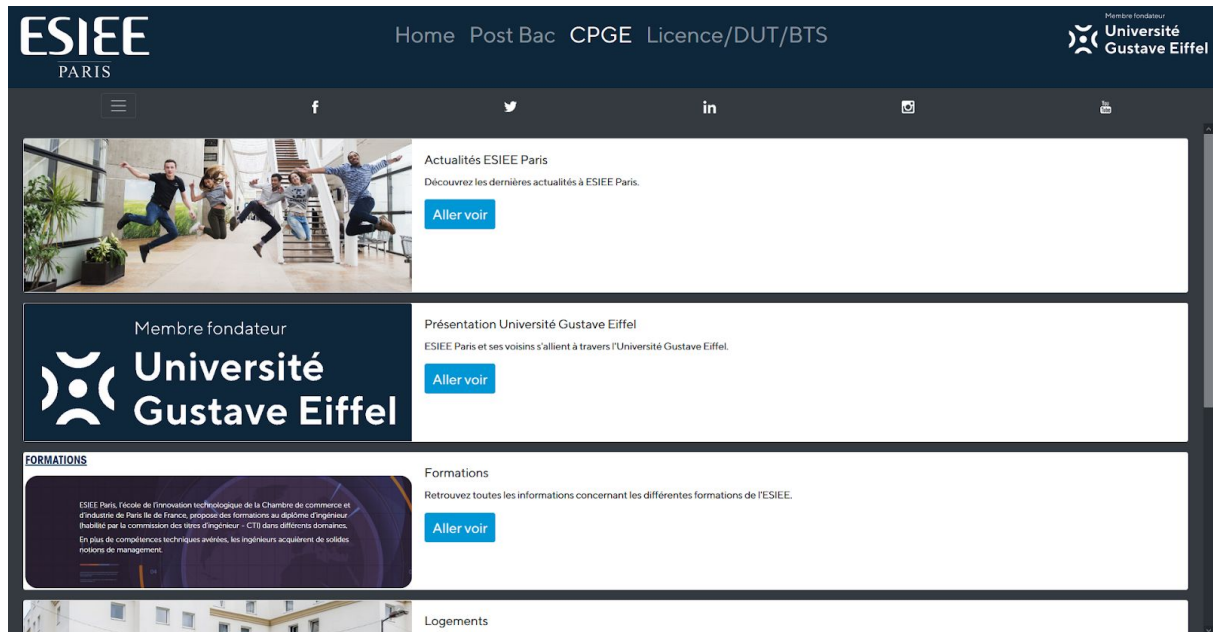
Cependant, en faisant plusieurs recherches, nous avons remarqué que certaines options n'étaient pas disponibles ou trop complexes, telles que les

dégradés. Nous avons donc gardé en partie le squelette de la maquette réalisée au début du projet et nous nous sommes adaptés pour le reste des éléments. Ainsi, plutôt que de garder la partie chatbot et la partie article côte à côte, nous avons décidé de maintenir la partie article dans un menu déroulant aussi bien sur téléphone que sur ordinateur.

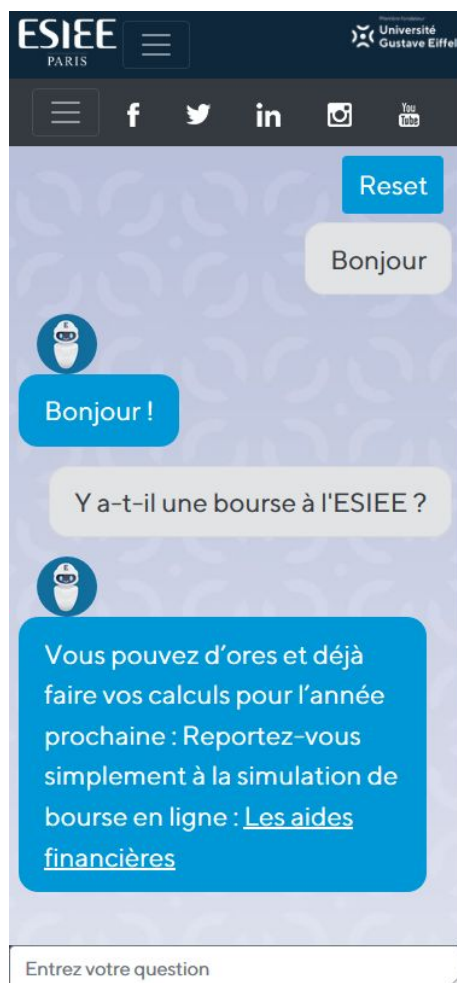
Après la création du site avec Bootstrap, nos tuteurs nous ont fait remarquer que faire prendre l'ensemble de la page au chatbot rendait la lecture plus difficile car les questions et les réponses étaient trop éloignées. Nous avons donc suivi leurs conseils et avons donné une largeur limite à la zone de discussion. Cela nous a aussi obligé à mettre un fond derrière le chatbot pour ne pas avoir de zones blanches. Ainsi, si l'écran est trop large, la zone de discussion gardera une taille lisible sans déranger le visuel du site.



Visuel final ordinateur : partie chatbot



Visuel final ordinateur : partie article



Visuel final téléphone

Maintenance du chatbot (partie web) :

Une des demandes du cahier des charges était de permettre l'amélioration et la maintenance du bot par l'administration sans avoir de connaissances en informatique.

Pour cela, nous avons ajouté un système de feedback avec le chatbot, celui-ci demandant à l'issue de chaque question si la réponse convient à l'utilisateur. Ce dernier doit alors répondre dans le chat "oui" si la réponse lui convient et "non" dans le cas contraire. Dans ce cas, le bot renvoie à l'utilisateur 3 autres balises (sujets) qu'il pense être en rapport avec la question de l'utilisateur. La réponse associée à la balise choisie sera alors affichée, et la question de l'utilisateur sera ajoutée avec la balise choisie par l'utilisateur dans les suggestions de feedback pour être validée et ajoutée par l'administration plus tard. Dans le cas où aucune des balises ne correspond, un choix "autre" est proposé, ce qui enverra la question de l'utilisateur dans le système de feedback pour qu'elle puisse être traitée par l'administration.

Afin de faciliter le travail de cette dernière, nous avons décidé d'ajouter d'autres pages à notre site web, dont l'accès serait réservé aux membres de l'administration. Cet ajout consiste en 5 pages :

- Une page "Recherche", permettant de visualiser l'intégralité du contenu d'une balise.
- Une page "Suggestions" qui répertorie l'intégralité des questions auxquelles le bot n'a pas su répondre correctement, avec la balise associée, si l'utilisateur en a choisi une. Cela aura pour conséquence d'associer la question à la balise en question, et l'administrateur aura le choix de l'ajouter ou non à la base de données. Dans le cas où la balise choisie est la balise "autre", alors c'est cette balise qui sera affichée avec la question associée. Cependant il sera impossible d'utiliser la fonctionnalité d'ajout, il faudra alors passer par les autres pages pour ajouter "à la main" cette suggestion.
- Une page "Ajout" permettant d'ajouter des balises dans le fichier .json, avec un nom de balise ; 3 modèles (questions) ; une catégorie (à sélectionner parmi celles disponibles) ; ainsi qu'une réponse type pour les questions posées dans les modèles.

- Une page "Modification", permettant de modifier le contenu des balises existantes, que ce soit les modèles (questions) ou les réponses.
- Une page "Suppression", permettant de supprimer des balises inutiles ou obsolètes (la balise COVID par exemple, une fois que l'épidémie sera terminée).

Fonctionnement de la maintenance du chatbot :

Dans un premier temps, on crée une structure de base qui sera partagée par toutes les pages dans laquelle on définira des blocs et on importera Bootstrap.

Puis on fait la routine de création d'une page en créant la route et la fonction qui lui correspond.

Ici il s'agit de pages "interactives" donc elles seront presque toutes dotées des méthodes POST et GET (pour récupérer les données des formulaires et des cookies), et nous utiliserons les paramètres de `render_template` pour passer des données à la page et ainsi y afficher des éléments du JSON.

Le but étant d'avoir des pages internet permettant de communiquer avec le JSON il nous faut alors un formulaire sur la page dont les inputs changeront selon la page.

```
<!-- Partie formulaire de la page-->
<div class="container-fluid mt-3">
  <form class="needs-validation" method='POST' novalidate>

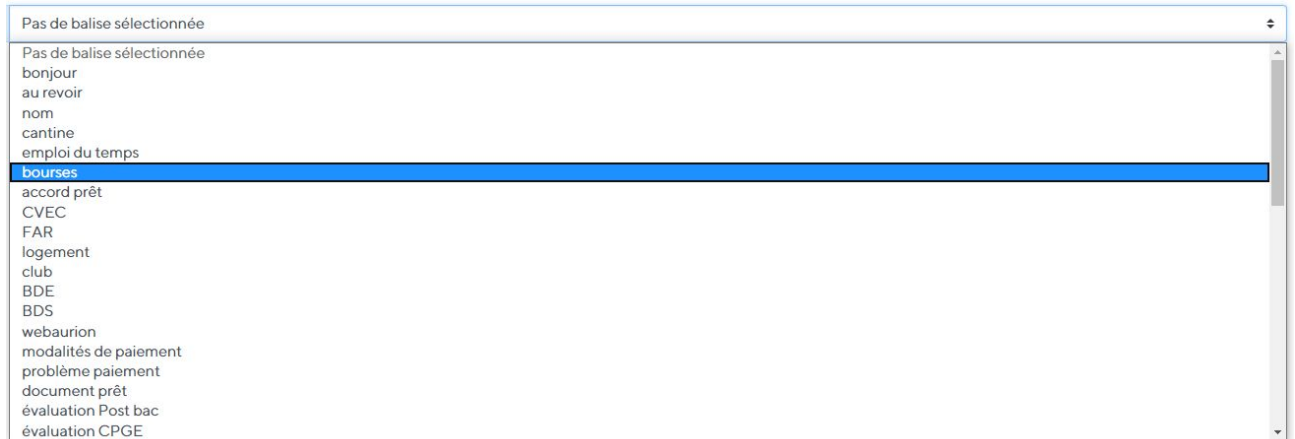
    {% block form %} {% endblock %}

    <!-- Bouton de validation qui va soumettre le formulaire -->
    <button type="submit" class="btn btn-success" name="Bouton">Valider</button>
  </form>
```

Code de la partie formulaire des pages de maintenance

On utilisera également dans les parties form beaucoup de blocs for qui nous permettront d'afficher tous les éléments des listes du JSON sur la page.

Pour sélectionner un élément parmi une liste on utilisera alors les menus déroulants et pour permettre de changer des éléments on utilisera les zones de texte (et l'on mettra en readonly si on veut juste informer l'utilisateur).



Exemple d'un menu déroulant

```
<div class="input-group mb-3">
  <div class="input-group-prepend">
    <label class="input-group-text" for="inputGroupSelect01">Veuillez sélectionner parmi la liste suivante la balise que vous souhaitez afficher :</label>
  </div>
  <select name="balise" class="custom-select" id="inputGroupSelect01" required>
    <option selected disabled value="">Pas de balise sélectionnée</option>
    {% for i in range (balises|length) %}
    <option value="{{balises[i]}}">{{balises[i]}}</option>
    {% endfor %}
  </select>
</div>
```

Code exemple d'un menu déroulant

Voici la balise proposée :	Echanges internationaux
Voici le modèle suggéré :	Comment avoir un logement

Exemple zone de texte

```
<!-- Balise proposée que l'on peut que lire -->
<div class="input-group mb-3">
  <div class="input-group-prepend">
    <span class="input-group-text" id="basic-addon1">Voici la balise proposée : </span>
  </div>
  <input type="text" name="balise" class="form-control" aria-describedby="basic-addon1" value="{{balise}}" required readonly>
</div>

<!-- Zone de texte que l'on peut modifiée dans laquelle on a de base le modele associé à la balise -->
<div class="input-group mb-3">
  <div class="input-group-prepend">
    <span class="input-group-text" id="basic-addon2">Voici le modèle suggéré : </span>
  </div>
  <input type="text" name="modele" class="form-control" aria-describedby="basic-addon2" value="{{modele}}" required>
</div>
```

Code exemple zone de texte

Il sera également nécessaire d'avoir une navbar qui nous permettra de naviguer facilement parmi les différentes pages.

Recherche Suggestions Ajout Modification Suppression

Navbar des pages de maintenance

```
<!-- Navbar contenant les liens vers les différents onglets de mise a jour du json-->
<div class="collapse navbar-collapse justify-content-around" id="navbarNav">
  <div class="navbar-nav">
    <!-- Les liens, qui font partie de la navbar, vers les différents onglets de mise a jour du json -->
    {% block navbar %} {% endblock %}
  </div>
</div>
```

Code de la navbar des pages de maintenance

Pour modifier le JSON avec les données récupérées du formulaire, on utilise alors les méthodes présentées dans la partie IA comme : ajout_dans_json, suppr_dans_json ou même extraire_suggestion.

Deux pages se démarquent au niveau de leur construction dans le code utilisant Flask, à savoir la page de suggestion et la page de modification.

En effet, pour la page de suggestion, deux routes existent et l'une d'entre elles utilise un redirect. La route qui n'utilise pas les méthodes GET et POST ne sert qu'à afficher la page en envoyant les informations dont elle a besoin pour s'afficher. La deuxième route récupère les données du formulaire et, selon la valeur de la radio, ajoute ou supprime la suggestion. Puis, la deuxième route redirige vers la première. Cette redirection permet de pallier la latence de la suppression dans le fichier texte (et donc afficher les bonnes données après chaque "exécution").

```
@app.route('/suggestions') # Emplacement de la page de modification de le site de maj du json
def prop():
    balise, modele, nb_sug = exsug('amelioration.txt') # Récupere les suggestions stockées dans le txt
    return render_template('suggestions.html', balise=balise, modele=modele, nb_sug=nb_sug) # affiche la page de suggestion en lui envoyant les données utiles
```

Code “d’affichage” de la page suggestions

```
@app.route('/suggestions', methods=['POST', 'GET']) # Emplacement de la page de modification de le site de maj du json (avec les méthodes de recup de données)
def prop_post():
    balise_form = request.form['balise'] # Recupere l'info du champs balise
    modele_form = request.form['modele'] # Recupere l'info du champs modele
    application = request.form['application'] # Recupere l'info du champs application

    if (application == "true"):
        ajsug(fichier_donnees, "amelioration.txt", balise_form, modele_form) # Ajoute la suggestion (en prennant en compte les modifs) au json
    else:
        resug("amelioration.txt") # Supprimer la suggestion du txt

    return redirect('/suggestions') # Redirige vers la page de suggestion
```

Code “d’exécution” de la page suggestions

La page de modification, elle, utilise les cookies et deux pages HTML différentes.

Puisqu’on utilise les cookies, on commence par gérer leur initialisation en activant un booléen de modification s’ils n’existent pas ou si les cookies de modifications sont vides.

```
if((not request.cookies.get('session'))): # Si il n'existe pas de cookies de session
    initCookie(modif=True) # On initialise les cookies avec le booleen de modification à True
elif(session['modif'] == ""):
    initCookie(modif=True) # On initialise les cookies avec le booleen de modification à True
```

Code d’initialisation des cookies de modification

Ensuite on va différencier les deux cas possibles (correspondant à chaque page HTML). Si les cookies de modification sont à “on”, on passe alors leur valeur à “off” (pour alterner entre les deux pages) puis on récupère toutes les informations du formulaire de la page que l’on stocke dans les cookies pour les utiliser dans la deuxième page HTML. On finit alors par passer à la deuxième page en l’affichant (et en lui donnant les paramètres dont elle a besoin).

```
if(session['modif'] == "On"):  
  
    session['modif'] = "Off" # On donne la valeur Off au cookies de modification  
    balise = request.form['balise'] # Recupere l'info du champs balise  
    objet = request.form['objet'] # Recupere l'info du champs objet  
    obj = odj(fichier_donnees, balise, objet) # On recupere tous les éléments de l'objet choisit  
    session['balise'] = balise  
    session['objet'] = objet  
    session['obj'] = obj  
  
    return render_template('modif2.html', objets=obj, page=obj) # affiche la deuxième page de modification en lui envoyant les données utiles
```

Code de la première page de modification

Sinon on récupère les informations des cookies et les modifications faites sur le formulaire (plus le champs supplémentaire s'il s'agit des modèles) pour ensuite remplacer les éléments dans le JSON avec la fonction `remplace_objet`.

Finalement on passe la valeur du cookie de modification à "on" (pour repasser à la première page) et on affiche la première page en lui envoyant les informations nécessaires.

```
else:  
  
    liste_modif = []  
    balise = session['balise']  
    objet = session['objet']  
    obj = session['obj']  
    for i in range(len(obj)):  
        modif = 'modif' + str(i)  
        liste_modif.append(request.form[modif]) # Recupere l'info du champs modif[i] et l'ajoute à la liste liste_modif  
    if(objet == "modeles"):  
        liste_modif.append(request.form['plus']) # Recupere l'info du champs plus (pour les modeles) et l'ajoute à la liste liste_modif  
  
    ro(fichier_donnees, balise, objet, liste_modif) # Remplace les éléments dans le json  
    session['modif'] = "On" # On donne la valeur On au cookies de modification  
  
    return render_template('modif.html', balises=balises) # affiche la page de modification en lui envoyant les données utiles
```

Code de la deuxième page de modification

Gestion du projet

Dans cette partie, nous allons vous décrire les outils que nous avons utilisés afin de mener à bien ce projet, à savoir Jira d'Atlassian et Git.

Git :

Nous avons décidé d'utiliser l'outil Git pour le développement du projet. Il nous fallait un moyen simple de partager notre avancement tout en conservant une trace des anciennes versions des fichiers du projet, et l'un des membres du projet était déjà familier avec l'outil, ainsi le choix s'est imposé de lui-même.

Afin d'éviter un quelconque problème de confidentialité et "d'espionnage" du projet par d'autres écoles, nous avons opté pour une version privée de Git en utilisant le serveur GitLab hébergé par le SMIG.

Pour simplifier l'utilisation de Git, nous avons installé le programme "ungit" fonctionnant avec node.js.

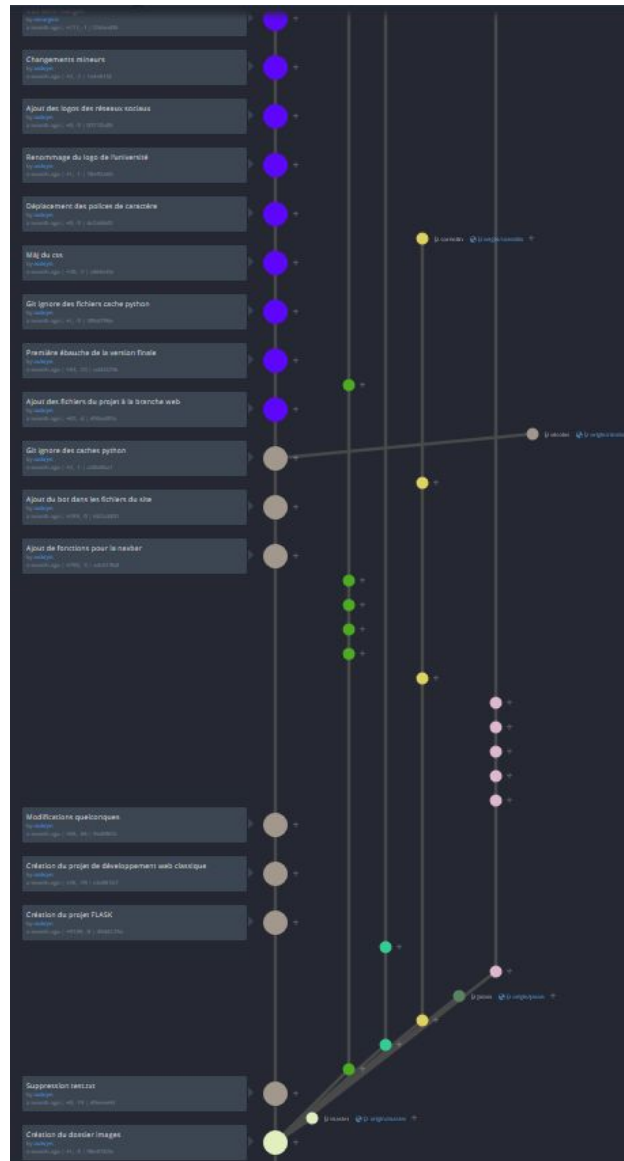
Ce programme permet d'avoir une vue graphique des branches du projet, de réaliser les commits plus facilement, et rend l'utilisation de Git très accessible (aucune commande n'est écrite).

Rappel de ce qu'est un commit : un commit est une modification du projet (des fichiers le constituant en général) que l'on envoie en lui donnant un titre et en le décrivant (optionnel). Les différents développeurs du projet récupèrent ensuite ce commit. Chaque commit laisse une trace dans l'historique du projet et il est possible d'en consulter le contenu à tout moment du projet afin de voir les modifications apportées. Il est aussi possible de revenir quelques commits en arrière.

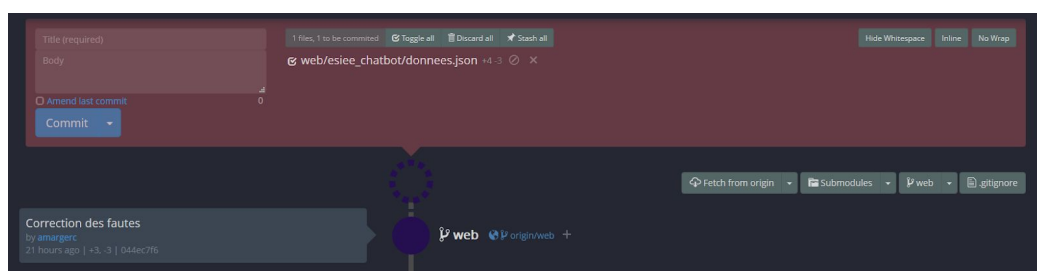
Git propose un système de branches pour travailler en parallèle sur plusieurs fonctionnalités sans impacter les autres développements. On peut passer

d'une branche à une autre facilement et on peut aussi fusionner des branches quand les fonctionnalités sont prêtes, par exemple.

Sur ungit, les commits sont représentés par des cercles sur les branches, voici deux exemples de l'interface d'ungit.



Arbre partiel du projet



Interface de commit

Jira :

Suite à la proposition de M. BUREAU, nous avons décidé de présenter notre groupe à la formation aux méthodes agiles. Nous avons été choisis pour être formés sur l'outil Jira par Samuel HABIF.

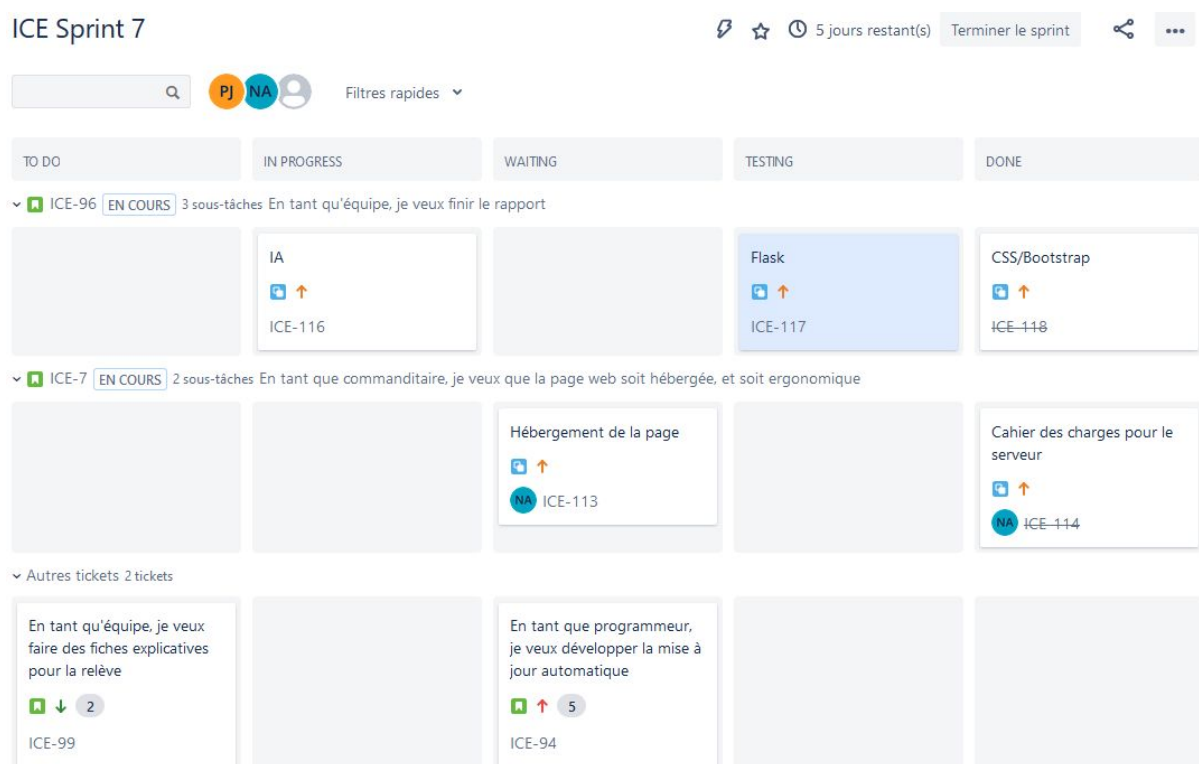
L'outil en question est un site internet sur lequel les différentes personnes du projet organisent leurs semaines de travail. Notre apprentissage a été réalisé sur 3 grandes parties du site :

Backlog : Cette page est celle sur laquelle les membres du projet créent des "tickets", représentant les tâches à réaliser pour mener à bien le projet. Ceux-ci ont plusieurs paramètres : l'ordre de priorité pour savoir quel ticket est le plus important, les story-points qui évaluent la difficulté de la tâche et le responsable (celui qui va réaliser le ticket). Il est aussi possible d'écrire une description pour expliquer plus en détail le travail demandé ainsi que de donner une étiquette au ticket pour le rattacher à une catégorie. C'est aussi sur cette page que le groupe va programmer ses "Sprints". Un sprint est un ensemble de tickets que l'on décide de réaliser sur une période définie. L'objectif est de réaliser les tickets au fur et à mesure de la période et d'arriver à les finir pour la date limite prévue.

The screenshot shows the Jira Backlog interface. At the top, there's a search bar and a 'Filtres rapides' dropdown. Below that, a section for 'ICE Sprint 7' with 4 tickets is visible. It includes a 'Démarrer le sprint' button and a 'Planifier un sprint' dropdown. The tickets are listed in a table with columns for description, ID, priority, and story points. Below the table, there's a '+ Créer un ticket' button. At the bottom, there's a section for 'Backlog' with 0 tickets and a 'Créer un sprint' button. A message states 'Votre backlog ne contient aucun élément.'

VERSIONS	EPICS	Description	ID	Priority	Story Points
		En tant que programmeur, je veux développer la mise à jour automatique	ICE-94	↑	5
		En tant qu'équipe, je veux finir le rapport	ICE-96	↑	7
		En tant qu'équipe, je veux faire des fiches explicatives pour la relève	ICE-99	↓	2
		En tant que commanditaire, je veux que la page web soit hébergée, et soit ergonomique	ICE-7	↓	5

Sprint actif : Sur cette page, on peut observer l'avancement du sprint ou des sprints actifs. Dans notre disposition, cette page sépare les tickets dans 5 colonnes différentes : "To do" pour les tickets qui n'ont pas encore été commencés, "In progress" pour ceux qui sont en cours, "Waiting" pour ceux qui sont en attente, "Testing" pour ceux qui sont en essais et "Done" qui possède deux catégories pour soit valider soit annuler le ticket. Afin de valider un ticket, il est d'abord nécessaire qu'il passe par la colonne "In progress" puis "Testing" sans quoi il sera annulé. Un ticket présent dans la partie "Waiting" est obligé de repasser dans la partie "In progress" avant d'être à nouveau déplacé.

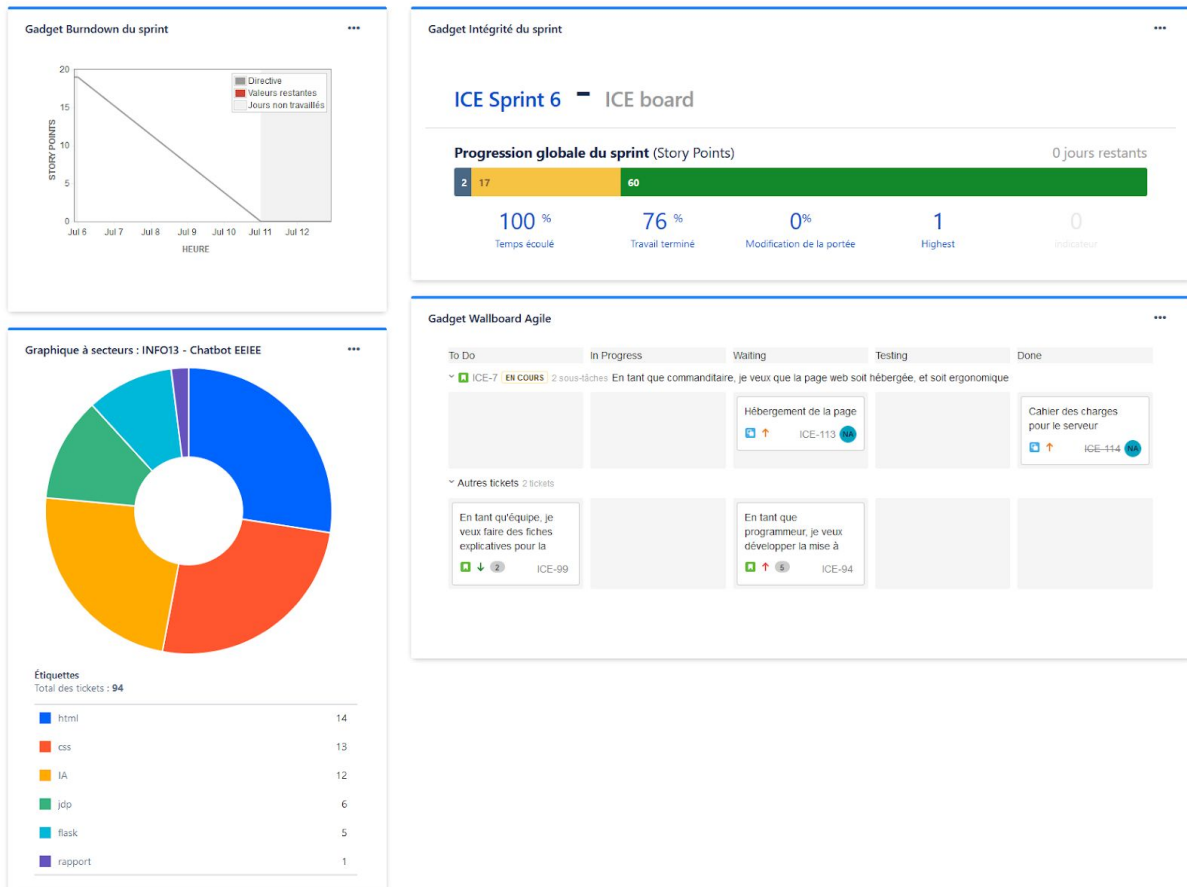


Rapports : Cette dernière page permet de voir l'évolution du projet sous forme de graphiques. Ceux-ci vont permettre d'interpréter l'avancement de validation des tickets au cours du projet ou au cours des sprints. Le graphique que nous avons principalement utilisé lors de nos réunions est le graphique de Burndown. Il affiche la quantité de story-points restants au cours du temps. Pour notre 4e sprint s'étalant du 9 juin au 16 juin, nous avons obtenu le graphique suivant :



Le tracé gris représente la courbe “directive” qui est donc celle que le groupe est censé suivre. Elle permet de vérifier si l’avancement est le bon. Si la courbe rouge est au dessus de la grise, la quantité de travail est peut être trop importante pour la durée donnée. Si à l’inverse elle est en dessous, la quantité de story points est potentiellement sur-estimée. Cela permet donc aux membres du projet de visualiser la quantité de travail réalisable en une certaine durée.

Pour finir, il est aussi possible de créer un tableau de bord regroupant de nombreux graphiques pour avoir une vue d’ensemble des sprints. Lors de notre projet, nous avons créé et utilisé le Tableau suivant, regroupant l’affichage du sprint actif, le graphique de burndown, les domaines des tickets en fonction de leur étiquette et un graphique d’intégrité :



Cette formation ainsi que ses outils nous ont été très pratiques. Ils nous ont permis de mieux nous organiser dans notre groupe et notre travail, ainsi que d'avoir un aperçu plus général de l'avancement du projet. De plus, la mise en place d'une réunion hebdomadaire pour faire un bilan sur le travail effectué et l'apprentissage de Jira permettait de nous remotiver pour le travail de la semaine.

Problèmes rencontrés

Le plus gros problème que nous avons rencontré a été de gérer les entrées, qui se font en python avec input mais qui ne fonctionnent que sur une invite de commande.

Or sur Flask, le principe du bot était d'avoir une text-box qu'on valide, le texte est ensuite traité par une fonction python en utilisant les modèle pré-entraîné, la réponse associée à ce traitement est ensuite ajoutée dans le cookie (ce cookie permet au moment du rafraîchissement de la page d'afficher l'historique du chat et donc la réponse de la question qui vient d'être posée). Ce principe de bot étant pseudo-dynamique voulait dire que si on voulait implémenter un système de notation il fallait que le bot fasse la différence entre une question posée et une réponse de note (oui ou non en fonction de si la réponse lui va).

Le passage d'un CSS codé "à la main" au framework Bootstrap a été compliqué. En effet, Pierre JANNOT et Corentin AMARGER ont alors dû apprendre très rapidement l'utilisation de Bootstrap afin de ne pas pénaliser le reste de l'équipe. Il a alors fallu reprendre le HTML du site ainsi que son habillage presque entièrement de zéro. Nous avons également pris la décision de s'écarter de la maquette établie puisque celle-ci ne prenait pas en compte l'utilisation de Bootstrap.

L'un de nos derniers problèmes a été l'implémentation du choix d'activation du feedback. Notre principale contrainte était le temps car il s'agissait d'une fonctionnalité que nous codions pendant les derniers jours du projet. Ensuite l'autre problème venait de la compatibilité avec le système de feedback lui-même car le fait de pouvoir l'activer ou le désactiver à tout moment amenait à de nombreux bugs. On peut rajouter aussi le problème de trouver et comprendre un code javascript permettant de garder l'état de la checkbox (sans connaissances de javascript), en très peu de temps de travail.

Finalement nous nous sommes heurtés à un problème auquel nous ne nous attendions absolument pas lors du déploiement du projet sur un serveur : un import de librairie bloquait. En effet, Python permet de développer un programme sans prendre en compte le système d'exploitation sur lequel celui-ci sera en marche. Ainsi il nous a été impossible de déployer le projet sur le serveur. Cela dit, lorsque le problème de la librairie Tensorflow ne voulant pas s'importer sera résolu, le déploiement ne devrait pas poser de problèmes.

Vous trouverez en Annexe un tutoriel vidéo expliquant la procédure de déploiement.

Améliorations possibles

Une des améliorations que nous n'avons pas eu le temps d'implémenter mais que nous avons imaginée était un système de mot de passe, pour la partie modification, pour empêcher que des élèves y accèdent en trouvant l'url. L'autre amélioration possible qui a été imaginée était la fonctionnalité existant dans IBM, le search.

Cela consiste à créer un nouveau bot, qui puisse chercher sur des pages web, et extraire de ces dites pages les informations correspondantes pour répondre à la question. Nous avons commencé à regarder comment faire, mais après avoir extrait les données des différents sites web en utilisant un scraper, Scrapy étant une librairie sur Python qui permet de récupérer les éléments définis. Mais après cela nous avons vite compris qu'avec le temps que nous prenait le bot, nous n'allions pas avoir le temps de développer cette fonction, nous avons donc préféré nous concentrer sur notre bot et l'amélioration de reconnaissance de questions plutôt que de créer de nouvelles fonctionnalités non vitales.

Il serait important d'ajouter dans la page de suggestions le fait d'afficher deux blocs de texte en plus, si la balise est autre, dans lesquels on rentrerait la balise à associer à la question et la réponse correspondante, pour ajouter tout cela dans le fichier JSON lorsque l'utilisateur de la page décide d'ajouter la suggestion au JSON.

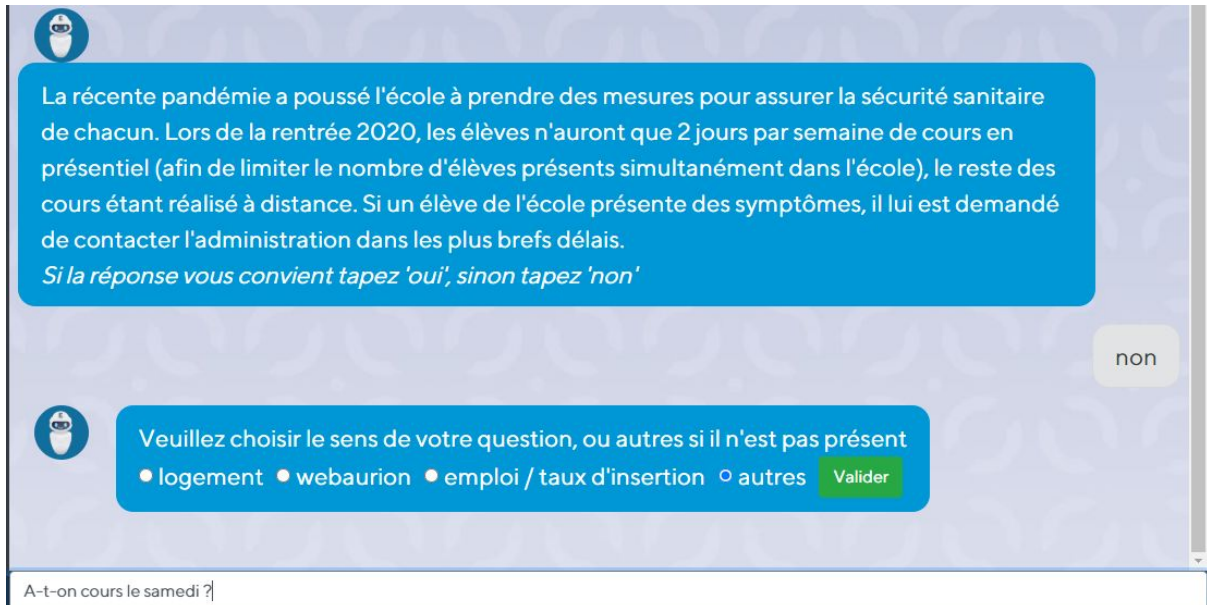
Une amélioration plutôt importante selon nous est de permettre à l'utilisateur de choisir s'il veut activer le système de feedback ou non. Nous avons déjà commencé à implémenter cette fonctionnalité. Cependant comme expliqué précédemment, le manque de temps a impliqué un "grand" nombre de bugs et le manque de certaines parties de cette fonctionnalité. Ainsi des parties commentées du code correspondent à cette fonctionnalité (checkbox d'activation et interaction avec le système de feedback). Il manquerait alors de

placer la checkbox, le fait de garder la checkbox dans son état après rafraichissement de la page et de corriger les bugs signalés ci-dessous.

L'un des problèmes actuels de notre chatbot est que si la personne ne trouve pas de balise correspondante dans le système de feedback et qu'il renvoie la balise autre en feedback, il n'aura pas de réponse à sa question. Ainsi une idée d'amélioration serait de fournir un contact de l'administration lorsque la réponse au feedback est autre. Il peut même être demandé à l'utilisateur de sélectionner le département de l'administration auquel se rattache leur question pour ainsi fournir le contact correspondant, ou mettre un lien vers la page "Contacter ESIEE Paris" <https://www.esiee.fr/fr/contact>.

Pour finir, il existe une interface qui permet de modifier le fichier de données pour quelqu'un qui ne s'y connaît pas en informatique mais cela n'est pas disponible pour les articles. En effet, pour modifier les articles il faut modifier le HTML directement. Ainsi il serait intéressant de créer une interface permettant de modifier facilement les articles, voire même en donner un aperçu avant validation dans cette interface.

Bugs connus



La récente pandémie a poussé l'école à prendre des mesures pour assurer la sécurité sanitaire de chacun. Lors de la rentrée 2020, les élèves n'auront que 2 jours par semaine de cours en présentiel (afin de limiter le nombre d'élèves présents simultanément dans l'école), le reste des cours étant réalisé à distance. Si un élève de l'école présente des symptômes, il lui est demandé de contacter l'administration dans les plus brefs délais.

Si la réponse vous convient tapez 'oui', sinon tapez 'non'

non

Veuillez choisir le sens de votre question, ou autres si il n'est pas présent

• logement • webaurion • emploi / taux d'insertion ○ autres Valider

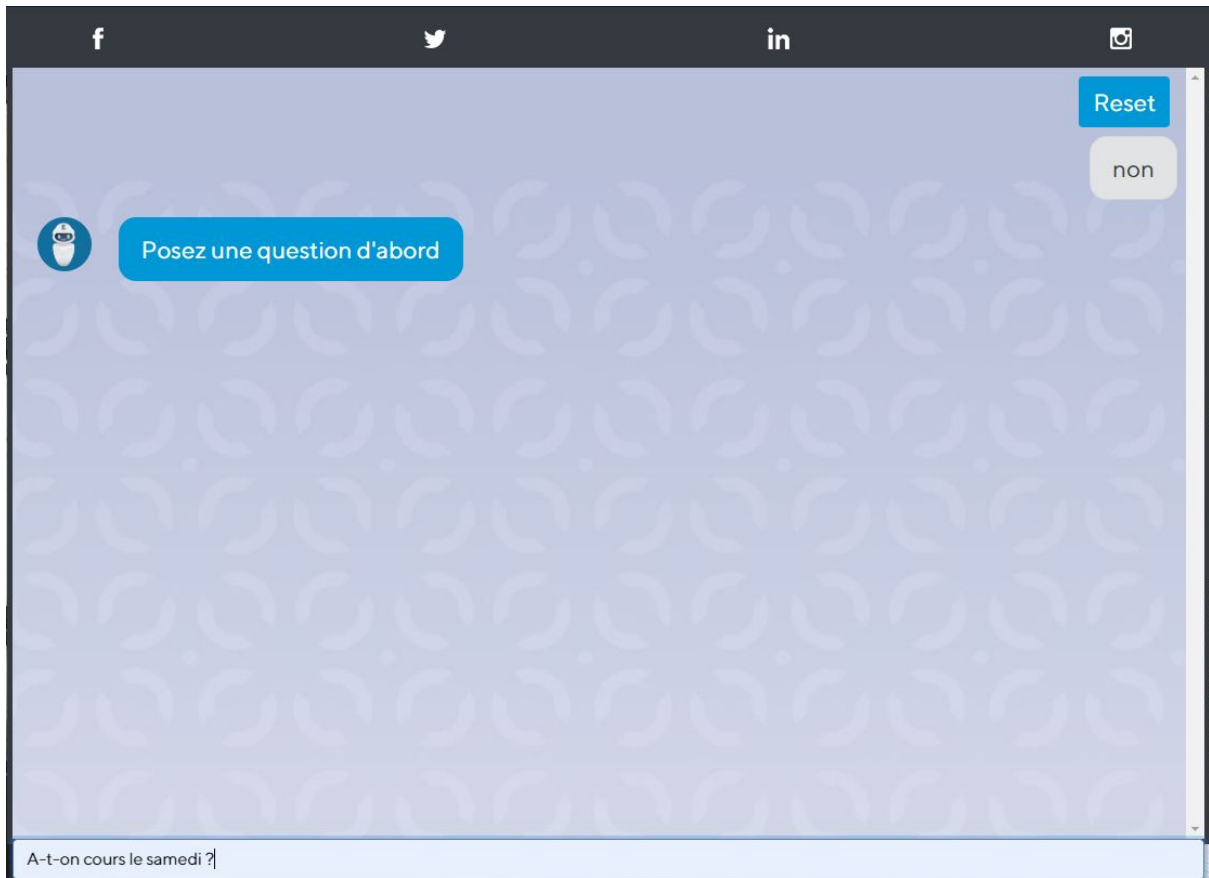
A-t-on cours le samedi ?

Quand on envoie le message de feedback qui permet de choisir la balise correspondant à la question posée, si l'utilisateur rentre une question dans la zone de texte sans répondre au feedback avec le bouton valider, on obtient l'erreur ci-dessous :

`werkzeug.exceptions.BadRequestKeyError`

```
werkzeug.exceptions.BadRequestKeyError: 400 Bad Request: The browser (or proxy) sent a request that this server could not understand.  
KeyError: 'amelio'
```

En effet, lorsque la page est en mode feedback, Flask cherche à récupérer le contenu du formulaire de feedback mais celui-ci ne soumettra rien dans le cas donné au dessus.

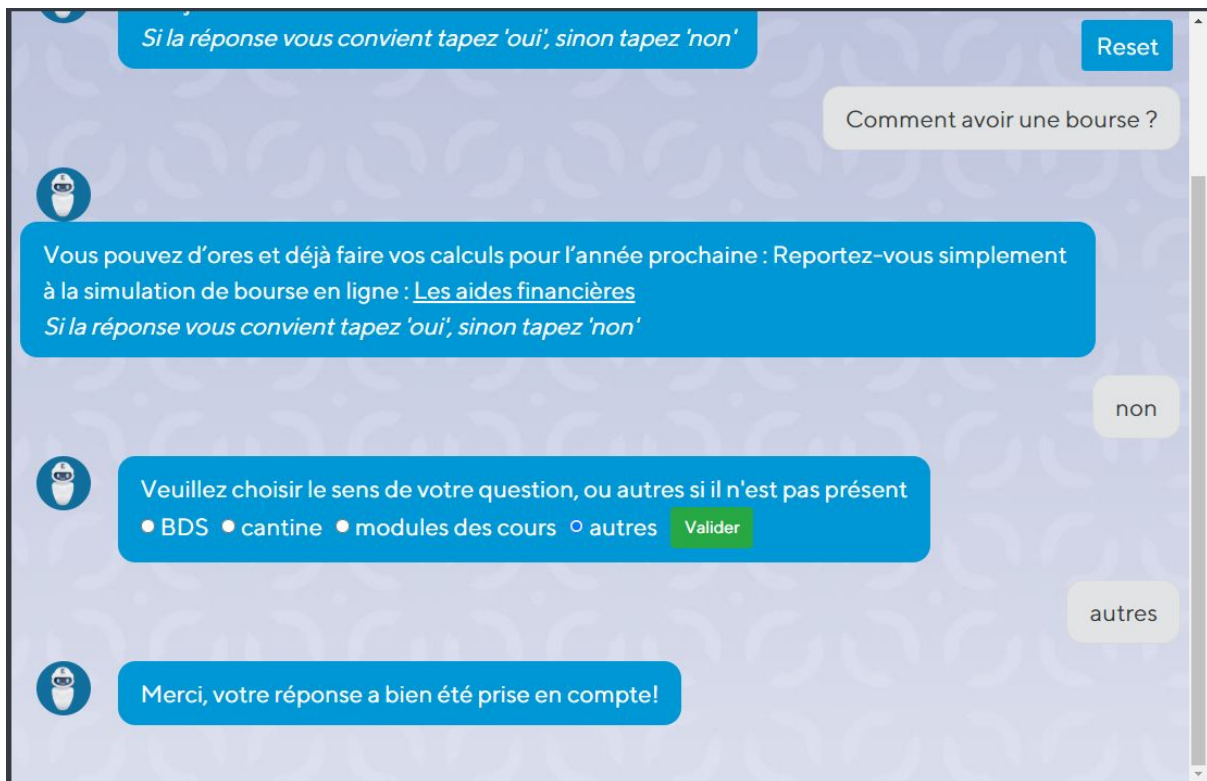


Quand le premier message envoyé est “non”, quelque soit le contenu du message, il y aura une erreur à la prochaine requête :

`werkzeug.exceptions.BadRequestKeyError`

```
werkzeug.exceptions.BadRequestKeyError: 400 Bad Request: The browser (or proxy) sent a request that this server could not understand.  
KeyError: 'amelio'
```

Celle-ci provient de comment est récupérée la question pour l’ajouter dans le fichier texte d’amélioration. En effet, on récupère alors l’élément -2 de l’historique de chat et dans ce cas, il n’existe pas.

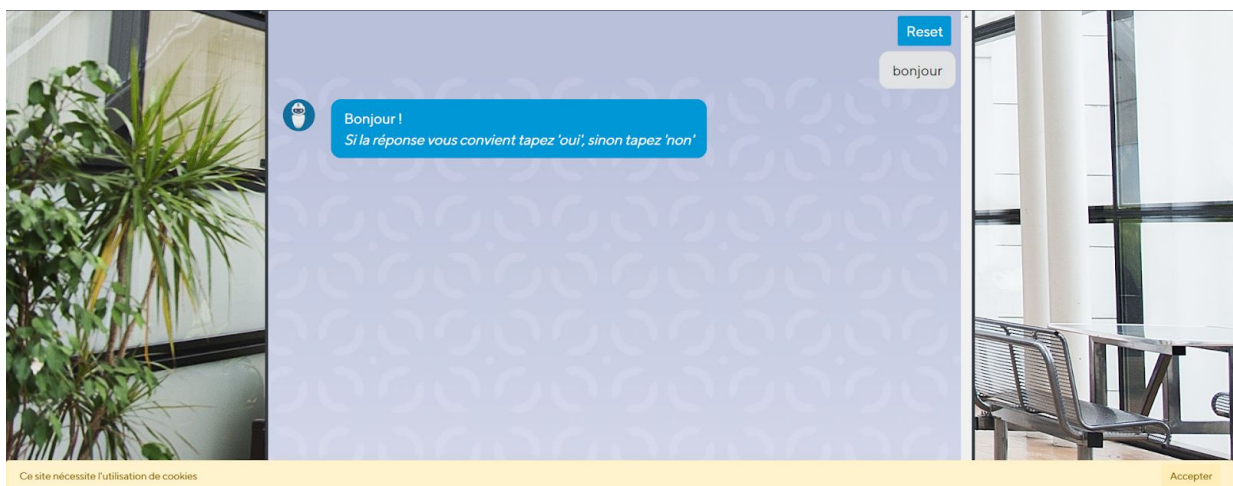


Quand on appuie sur un bouton valider d'un formulaire de feedback sans avoir envoyé un message "non" avant, on obtient l'erreur suivante :

werkzeug.exceptions.BadRequestKeyError

`werkzeug.exceptions.BadRequestKeyError: 400 Bad Request: The browser (or proxy) sent a request that this server could not understand. KeyError: 'chatzone_g'`

Cette erreur provient du fait que sans envoyer un message "non", Flask ne détecte pas que le bot est en mode feedback.



Il faut accepter deux fois l'alerte des cookies car celle-ci s'affiche de nouveau après avoir envoyé le premier message mais ne s'affiche plus par la suite.

Conclusion

Pour conclure et récapituler quant à l'état final du projet, nous avons réussi à implémenter un chatbot capable de répondre à un large panel de questions sur l'admission à l'ESIEE, et ce sur une page web respectant la charte graphique de l'école, cette dernière étant, d'après nos retours, intuitive à l'utilisation. Nous avons de plus mis à disposition de l'administration des outils permettant de modifier et d'améliorer le bot selon leurs besoins, facilement et sans avoir à modifier le fichier à la main.

Qui plus est, même s'il advenait que le bot ne soit pas capable de répondre correctement à une question d'un étudiant, l'option de pouvoir ajouter ladite question facilement à la base de donnée afin qu'elle soit reconnue la fois d'après est disponible, permettant ainsi, à terme, d'avoir un chatbot qui pourra répondre correctement à toutes les questions possibles.

Des pistes d'amélioration que nous n'avons pas eu le temps d'implémenter, faute de temps, sont aussi présentées dans ce rapport, tout comme les bugs présents suite aux derniers ajouts au projet, mais le chatbot est malgré tout fonctionnel dans sa globalité.

Remerciements

Nous remercions monsieur Samuel HABIF pour nous avoir formé aux méthodes agiles tout au long du projet.

Nous remercions également madame Corinne BERLAND et monsieur Denis BUREAU pour leurs conseils.

Nous tenions aussi à remercier monsieur Ange MICAELLI pour le temps qu'il nous a consacré lors de la mise en production du site.

Nous remercions monsieur Jean-François BERCHER pour les informations qu'il nous a fournies, mesdames Murielle ROUGIER et Isabelle CARO pour leur aide concernant les questions récurrentes et madame Christine CEVAER pour ses précisions concernant les livrables du Jour Des Projets.

Nous remercions l'ensemble des étudiants d'ESIEE Paris qui ont répondu à notre Google Form visant à collecter un maximum de données pour le robot.

Nous remercions également monsieur Mathias AUBRY et madame Alicia LAROCHE pour leurs participations à la vidéo de présentation du projet.

Enfin nous remercions nos familles pour leur soutien tout au long du projet.

ANNEXES

Librairies utilisées avec leur version (pour python 3.7.6):

- Tflern 0.3.2
- tensorflow 1.14 (problème avec la version 2.0 où une des tensorflow.contrib a été enlevé et tflearn pas mis à jour donc ne fonctionne pas)
- numpy 1.18.4
- NLTK 3.5
- Flask 1.1.1
- pickle 0.7.5
- json 0.9.1
- Flask-Bootstrap4 4.0.2

Nous avons effectué nos développements sous Windows 10 avec les IDE Visual Studio Code et Spyder.

Contenu du dossier de projet :

Dans le dossier static vous trouverez le CSS, les images et les polices de caractères utilisées pour le site.

Dans le dossier templates vous trouverez les fichiers HTML du site.

Les dossiers static et templates doivent absolument s'appeler comme ça et être à la racine du projet comme le fichier contenant le script Flask.

Le fichier website.py contient le script Flask et est le script à lancer pour allumer le site.

Les fichiers entraînement.py, fonction.py et chat_utilisation_web.py sont les fichiers python liés à l'IA. Le fichier entraînement.py sert à entraîner le bot, il devra être lancé idéalement tous les soirs vers 01h00 du matin afin d'améliorer le robot. Vous trouverez dans fonction.py les fonctions liées à la maintenance du fichier donnees.json. Vous trouverez dans chat_utilisation_web.py les fonctions nécessaires au fonctionnement du chatbot.

Le fichier donnees.json est le fichier contenant le dictionnaire utilisé par l'IA

pour répondre.

Les différents fichiers au format .pickle sont des fichiers de sauvegarde de données, ils sont importants, ne pas les supprimer.

Les différents fichiers contenant .tflearn sont les fichiers liés à l'IA et à ses modèles. Ils sont importants, ne pas les supprimer.

Le fichier amelioration.txt est le fichier contenant les suggestions des utilisateurs.

Code de la fonction page()

```
def page(champ_q, champ_r, chatzone, botname, feedback="False"):

    # Stockage de l'historique de chat
    if (request.cookies.get('session') == None):

        # Initialisation du cookie de session

        initCookie()

        # Stockage de l'historique de chat sous forme de string

        questions = session[champ_q] # Récupération des données du cookie
        reponses = session[champ_r]

        questions += request.form[chatzone] # On ajoute la première question
        # On ajoute la première réponse
        reponses += bot((session[champ_q] + "|" +
            request.form[chatzone]).split("|"), botname, feedback)
        session[champ_q] = questions # On stocke dans le cookie de session
        session[champ_r] = reponses # On stocke dans le cookie de session

    else:

        # Stockage de l'historique de chat sous forme de string
        # On récupère le contenu des questions posées et on le stocke dans une variable
        questions = session[champ_q]
        # On récupère le contenu des réponses données et on le stocke dans une variable
        reponses = session[champ_r]

        if((((session[champ_q]).split("|"))[-1] == "non") & (feedback == "True")):
            #ancien_chatzone = chatzone
            chatzone = "amelio"

        if(chatzone == "amelio"):
            sto("amelioration.txt",
                request.form[chatzone], (session[champ_q].split("|"))[-2])
```

```
if((questions == "") & (reponses == "")):
    reponses = bot(
        (session[champ_q] + "|" + request.form[chatzone]).split("|"), botname, feedback)
    questions = request.form[chatzone]

else:
    # On ajoute les nouvelles questions à la variable
    questions += "|" + request.form[chatzone]
    # On ajoute les nouvelles réponses à la variable
    # True c'est pour le cookie amélioration
    reponses += "|" + \
        bot((session[champ_q] + "|" + request.form[chatzone])
            ).split("|"), botname, feedback)

# On stocke la nouvelle variable de questions dans le cookie
session[champ_q] = questions
# On stocke la nouvelle variable de réponses dans le cookie
session[champ_r] = reponses

# On récupère le champ du cookie de questions correspondant à la page, qu'on split
quest = session[champ_q].split("|")
# On récupère le champ du cookie de réponses correspondant à la page, qu'on split
rep = session[champ_r].split("|")

return quest, rep
```

Lorsque le système de feedback est actif, et que le message attendu est le form envoyé par le bot, le nom de la “chatzone” change afin de récupérer les informations du form de feedback. Cette astuce nous permet d’implémenter ledit système de retour des utilisateurs, seulement elle est aussi à l’origine d’un bug qui survient lorsque l’utilisateur remplit la “chatzone” classique sans avoir validé la chatzone nommée “amelio” (pour amélioration). Flask attend en effet des informations provenant du form “amelio” et non du form “chatzone” (dépendant de chaque onglet).

Tutoriel vidéo pour le déploiement du projet sur un serveur :

<https://www.youtube.com/watch?v=goToXTC96Co>

BIBLIOGRAPHIE:

-Bird, Steven, Edward Loper and Ewan Klein (2009), *Natural Language Processing with Python*. O'Reilly Media Inc.

-<https://www.youtube.com/c/TechWithTim/featured> (chaîne YouTube que l'on a regardé lors de la première semaine, qui nous a permis de comprendre le fonctionnement d'un bot utilisant un fichier json)

-<https://www.youtube.com/user/grafikarttv> (chaîne YouTube que l'on a regardé pour le CSS, PHP, HTML, Git)

-<https://www.youtube.com/user/schafer5> (chaîne YouTube que l'on a regardé pour les tutoriels Flask)