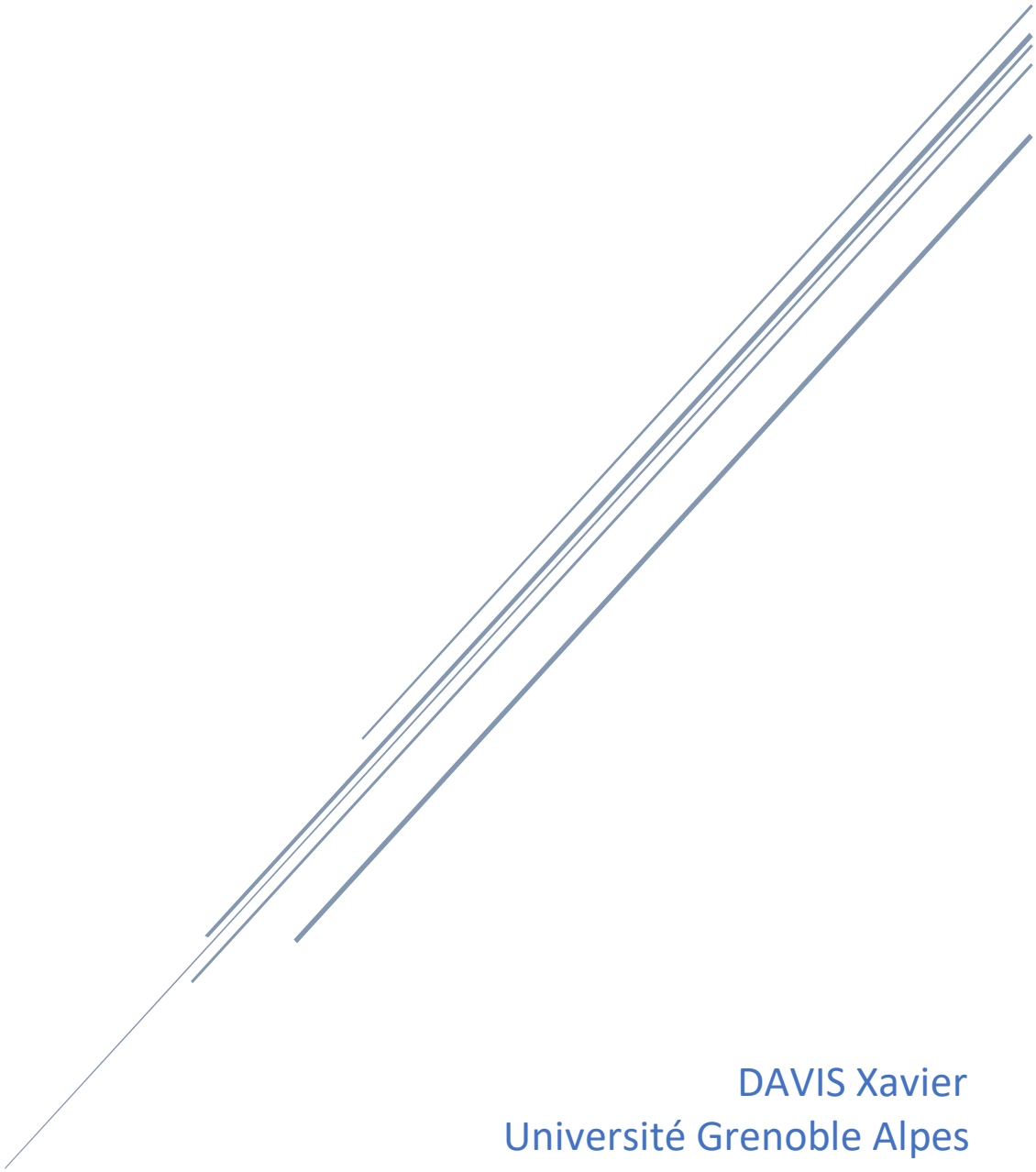
RAPPORT DE STAGE – MI-PARCOURS

Développement de formulaires et intégration en base de données



DAVIS Xavier
Université Grenoble Alpes
Master 2 Double Compétence Informatique et Sciences Sociales

Table des matières

1. Contexte du stage

1.1 Le Parc National des Écrins

Situé au cœur des Alpes, à cheval entre le département de l'Isère et celui des Hautes-Alpes, le Parc National des Écrins est une zone de conservation de faune et de flore, qui fête en 2023 ses 50 ans. C'est un des 11 parcs nationaux français, avec ceux de la Vanoise, de Port-Cros, des Pyrénées, des Cévennes, du Mercantour, de la Guadeloupe, de la Réunion, de la Guyane, des Calanques et le parc national de forêts. Le parc possède des sommets culminants jusqu'au 4102 mètres de la Barre des Écrins.

Le parc se divise en multiples sites, avec comme base les Maisons du parc, responsables d'une ou deux zones différentes. présentes dans les zones Il y en a pour le Champsaur-Valgaudemar, pour l'Embrunais, pour la Vallouise-Briançonnais, et pour l'Oisans-Valbonnais. De ces bases travaillent un.e chef.fe de secteur, des assistant.es, des agent.es d'accueil, des technicien.nes et des gardes-moniteur.trices

Le siège du parc se situe au château de Charance, dans le parc de Charance au-dessus de la ville de Gap. C'est là que sont situés la direction, le secrétariat général, le service aménagement, le service d'accueil et de communication, et le service scientifique et son pôle Service Informatique.

1.2 Le pôle SI et ses membres

1.2.1 Le pôle SI

1.2.2 Camille Monchicourt (chef de service et géomaticien)

Chef du service Informatique, Camille a effectué un stage au parc avant de l'intégrer officiellement en CDI derrière. Il est géomaticien, et non développeur, mais passe la plupart de son temps à gérer le service.

1.2.3 Théo Lechémia (développeur Geonature, maître de stage)

Théo est un ancien étudiant de DCISS, qui a effectué un stage au parc dans le cadre de son master. Il est un développeur, et est une des personnes responsables de l'utilisation de Python et du framework Flask.

1.2.4 Vincent Pietri(ingénieur hardware)

Vincent est l'informaticien, et est chargé du bon fonctionnement des services informatiques sur l'ensemble des sites, que ce soit à Gap ou dans les secteurs.

1.2.5 Élie Bouttier(développeur Geonature)

Pendant le premier mois du stage, Élie était présent. Il est parti fin mai à la fin de son contrat. Élie est un développeur apprécié

1.2.6 Pierre Narcisi(développeur)

Pierre est venu en pour renforcer l'équipe travaillant sur le software Geonature.

1.2.7 Aurélien Coste(stagiaire intégration IA)

Un autre stagiaire, Aurélien est en stage de M1. Il travaille sur les données de fréquentation du parc, mais aussi sur la mise en place potentielle de l'IA à ces fins, et pour son utilisation possible pour détecter d'autres animaux.

1.3 Objectifs du stage

1.3.1 Développer des formulaires de saisie mobile sur ODK Collect

Avec l'utilisation du software ODK, la création de formulaires que les agents du parc chargés des observations peuvent utiliser pour améliorer la chaine de travail.

1.3.2 Déployer ces formulaires sur les téléphones des agents

Ceci serait simplement le téléchargement des bons formulaires sur les téléphones concernés.

1.3.3 Assurer la formation des agents (en binôme avec le chargé de mission base de données)

Il s'agit de faire connaître la plateforme ODK Collect aux agents.

1.3.4 Assurer la transmission des données dans l'outil GeoNature

Avec pyODK et odk2gn, créer un moyen de transférer les données sur le software et la base de données du parc.

1.3.5 Intégrer les données historiques des protocoles scientifiques dans GeoNature

Ceci n'a pas vraiment été abordé lors de la première partie du stage.

1.3.6 Participer à la diffusion des données au grand public

Ceci n'a pas vraiment été abordé lors de la première partie du stage

1.3.7 Documenter et illustrer les outils mis en œuvre

Le code faisant le lien entre ODK et Geonature n'existant seulement depuis quelques mois, il s'agit d'améliorer sa documentation à l'implémentation des nouvelles fonctionnalités.

1.3.8 Participer à la modernisation et à la structuration du SI du Parc national des Écrins

La création de formulaires ODK et leur utilisation par les agents permettrait d'avoir une meilleure manière d'obtenir des données pour les protocoles scientifiques, et de mieux intégrer leurs résultats dans la base afin d'avoir un meilleur suivi de la biodiversité dans le parc à long terme.

1.4 Méthodes de gestion

1.4.1 Scrum quotidien

Tous les matins à 9:15, l'équipe effectue un SCRUM, ou chacun présentait très brièvement les taches effectués la veille, les problèmes rencontrés, et les taches qu'il a prévu pour la journée. Ceci est fait en personne pour les personnes présentes sur le site, mais également en visioconférence avec les organisations partenaires, et les membres du pôle étant en télétravail ce jour-là.

2. Missions réalisées

2.1 Étude de l'existant et technologies utilisées

2.1.1 Le framework Python Flask

Le langage de programmation utilisé est le Python. Du fait de son utilisation par des dizaines de millions de personnes à travers le monde, et des nombreuses librairies existantes, c'est un outil très puissant dans le développement de logiciels.

Il y a de nombreux frameworks basés utilisant Python. Les applications crées au parc des Écrins sont écrits avec le framework Flask. Ce framework est léger, n'ayant que peu de librairies. Flask possède des extensions permettant de réaliser certaines fonctionnalités, telle la validation de formulaires, l'authentification, ou l'utilisation de mappings objet-relationnels.

Flask est également compatible avec un grand nombre d'autres librairies standards de Python, ce qui est une force de ce framework, permettant à d'autres développeurs Python d'utiliser des méthodes et pratiques qu'ils ont pu connaître autre part dans leurs applications.

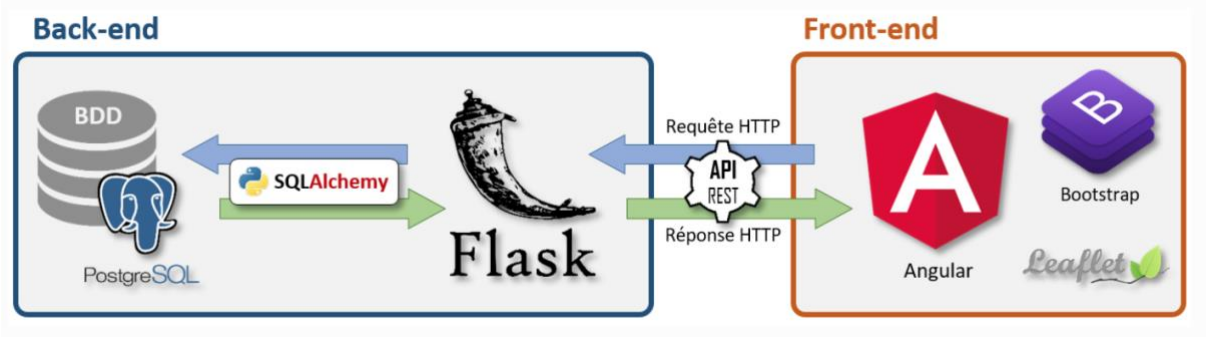
2.1.2 Geonature

Geonature est l'application crée par le parc pour gérer les données de faune et de flore dans le parc. Cette application a été créée dans sa première version en php en backend et en ExtJS en frontend pour sa première version en 2010. Elle était destinée d'abord à une utilisation uniquement par le parc. Depuis, la version 2 est sortie, cette version est codée en Python et structuré avec Flask pour le backend,

tandis que le frontend est en Angular et utilise Bootstrap. Le backend utilise également le mapping objet-relationnel (ORM) SQLAlchemy.

Un tel ORM nous permet à la fois d'éviter de mélanger du code SQL dans du code Python, améliorant la lisibilité, et également d'obtenir directement des objets Python, plus faciles à manipuler, mais également à construire et à mettre en base de données.

Voici un schéma du fonctionnement de l'application de Geonature.

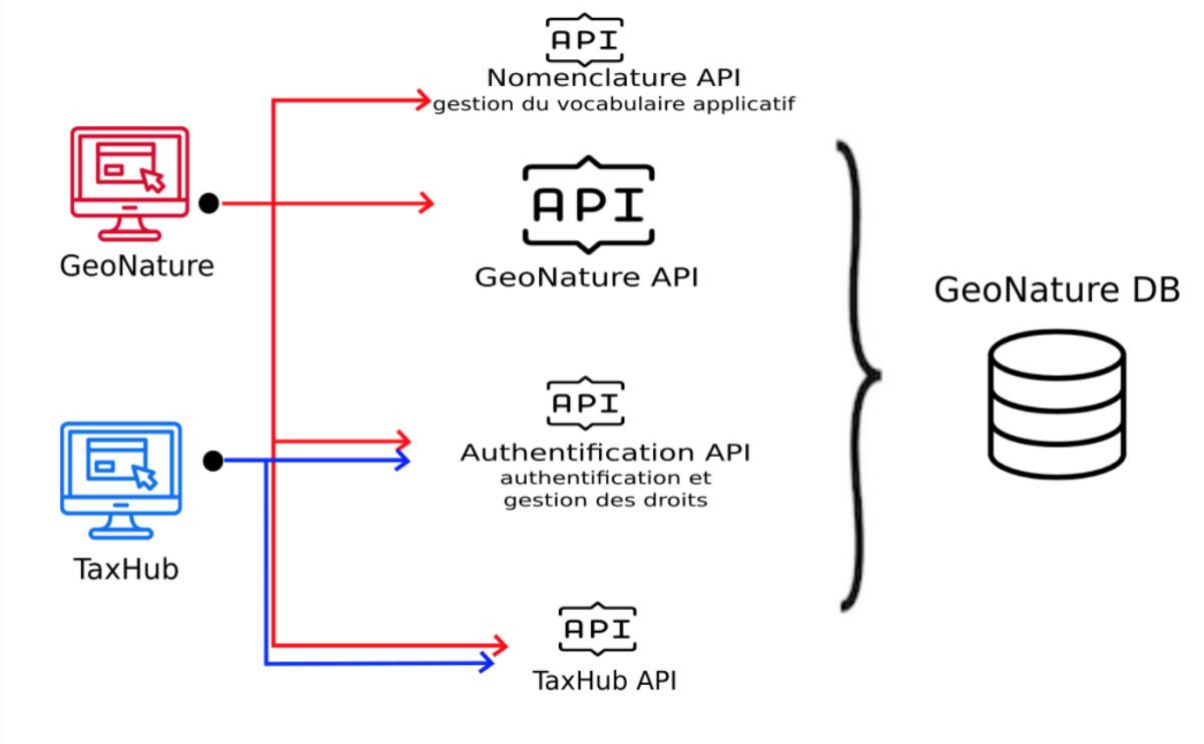


L'application est open-source, et elle est utilisée actuellement par une centaine de structures en France, dont des parcs nationaux, des métropoles (dont Grenoble), et par d'autres structures gérant des données de faune et de flore. La base de données utilisée est une base PostgreSQL.

Geonature n'est pas composée seulement d'une seule application. C'est l'application cœur qui gère un ensemble de modules et les données que ces modules utilisent. Pour la gestion des taxons (les espèces), il y a une autre application : TaxHub. TaxHub est mis à jour tous les ans avec les changements sur la liste du ministère de la biodiversité. Une utilité particulière de TaxHub est de gérer des listes de taxons pour les divers modules Geonature. TaxHub utilise la même base de données que Geonature.

Pour gérer les listes d'utilisateurs, on peut utiliser le bloc UsersHub, qui permet également de créer et gérer des listes.

Voici une schématisation des différentes applications qui utilisent la même base de données Geonature pour une installation simple.



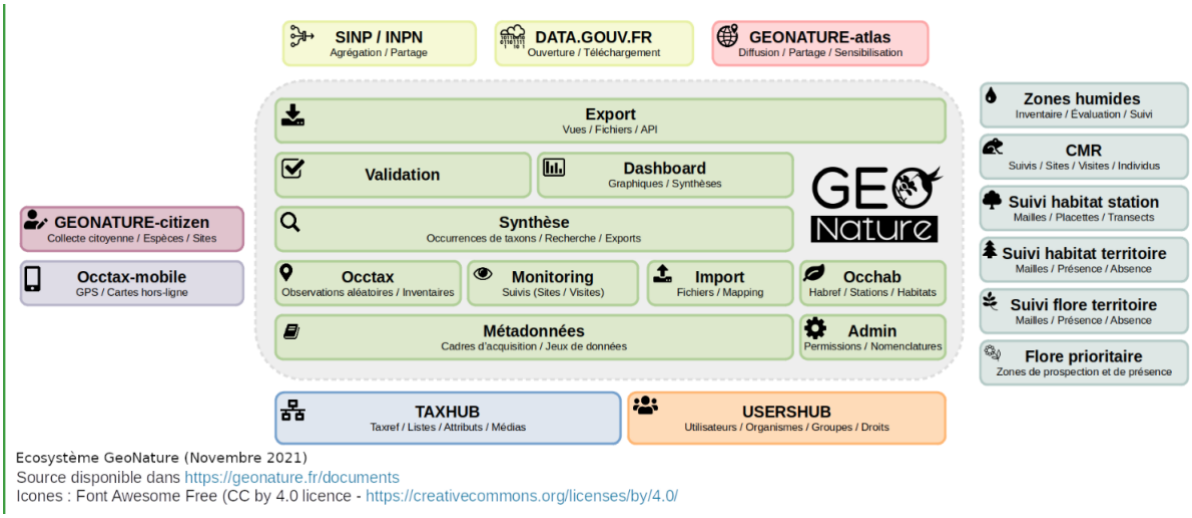
Geonature et Taxhub sont les deux applications indispensables dans une telle installation, et reposent bien sur une seule et même base de données. Les autres applications existantes le font également, mais n'étant pas forcément nécessaire à tout développement, ne sont pas sur ce schéma.

Geonature est composé d'une multitude de modules. Le module OccHab est utilisée pour stocker des données relatives à des habitats. Le module OccTax (abréviation d'Occurrences de Taxons) est celle avec lequel peuvent être faits certaines observations pas nécessairement structurées. Les modules ayant le plus de rapport avec le stage sont les modules Monitoring et Flore Prioritaire.

Pour toute observation dans toute module ou sous-module, afin de standardiser les valeurs de retour descriptifs de l'état des êtres vivants observés, ainsi que celui des sites, nous avons une table de nomenclatures. Ces nomenclatures sont utilisées pour une très grande quantité de catégorisations, de l'état de vie d'un animal, au typage d'un site d'observations et bien d'autres utilités également. Ils possèdent tous un id, un type, défini dans une table de types de nomenclatures, un code, et des labels en français, et un label par défaut.

Voici un schéma représentant Geonature et tous ses modules. Les modules en vert dans la boîte grise sont les modules plus fondamentaux qui appartiennent à l'application Geonature à proprement parler. En haut, le SINP et l'INPN sont des institutions qui définissent les taxons qui sont dans la base de données, et Atlas est l'outil de diffusion de données. À gauche nous avons Geonature-citizen, l'application pour la collecte citoyenne de données, et Occtax-mobile, une application mobile qui est une version mobile du module Occtax de collecte d'observations aléatoires. Sur le côté droit sont des modules extensions de Geonature, dont plus particulièrement le

module flore prioritaire. Enfin, en bas sont les applications TaxHub et UsersHub qui définissent les listes de données de taxons et d'utilisateurs utilisés dans les autres modules.



Expliquons rapidement comment installer Geonature : dans un dossier de l'arborescence dans l'ordinateur appelé Geonature, il faut installer la dernière version existante de l'application Geonature, l'application TaxHub, et les divers modules Geonature sur lequel vous voulez travailler. Dans notre cas il s'agit du module Monitoring, du module flore prioritaire ou bilan stationnel et de la librairie odk2gn trouvée sur GitHub. Nous créons comme pour le microblog un serveur de données en local spécifique, et nous configurons Geonature et TaxHub à l'utiliser. Nous configurons les modules à utiliser les bonnes listes d'utilisateurs et de taxons. Pour odk2gn la configuration signifie donner au module les informations nécessaires pour accéder au serveur ODK Central que l'on souhaite.

Pour lancer Geonature, il faut aller dans son environnement virtuel, puis son dossier backend et le lancer avec la commande ‘’geonature dev-back’’. Le backend de Geonature tourne alors sur le port 8000. Ensuite, il faut passer dans le dossier frontend et le démarrer avec la commande ‘’npm run start’’. Le frontend tourne alors sur le port 4200. TaxHub se démarre dans son propre environnement virtuel dans son dossier avec la commande ‘’flask run’’, et tourne sur le port 5000.

2.1.3 Le module Monitoring

Le module Monitoring est le module qui sert pour les observations dans le cadre de protocoles scientifiques. Ce module est divisé en une multitude de sous-modules, chacun servant pour un protocole scientifique spécifique. Le module monitoring est basé sur un tronc commun :

- les sites (correspond à la zone qui va être prospecté dans le protocole scientifique (par exemple un point d'écoute des oiseaux, un gite de chauve-souris)
- les visites (on fait des visitez sur les sites)
- les observations (lors des visites on fait des « observations »)

chaque sous-module pouvant étendre les champs saisissables à chacun de ces niveaux

Le fait que ces sous-modules ont tous la même structure fait qu'il y a une généricité exploitable à la fois dans leur installation mais plus encore lors de leur

manipulation, ce qui est très utile. Tous les sites de tous les sous-modules sont définis dans une seule et même table, et de même pour tous les visites et toutes les observations.

Le module de flore prioritaire ressemble à un module monitoring, mais les données stockées ne sont pas les mêmes. Lors du protocole flore prioritaire, nous créons d'abord une zone de prospection pour une espèce, c'est-à-dire une grande zone ou toutes les données seront prises. La capture de données à lieux dans une aires de présence, plus petite surface présente à l'intérieur de la zone de prospection. Dans ces zones sont faits le comptage et les caractéristiques spécifiques au sous-module, telle la présence d'éléments potentiellement dangereux pour les plantes recherchées, ou l'état de vie. Il existe également d'autre modules spécifiques à d'autres protocoles qui ne peuvent pas utiliser la généricité de monitoring, comme pour les zones humides.

2.1.4 ODK

ODK est une plateforme de collecte de données mobile par des formulaires. Avec ODK, on peut remplir un formulaire hors-ligne, et l'envoyer sur un serveur. ODK est composé d'une application mobile, ODK Collect, et d'une plateforme en ligne, ODK Central. Le parc utilise ODK pour faire du suivi de protocole scientifiques sans avoir à développer leur propre application mobile.

Un serveur ODK Central possède des projets, dans le cadre desquels sont utilisés les formulaires. Pour créer un formulaire ODK, il faut créer un fichier contenant un nom, un type de donnée de retour, et un label pour chaque question. Ce fichier est de format XLSForm. Les types de retour possible incluent des nombres entiers ou décimaux, des blocs de texte, des choix (définis dans le formulaire ou dans des fichiers séparés, ou encore des données géographiques qui déterminés sur une carte. Les choix peuvent également être des données calculées à partir de chiffres fournis dans une autre question. Si l'on utilise un fichier externe pour des choix (un fichier de format CSV est le cas le plus fréquent, on peut également filtrer les choix. Ce fichier contenant les questions ainsi que les éventuels fichiers externes nécessaires à sa création doivent être téléversés sur ODK Central.

À partir de là, nous pouvons entrer les valeurs des soumissions soit depuis l'outil Enketo de réponse de questionnaires si l'on a accès à ODK Central, ou depuis l'application mobile ODK Collect. Collect permet de collectionner et stocker des données de formulaire même quand il n'y a pas de service permettant d'accéder à Internet. C'est une application open source Android qui permet pour un utilisateur mobile créé sur ODK Central de télécharger ses formulaires et de les envoyer à Central quand c'est possible. Ceci se fait en scannant un QR-code sur ODK Central.

Pour voir les résultats d'enquêtes faites avec ODK, il y a plusieurs options. ODK Central possède d'abord son propre API pour accéder aux soumissions. Nous pouvons également exporter les soumissions en format CSV. De plus, depuis quelques mois, les données peuvent également être exportés en Python avec la librairie pyODK.

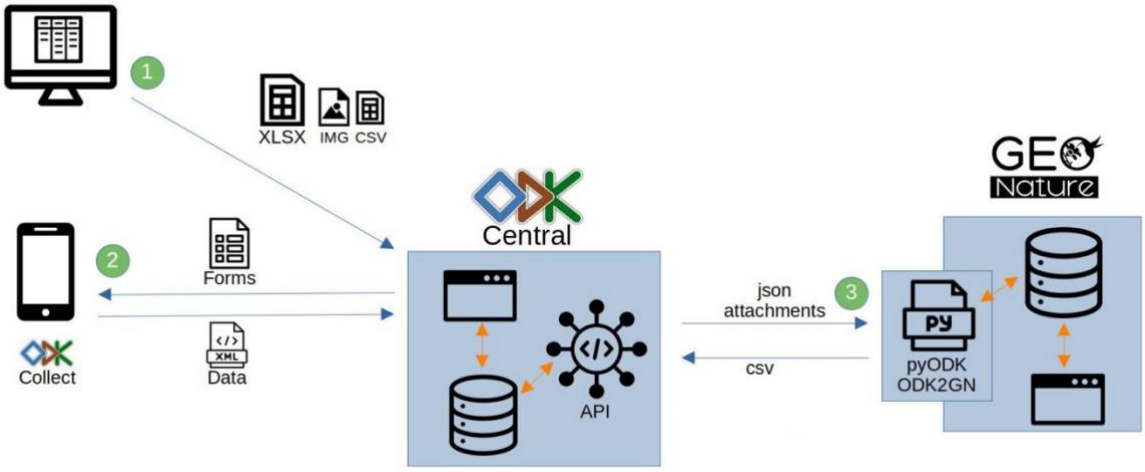
2.1.5 pyODK et odk2gn : Interactions entre ODK et Geonature

Une fois que nous avons des données avec ODK, il est temps de les stocker. ODK Central le fait mais le parc veut avoir les données dans leur propre base de données, c'est-à-dire celle de Geonature. Il faut donc un moyen de prendre les données depuis ODK Central, et de les formatter pour pouvoir les envoyer à la base Geonature. Il faudrait donc quelque chose pour pouvoir obtenir les données transmises à ODK en un format lisible par Python. Ceci est l'utilité de la librairie pyODK.

pyODK est la librairie officielle faisant le lien entre ODK et python. Cette librairie à comme élément le plus important la classe Client, qui se paramètre avec l'URL du serveur sur lequel nous travaillons. Un Client possède une session, qui permet d'accéder aux données à proprement parler. Avec la session, nous avons accès aux objets projet (Project), et soumission (Submissions), d'où sont extraites les données.

Pour une utilisation spécifique à Geonature, une librairie odk2gn a été créée lors d'un workshop fin 2022. Le parc se sert de ce module pour avoir un outil mobile pour les protocoles scientifiques, donc pour des modules Monitoring et autres modules spécifiques, tel celui de flore prioritaire. odk2gn a des méthodes permettant la lecture, le formatage et l'ajout en base de données des données ODK, cette méthode s'appelle synchronize. De plus, a été créée une méthode de mise à jour des formulaires et de génération de leur fichiers csv associés, permettant de faire toutes les manipulations nécessaires depuis la ligne de commande, avec une méthode nommée upgrade-odk-form.

Voici un schéma expliquant l'utilisation d'ODK et odk2gn faite par le parc dans le cadre des protocoles scientifiques.



D'abord (1), le formulaire est créé avec ses fichiers d'image et de choix attachés, puis envoyé sur le serveur Central. Ensuite (2), les utilisateurs mobiles sont créés, ils chargent sur ODK Collect les formulaires, et renvoient des soumissions à ces formulaires. Enfin (3), les soumissions sont transmises à Geonature et intégrés à la base, tandis que les fichiers csv du formulaire peuvent être mis à jour depuis Geonature.

Dans odk2gn tel que le dossier était à mon arrivée, il existait un sous-module Monitoring avec une implémentation qui était déjà fonctionnelle et un formulaire existant. Il s'agissait du sous module pour le suivi temporel des oiseaux de montagne, ou STOM. Ce module est celui avec lequel j'ai abordé Geonature pour la première fois. Les fonctionnalités pour synchroniser les données et mettre à jour le formulaire étaient déjà fonctionnels pour ce sous-module. Une première version de formulaire pour les chiroptères était également existante, mais n'était pas implémentée dans odk2gn. Ces deux protocoles avaient été développés lors d'un workshop fait avec le Parc National des Cévennes.

La particularité de l'utilisation d'odk2gn ici est dans le fait que vu la généricité des sous-modules et de leurs composantes, il faut que le code soit très générique pour pouvoir créer les fichiers nécessaires à chaque sous-module, et à l'extraction des données soumises depuis ODK Central.

La librairie pyODK est la librairie sur laquelle se repose partiellement odk2gn. Cette librairie contient une classe Client, ainsi que quelques classes représentant les divers projets, formulaires, et soumissions en particulier. La première chose à faire dans cette librairie est de la configurer, en lui fournissant l'URL de l'instance d'ODK Central sur lequel sont les formulaires, et de nos identifiants permettant d'y accéder.

Les fonctionnalités demandées dans notre extension qui utilisent les méthodes de pyODK et de sa classe Client sont dans un fichier spécifique. La majorité des fonctions font des requêtes sur un URL spécifique. Si l'on utilise la classe Client pour faire ses requêtes, nous pouvons nous permettre de ne pas réécrire systématiquement l'URL du serveur. Ces URLs sont de la forme ‘’/projects/<id du projet>/forms/<id du formulaire>/submissions<uuid de la soumission>/attachments/<nom du fichier>’’ au plus long, ou seulement les premières parties de l'URL selon ce sur qui est recherché.

Pour ce qui est des méthodes elles-mêmes, tout d'abord, il y a une fonction get_attachment qui récupère les fichiers médias (photos en particulier), s'ils existent. Ensuite, nous avons une méthode get_submissions pour récupérer les nouvelles soumissions, puis une méthode update_review_state pour prouver que les soumissions ont été relues et ne seront pas pris en compte plusieurs fois. De plus, nous avons une méthode form_draft pour mettre un formulaire en état de brouillon, une méthode update_form_attachment pour mettre à jour les fichiers nécessaires pour l'élaboration du formulaire, et une méthode publish_form pour le publier et faire en sorte qu'on accepte de nouveau les soumissions. Enfin nous avons une méthode get_schema_fields qui récupère les champs du formulaire et les formatte en JSON. Il existe aussi une classe pour créer des objets schémas.

Les autres fichiers sont tous composés de code qui n'utilise pas directement des méthodes pyODK, mais qui sont plus liés à Geonature et à sa base de données. Un premier fichier qui existe récupère la configuration du module dans lequel nous travaillons. Un second définit des schémas pour un site, une visite et une observation. Ce sont les autres fichiers qui sont plus intéressants. Un premier contient les méthodes pour créer les fichiers à téléverser sur ODK Central, c'est-à-dire ceux pour récupérer le sous-module, les taxons, les observateurs, les jeux de données, les sites et les nomenclatures nécessaires depuis la base, ainsi qu'une méthode pour formatter les résultats de ces méthodes en format csv. Un second contient les méthodes pour parcourir une soumission, et d'en extraire les données pour créer un objet visite et des objets observations À l'instant nous ne pouvons pas encore créer des sites avec ODK.

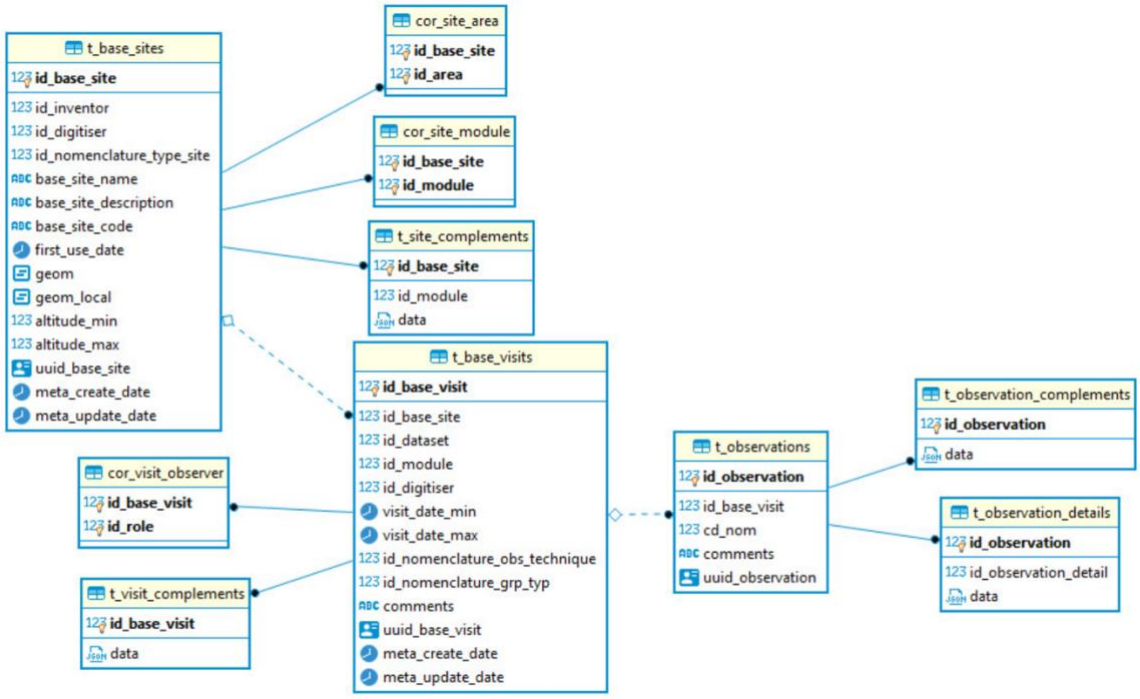
Nous arrivons enfin au fichier contenant les fonctions en ligne de commande pour mettre à jour ces formulaires, et d'en extraire les résultats. La méthode upgrade-

odk-form prend en paramètres le module et quels fichiers à mettre à jour, détermine le module et crée les fichiers à téléverser, met en brouillon le formulaire, met à jour les fichiers, puis publie le formulaire.

La méthode synchronize crée des objets schéma pour la soumission, et un deuxième pour le module. Il récupère ensuite la configuration, dont les champs génériques et spécifiques du module, et puis les soumissions. Pour chaque soumission, une liste d'observation est créée, puis l'objet visite est créé, et ses photos sont extraites et postés, et ensuite nous faisons de même pour chaque soumission et leurs photos. Ensuite nous essayons de tout poster en base de données. Si oui, la relecture est approuvée, sinon, il est affiché que la soumission comporte des erreurs.

2.1.6 Conception et schématisation des modèle de données

Voici le schéma de données du module monitoring. Ceci est une partie d'une plus grande base de données, mais le reste de la base n'a pas forcément un lien avec le stage. Ces tables se trouvent dans un schéma gn_monitoring de la base. Les noms de objets définissant les tables en sont en camel-case dans le code source et non en snake-case car les classes d'objets suivent cette norme, et SQLAlchemy les passe en snake-case.



La table t_base_sites représente les sites où ont lieu les observations. Un tel site a besoin au minimum de l'id de nomenclature de type de site, d'un nom, et d'une géométrie (geom). Dans ses tables de correspondance, l'id_area sort d'une table de références des communes, départements et régions et l'id_module sort d'une table des modules et sous-modules. La ligne data dans la table t_site_complements est un JSON contenant des informations spécifiques à un module sur un site.

La table t_base_visits stocke les objets visite, qui se font sur un site. L'id_dataset dans la table vient d'une table de jeux de données défini dans un autre schéma. Dans ses tables de correspondance, l'id_role appartient à la table des utilisateurs. La colonne data est un JSON qui contient toutes les informations spécifiques sur les visites dans chaque module.

La table t_observations stocke les objets observation, ils sont associés à un site. L'attribut cd_nom est le code du taxon observé, qui est défini dans une table dans un autre schéma. La table t_observation_complements contient la colonne data qui, comme pour les sites et les visites, contient les informations spécifiques à un module.

2.2 Réalisation et tests

2.2.1 Montée en compétences sur Flask et Python

Les premières semaines du stage ont été consacrés à une montée en compétence en Python et d'un premier abord de Flask en suivant un tutoriel en ligne. Ce tutoriel a été fait par Miguel Grinberg et permet d'aborder la majorité des éléments utiles dans Flask. Ceci se fait par la création d'un site de microblogging, dans le style de Twitter. De plus, nous avons ajouté une ou deux fonctionnalités non présentes dans le tutoriel à cette application.

La première étape dans la création d'une application Flask est la préparation du terrain. Ceci se fait par l'installation d'une version de Python, et de l'outil pip qui permet l'installation de packages Python. Ensuite vient la création d'un dossier et d'un environnement virtuel pour l'application, ainsi l'application et seule l'application sera concernée par les packages nécessaires à elle qui ne le sont pas forcément pour chaque application. L'installation de la librairie de base Flask vient ensuite. La première utilisation de toute technologie de code est de voir comment se fait un programme très simple, le « Hello, World ».

L'initialisation d'une application se fait par le biais d'un dossier app, où seront contenues tous les dossiers et fichiers nécessaires au fonctionnement d'une telle application, dont le fichier __init__.py, contentant la création de l'objet app, et de l'import des routes et des modèles nécessaires. Flask possède des éléments ressemblant à un modèle Modèle-Vue-Contrôleur, mais se diffère d'un modèle MVC car il n'y a pas vraiment de contrôleurs, en effet tout ce qui serait dans un contrôleur, l'interaction des objets, se gère au niveau du routage de l'application, dans un fichier routes.py. Les objets et les classes sont créées dans un fichier models.py. Le premier Hello, World se fait donc dans la route index() du fichier routes.py. Enfin, dans une application Flask existe un script de haut niveau définissant l'application, ici ce serait un fichier microblog.py.

Pour faire tourner une application Flask, il faut lancer la commande « flask run » qui lance l'application sur un localhost au port 5000.

Les premières vues vont dans le dossier app/template. Le premier à créer est la vue index. Les fichiers html pour l'application sont assez basiques avec quelques

zones dynamiques (entre deux accolades dans le code). Dans la plupart routes sont utilisés en retour une fonction render_template de Flask qui permet de créer les vues avec comme données ce qui est passé en paramètre, comme un nom d'utilisateur si l'on veut qu'il apparaisse sur la page.

Avant de créer plus de pages, il serait utile de créer un template de base pour faciliter la création de toutes les autres pages. Ce template serait dans un fichier base.html, et contiendrait toute les fonctionnalités et éléments présents sur chacune des pages, c'est-à-dire l'en-tête, certaines métadonnées, le titre présent sur les pages et la barre de menu permettant le changement de page, et un bloc content qui varierait selon la vue. Ainsi, sur chaque vue suivante, nous pourrions avoir juste la ligne {% extends ''base.html'' %} en haut de chaque fichier pour ne plus avoir à gérer la structure de chaque page.

La configuration de l'application se fait dans un fichier config.py, et contient des clés pour chaque élément qui serait soit variable selon le développeur. Tous ces éléments sont des attributs d'une classe Config, maître d'un objet app.config associé à l'application.

Un élément qui fait la puissance de Flask est ses nombreuses extensions. Il y en a beaucoup, ce qui permet à un développeur d'avoir vite accès à des outils puissant. Pour la création des formulaires utilisés pour poster, il y a l'extension Flask-WTF, qui possède des classes de formulaires héritables pour créer des pages de formulaire. Son import donne également accès à une multitude de classes pour les différents champs, que ce soit pour des zones de texte à encrypter ou non, pour des booléens, ou pour la validation. Dans cette application tous les formulaires sont dans un fichier forms.py

Un exemple serait celui pour le login d'un utilisateur. La classe LoginForm hérite d'une superclasse de formulaires flask, et possède comme attributs un champ texte basique pour créer un pseudonyme, un champ texte d'un type spécifique aux mots de passe pour ceux-ci, un booléen d'option pour se souvenir de l'utilisateur et d'un champ de validation, qui serait un bouton HTML.

La vue de ce formulaire, login.html, contient un titre, une balise <form> contenant les formulaires, et des éléments <p> pour chaque élément du formulaire. L'action pour un tel formulaire serait un POST.

Pour le routage, nous créons dans le fichier des routes une fonction login associé à la route '/login' qui permet de générer la vue demandée. Dans cette fonction nous créons un objet LoginForm, que nous mettons dans la fonction render_template pour obtenir la vue. Nous ajoutons cette route dans le template de base de l'application et nous avons un menu login qui est accessible.

Pour gérer la validation, dans le la package Flask-WTF, il existe pour la classe mère des formulaires une méthode qui indique si la validation à lieu ou non. Nous associons au décorateur de route les méthodes nécessaires, GET et POST ici, et nous appelons la fonction testant la validation, et si le formulaire est valide nous redirigeons.

Nous avons désormais le type de formulaire existant pour que notre plateforme accepte l'inscription d'utilisateurs, il faut maintenant pouvoir représenter ces utilisateurs. À ces fins, nous avons utilisés une base de données Postgres, bien que non figurant dans le tutoriel c'est ce qui est utilisé par le parc pour Geonature, et la compatibilité de Flask avec l'ORM SQLAlchemy. Nous configurons l'application à utiliser la base dans le fichier config.py. Ensuite, dans un fichier models.py, nous

créons le modèle pour l'utilisateur, définie dans une classe User. Un objet de cette classe possède un id (compatibilité avec la base de données oblige), un pseudonyme, une adresse électronique associée, et un mot de passe. L'attribut id représentera la clé primaire pour toutes les classes ici.

Pour migrer tout ceci en base, nous utilisons la librairie Flask-Migrate. Avec la commande ''flask db init'', nous créons un dossier de migrations, dans lequel figurent toutes les migrations nécessaires à la création de la base. Toute modification de la base y sera également le moment venu, en créant une migration avec ''flask db migrate -m <nom-de-la-migration>. La mise à jour est poussée avec ''flask db upgrade''. De plus, si l'on veut passer à une version antérieure de la base, nous pouvons le faire avec ''flask-db-downgrade''.

Un utilisateur doit pouvoir poster sur le site de microblog, d'où la création de la classe Post. Cette classe possède un id pour la base de données, un corps (le contenu), un timestamp pour signaler le moment exact où le post a été soumis, et l'id de l'utilisateur auteur. Afin de faire le lien entre les deux tables, dans le cadre de relations one-to-many, un attribut de type Relationship est utilisé dans la table où figure nous avons l'élément seul. Dans la table avec plusieurs éléments, c'est la colonne typée comme une colonne normale, mais nous indiquons que c'est bien une clé étrangère. Ces objets sont passés à la base avec les fonctions add, puis commit de la session de la base de données.

Pour créer un post, nous ajoutons sur la page index un formulaire dédié pour créer un post, contentant un champ texte et un bouton de soumission. La soumission n'a lieu uniquement si le champ texte n'est pas vide. Nous adaptons la vue index pour pouvoir contenir ce formulaire, et une route qui ajoute le post dans la base et dans la page.

Afin de sécuriser les comptes utilisateurs dans une telle application, il existe une librairie Flask nommée Flask-login. Les premières utilités de cette librairie implémentée ici est la méthode de hachage des mots de passe, qui les rend indéchiffrables, par un humain, mais qui sont vérifiables par l'application. Certaines autres méthodes sont utilisées ici, surtout les divers booléens qui vérifient que l'utilisateur a des identifiants valides, et que le compte est actif. Il existe également une fonction qui attribue et retourne un id unique à un utilisateur. Cependant, cette librairie n'est pas associée par défaut à une base de données, donc il faut une méthode loader pour faire le lien dans le sens base – application. La route login peut être adaptée avec la méthode de login de la librairie, et une route logout peut être créée avec la méthode de la librairie. Un décorateur peut être utilisé pour rendre obligatoire la connexion d'un utilisateur. Nous créons enfin un menu d'inscription assez classique, avec des méthodes pour vérifier l'unicité du pseudonyme, évitant ainsi un problème lors de l'écriture en base.

Une page profil est créée avec une route /user/<pseudonyme>, cette route exige que l'utilisateur soit connecté. Cette page affiche les posts de l'utilisateur. Il est possible ici d'ajouter un avatar pour chaque utilisateur dans un attribut (à l'aide du site Gravatar). Nous adaptons ici l'affichage des posts avec des sous-templates très simples, contenant l'avatar de l'auteur, son pseudonyme et le contenu du post. Nous utilisons désormais ce sous-template lors de la création des pages utilisateur et index. Nous ajoutons à cette page la possibilité d'avoir une petite description, et l'affichage

de la dernière visite sur le site de l'utilisateur. Afin de pouvoir modifier un compte, nous créons un formulaire Flask, une vue et une route.

Pour gérer des éventuelles erreurs, nous créons des vues pour les types d'erreur 404 (objet non trouvé) et 500 (erreur interne). Leurs routes affichent seulement les différentes pages d'erreur (et effectue un rollback sur la base de données dans le cas de l'erreur 500). Sur les vues, nous créons un lien pour revenir à la page index. Il n'y aura que très peu de possibilités dans cette application basique d'obtenir des erreurs, nous lançons une exception lors du cas où un utilisateur essaie de s'inscrire avec un pseudonyme figurant déjà dans la base.

Dans un site de microblogging tel que nous le créons, l'idée de pouvoir s'abonner à un utilisateur pour voir ces postes est un principe assez fondamental. Nous implémentons ceci avec une relation n-n entre la table des utilisateurs et lui-même. Une telle table s'implémente avec une table ne figurant pas en-dehors des classes, contenant l'id de l'utilisateur abonné, et l'id de l'utilisateur auquel il s'abonne. Nous ajoutons à ceci un attribut followed, qui est une relationship avec la table User, dans la classe User. Afin de gérer les abonnements, nous ajoutons à cette classe des méthodes pour s'abonner et se désabonner (qui sont en fait juste l'ajout et la suppression d'un objet dans une liste). Nous ajoutons également une méthode retournant un booléen vrai si l'utilisateur est abonné à un autre en particulier qui est utilisé dans sa page. Afin d'obtenir les posts des profils auxquels nous nous sommes abonnés, nous faisons une requête SQLAlchemy avec des jointures sur les deux tables User et Post de la base pour les retourner. Pour compléter ceci, nous affichons également les posts de l'utilisateur connecté. Enfin nous créons des routes pour gérer ces abonnements, puis adaptons les pages utilisateur pour ce qu'ils affichent le bouton de modification de profils si l'utilisateur est sur sa propre page, un bouton pour s'abonner si l'utilisateur est sur la page d'un autre utilisateur auquel il n'est pas encore abonné, ou un bouton pour se désabonner si l'on est sur la page d'un utilisateur auquel nous sommes abonnés.

Afin d'améliorer la lisibilité des posts dans l'application, nous créons un système de pagination. Nous créons d'abord une route /explore qui permet de voir tous les posts de tous les utilisateurs. Pour ce qui est de la pagination, nous injectons dans le fichier de configuration le nombre de posts par page que nous désirons, puis nous utilisons une méthode paginate de la librairie SQLAlchemy avec en paramètre les posts et ce nombre de posts par page. Cette méthode est dans une classe SQLAlchemy, dans lequel figurent des booléens pour savoir si existeraient des pages précédentes et suivantes. Nous ajoutons des boutons pour parcourir ces pages dans toutes les routes des pages où ceci serait utilisé, ainsi que dans leurs vues.

Notre frontend n'est toujours pas très beau à ce moment. Pour gérer le frontend, Flask possède une librairie Bootstrap. L'import de cette librairie fait en sorte que toutes les pages hériteront d'une vue bootstrap/base.html. Dans les pages où figurent des formulaires, nous importons la librairie bootstrap/wtf.html, et utilisons la gestion des formulaires de Bootstrap. Avec quelques petits changements, nous avons un site nettement meilleur visuellement.

Continuons sur le thème d'éléments plus propres en utilisant les blueprints. Les blueprints sont la séparation des divers fichiers dans différents dossiers de la

hiérarchie selon leur utilité. Ici, nous séparons d'abord les quelques fichiers de routage pour la gestion des erreurs dans un dossier errors, ainsi qu'un fichier d'initialisation dans lequel le blueprint est créé. Nous créons ensuite un deuxième blueprint de la même manière pour l'authentification qui contient en plus la définition de ses types de formulaires. Nous en créons un troisième dans lequel figurent toutes les vues, avec en particulier celles pour les erreurs et l'authentification dans des dossiers spécifiques. Enfin, nous en créons un dernier appelé main pour les fonctionnalités principales de l'application, contenant les routes et formulaires nécessaires à ceci, mais ne contenant pas les modèles.

Une fonctionnalité que nous souhaiterons d'ajouter à notre application est un système d'envoi et de réception de messages directs. Une classe Message est implémenté, avec un id, les id de l'utilisateur qui envoie le message et de celui qui le reçoit, un corps, et un timestamp pour indiquer quand le message a été envoyé. Nous ajoutons dans la classe User des relationships avec la classe Message pour stocker les messages envoyés et reçus, ainsi qu'à l'heure de lecture du dernier message. Afin de voir si des nouveaux messages ont été reçues depuis, nous créons une méthode qui compte les messages reçus depuis la dernière lecture.

Pour l'envoi des messages, nous créons un formulaire, une vue, et une route. Nous ajoutons dans la page utilisateur un bouton pour créer et envoyer des messages. Cependant pour les voir nous allons utiliser une autre vue et une autre route. Dans la vue, nous utilisons le sous-template des posts car les messages vont leur ressembler.

Nous utilisons la méthode pour le comptage des nouveaux messages pour créer dans la vue de base un badge contenant ce nombre s'il n'est pas nul. Ceux-ci sont une sorte de notifications dans notre application. Nous ajoutons un court script JavaScript pour que le badge ne soit pas visible lorsqu'il n'y a pas de messages non lus. Pour mieux manipuler ces notifications, nous créons une classe Notification, contenant un id, un nom, l'id de l'utilisateur qui les reçoit, un timestamp, et du texte formatté en JSON. Nous adaptons la configuration, la base de données et les routes, et créons une route /notifications et nous avons cet élément de configuré.

Une Application Programming Interface ou API est également ajouté. Pour ceci nous créons un blueprint API, puis des routes pour les fonctions que nous souhaiterions y implémenter. Une particularité des API est que nous ne pouvons pas forcément avoir une authentification par mot de passe. Il faut donc implémenter une authentification par token. Pour une authentification de ce type, nous utilisons la librairie Flask-HTTPAuth. Cette librairie contient des méthodes de vérification de mots de passe, mais également de création de tokens. Nous utilisons sa classe d'authentification par tokens et protégeons les routes avec un décorateur à ces fins. Les tokens sont retirés avec une méthode spécifique également.

Une autre particularité de l'API est la représentation des objets à laquelle les classes Python ne sont pas très adaptés. Cependant, nous pouvons les convertir en format JSON, en les transformant en dictionnaires Python.

Pour aborder la librairie pyODK d'une manière intéressante, nous avons décidé d'implémenter la création de posts depuis ODK Collect. Pour ceci, nous avons créé un fichier Python spécifique, un très petit formulaire ODK et la configuration pyODK. Le formulaire ODK contient deux éléments, un choix à partir d'un fichier csv de l'auteur du poste (qui ne sont pas créés ici mais dans l'application web), et un champ texte. Le fichier contient deux méthodes, un qui met à jour l'élément ‘’reviewState’’, de

relecture de la soumission de post, de manière qu'on n'ajoute pas plusieurs fois dans la base de données le même post, et une méthode qui envoie les posts en base et utilise la méthode précédente. Cette méthode récupère sur ODK central les nouvelles soumissions, les compte, prend leur contenu et créé un objet Post, et l'ajoute en base, avant de mettre à jour l'état de relecture.

Je vais enfin faire le lien entre l'application développée dans le tutoriel et Geonature. Les deux utilisent le même framework, avec les mêmes utilités d'authentification, le même ORM, et les mêmes principes pour leurs API entre autres éléments, et l'ajout de l'utilité d'ODK pour créer des posts fait un lien avec le sujet principal du stage.

2.2.2 Module Monitoring de Geonature et leurs divers formulaires mis en place

Dans odk2gn tel que le dossier était à mon arrivée, il existait un sous-module Monitoring avec une implémentation qui était déjà fonctionnelle et un formulaire existant. Il s'agissait du sous module pour le suivi temporel des oiseaux de montagne, ou STOM. Ce module est celui avec lequel j'ai abordé Geonature pour la première fois. Les fonctionnalités pour synchroniser les données et mettre à jour le formulaire étaient déjà fonctionnels pour ce sous-module. Une première version de formulaire pour les chiroptères était également existante, mais n'était pas implémentée dans odk2gn. Ces deux protocoles avaient été développés lors d'un workshop fait avec le Parc National des Cévennes.

Une fois ce premier abord fait, la prochaine tâche était de faire fonctionner odk2gn avec le sous-module des chiroptères. La première chose à faire était d'aborder le protocole existant et de le comparer avec celui qui existe dans Geonature. Étant réalisé par mes collègues qui travaillent sur l'application, c'était le cas. Cependant, une petite modification y a été apporté : l'ajout d'un booléen pour signaler si des chiroptères ont bien été observés. Ensuite, vérifier que les données sont bien extraites convenablement. Du fait que les questions du formulaire possèdent le même nom que les attributs des objets à ajouter en base, c'était le cas.

Le suivi des chiroptères n'était pas le seul module monitoring manipulé. Une manipulation du sous-module de suivi des pelouses de nard du parc a également été réalisée, à partir d'un template de formulaire qui ressemble à celui des chiroptères. Un formulaire ODK a été crée pour ceci, étant une copie des champs existant sur Geonature, et en veillant bien à nommer les questions de la même manière que les attributs génériques de Monitoring dans le cas où nous sommes dans ces colonnes de la base.

C'est ici qu'un autre objectif a été ajouté : une meilleure solidité des données. Ainsi, l'instruction a été donnée à ce que le jeu de données dans le cadre duquel l'enquête est faite ne peut pas être passée par l'utilisateur d'ODK Central. La question a donc été cachée sur le formulaire. Seulement, en faisant ceci, lors de l'étape d'extraction des données, l'exécution ne réussissait pas car l'intégrité des données était violée par le non-passage de l'id du jeu de données. Il fallait donc une seconde manière de passer en base l'id du jeu de données. Du fait que le csv contenant les jeux de données n'est censé contenir qu'une seule valeur, la récupération de ceci se renvoie un tableau avec une seule valeur, un tableau contenant son nom et son id.

L'id est en position 0, c'est lui qui est recherché, nous le prenons donc, et le passons dans le backend.

Un deuxième objectif d'amélioration de l'intégrité des données a été fixé. En effet, dans les données récupérées depuis ODK, certaines données doivent être déterminés par un choix parmi des nomenclatures, et ces résultats doivent être stockés sous forme d'entiers dans le JSON, et non des chaines de caractères de ces entiers, ou de leur label, alors qu'ODK ce sont des chaines de caractères. Donc un transtypage est requis, mais seulement sur les chaines de caractères qui représentent des nomenclatures, et seulement lorsque c'est possible. Un transtypage dans un bloc try/except a donc été mis en place, et ceci seulement pour des types chaines ou le widget Geonature est une nomenclature.

La troisième et dernière grande tâche réalisée dans ce cadre était également dans le cadre de renforcement du code, cette fois du côté de l'a créations des fichiers csv. Précédemment, ceci a été fait en construisant une chaine de caractère dans ce format, et les données ont été passés sous forme de tableaux. Ceci pose souci si l'ordre dans le tableau change, nous avons des fichiers csv qui ne correspondent absolument pas. Ceci a mené à un changement de tous les types de retour de tous les fonctions get. Ces fonctions font une requête SQLAlchemy, qui renvoient des tuples. Ces tuples ont été transformés en dictionnaires, soit par une méthode quand les requêtes avaient lieu sur une seule table, soit ‘'à la main'' afin de garantir une meilleure solidité de données. De plus, la méthode de formatage csv a été changé afin d'utiliser la librairie csv de Python, en créant des fichiers temporaires. Nous nous retrouvons avec des données à la fois plus lisibles et plus sûrs.

Avec ces implémentations, nous avons un formulaire pour le suivi des chiroptères qui fonctionne avec les ‘'fausses données'' fournies par Geonature. Nous avons ensuite importé une copie de données de la vraie base de données de production dans notre base de développement afin de vraiment tester ce formulaire. Et ceci a été un succès. Au moment de l'écriture, ceci n'a pas été fait pour les pelouses de nard.

2.2.3 Module flore prioritaire

Le module monitoring possède une manière bien structurée de fonctionner, avec son modèle site-visite-observation. Cependant, les protocoles scientifiques n'existent pas uniquement dans le cadre de ce module. Le module flore prioritaire, ou bilan stationnel, est un module Geonature qui suit l'évolution de certaines espèces de plantes, et se fait d'une manière complètement différente de monitoring. Ce module fonctionne d'abord en créant sur la carte une zone de prospection. Dans cette zone de prospection sont créés des aires de présence, et un taxon sur lequel la recherche de données est faite. Ces aires de présences sont des plus petites surfaces, et ce sont à leur niveau que les relevés et les données qualitatives sont prises. Il existe dans la base de données un schéma spécifique, avec des tables pour les zones de prospection et les aires de présence.

La première étape était la création d'un formulaire ODK pour se passer de Geonature. Ce formulaire est le seul créé au parc dans lequel l'utilisateur crée lui-même des éléments géographiques. Les deux géographies ici sont tous les deux de type polygone, qui sont représentés par le type polygone et une série de coordonnées, dont le premier et le dernier qui sont identiques. C'est également le premier formulaire ou la relevance de questions de type choix de nomenclatures dépend de la réponse

à d'autres questions du même type, ce qui est une difficulté en lui-même. Cependant, il n'y a que trois fichiers csv utilisés dans ce formulaire, contrairement aux cinq pour les formulaires de monitoring, les sites n'existant pas et du fait qu'il n'y ait qu'un jeu de données possible, ceci est passé en backend également.

Ce formulaire est sans doute le plus compliqué à créer par le parc jusqu'à présent, étant donné la gestion des géographies et de celle des nomenclatures. De la part des géographies, celles renvoyées par ODK Central sont en format GeoJSON , alors qu'en base elles sont en format WKT. Il faut donc les reformatter à être du bon format, une fonction a été créée à ces fins.

Du côté des nomenclatures, ceux-ci ne seront probablement pas les mêmes selon l'instance de Geonature, le choix doit pouvoir prendre ceci en compte. Le code des nomenclatures est ici passé lorsqu'existe une question dépendant d'une autre question de nomenclatures, et le backend détermine l'id de la nomenclature à passer selon son code et son type.

Un troisième problème ici est que dans Geonature, la zone de prospection est tracée sur la carte où sont créées les aires de présence, mais ceci n'est pas possible en ODK. Un ticket a été déposé pour demander à ODK d'implémenter une fonction pour rectifier ceci, mais ceci seulement fin mai, donc ce n'est pas encore le cas.

Pour pouvoir utiliser odk2gn ici, un nouveau dossier a été créé avec toutes les méthodes spécifiques à ce module. La fonction d'écriture des fichiers csv est une des rares qui a changé, tout comme celle qui récupère les nomenclatures, plus simple ici. Cependant, dans la mesure du possible, les fonctions de monitoring ont été réutilisés, dont tous ceux servant à interagir avec ODK Central, De plus, les fonctions de ligne de commande ne sont plus utilisables car il manque l'existence du code de module. De ce fait, deux nouvelles fonctions de ligne de commande ont été créés, un de type synchronize, un de type upgrade-odk-form. Du fait que ces deux fonctions sont très semblables à ceux pour monitoring, deux groupes de fonctions ont été créés pour une meilleure modularité.

La fonction upgrade-odk-form ici utilise la nouvelle fonction d'écriture des csv, et les méthodes générique de mise en brouillon, remplacement des fichiers, et de publication.

La fonction de synchronisation a nettement plus changé. Il n'est plus question de création de visite et d'observation, mais de zone de prospection (de type TZprospect), et d'aire de présence (TApresence). Pour chaque attribut, celle de l'objet Python est remplie des données fournies par ODK et éventuellement formatée. À la fin de la définition de l'objet, nous essayons de le pousser, et si c'est possible, ceci est fait et nous vérifions que l'objet ne sera pas remis en base, sinon nous expliquons qu'il y a des erreurs dans la soumission comme pour la synchronisation de Monitoring.

2.2.4 Tests

Une autre nouveauté implémentée pendant la première moitié de stage vient de l'introduction de tests unitaires dans ODK2GN. Ces tests n'existaient pas avant, s'il fallait tester une fonction, il fallait créer une soumission dans ODK, puis la tester manuellement, et s'il plus de tests étaient nécessaires, il fallait retourner dans ODK et recréer une soumission. Tous les tests implémentés pour l'instant sont ceux de monitoring, pour toutes les fonctions get, pour l'écriture des fichiers csv et pour la synchronisation. Ce sont ces tests qui ont mené à tout le travail de renforcement de données.

Afin de pouvoir effectuer ces tests, certaines fixtures ont été créés. Dans le cas où nous aurions besoin d'un taxon, nous en avons pris Homo sapiens et nous lui avons créés une liste. Pour les jeux de données, nous avons un petit tableau de faux jeux de données, un faux module aussi simple que possible avec un faux site, et des observateurs. Nous avons également une fausse en-tête et des fausses données pour tester l'écriture des fichiers csv. Enfin, nous avons attribués à ceci la configuration de module la plus générique possible, et également une fausse soumission qui est le plus simple possible.

Pour les tests de tous les fonctions get, nous créons un objet résultat des gets, nous vérifions que c'est bien une liste, nous regardons les colonnes et vérifions que les colonnes en question sont bien les bonnes. Pour tester l'écriture des csv, nous vérifions que le retour de la méthode est bien le tableau à laquelle nous nous attendions.

Le test de synchronize monitoring est la plus compliqués . Ceci n'est pas seulement le cas parce que ce test est long, mais également car il n'est pas évident à mettre en œuvre. Cette fonction demande le plus de mockage, car elle fait le plus de requêtes sur ODK, une plateforme externe à laquelle nous ne voulons pas toucher lors des tests, et demande une certaine manière d'être appelée. L'assertion pour cette fonction est bien que le code de sortie soit bien le 0 qui signifie que tout va bien. La difficulté de cette fonction figurait dans le contexte de l'application dans sa fonction. En effet, un grand nombre de méthodes utilisées dans le cadre de synchronize créent un nouveau contexte à son appel, signifiant que nous perdons ce qui est déjà dans la base de données pour le test, et qu'une tentative de test essaierait d'envoyer des données en base qui violent les contraintes d'intégrité de la base.

3. Conclusion

Ce stage m'a permis d'acquérir des connaissances, des méthodes de travail et des compétences auxquels je n'aurais jamais pensé au début du stage. Une chose parmi cela est l'utilisation de documentation officielle dans le monde de l'informatique. Le début de maîtrise d'ODK réalisé jusqu'à aujourd'hui s'est faite avec la documentation officielle de cet outil. Le stage a également été le moyen pour moi de replonger dans le langage de programmation Python, que je n'avais à peine utilisé depuis ma première année de licence (qui s'est terminée en 2018). L'utilité des ORM est une autre chose que je tire de ce stage, cet outil est très puissant et très pratique pour récupérer ou ajouter en base de données des objets. De plus, j'ai vraiment appris comment me servir de la plateforme de partage de code GitHub. Un dernier élément que je retiendrai de ce stage est la puissance de l'open-source. N'étant pas dans une organisation existante pour les profits, j'ai pu prendre mon temps pour bien comprendre comment fonctionnent les outils que j'utilise, ainsi que l'utilité d'avoir toute une communauté qui développe les mêmes outils que vous est puissant.

Je n'ai pas parlé de toutes les implémentations que j'ai essayé de réaliser encore, car certaines parties que ce soit dans le tutoriel ou dans l'application Geonature ont été assez difficile à implémenter. D'abord dans le tutoriel, il y a une partie sur la traduction dans l'application passant par la ligne de commande. Traduire les phrases

à la main n'est pas difficile pour moi, étant anglophone de naissance, mais je n'ai pas pu mettre ceci en œuvre. C'est une histoire semblable pour la fonctionnalité de recherche dans l'application du tutoriel. De plus, le codage de frontend dans le tutoriel était également une difficulté pour moi. Dans Geonature, dans le module flore prioritaire, pour calculer les altitudes, nous sommes censés utiliser un trigger qui est alimenté par une fonction déjà existante, mais que je n'ai pas pu implémenter alors que le code semblait bon. Enfin, j'ai passé trop de temps pour essayer de faire passer la création de sites de suivi de nard par ODK, alors que ceci n'est pas encore possible pour tout autre module Monitoring. J'ai également rencontré quelques difficultés à trouver les erreurs dans le code.

Dans l'état actuel, nous avons un outil ODK2GN beaucoup plus solide, avec des données plus rigides, plus abouti et avec plus d'utilisations possibles par les modifications que j'ai pu lui faire. Nous avons plus de formulaires plus adaptés. De plus, lors de cette création de formulaires, nous avons pu voir lesquels sont plus adaptés à l'usage futur qui en sera fait, et lesquels ont besoin d'être améliorés. ODK2GN pourrait être adapté à créer des sites pour certains modules où ceci serait intéressant, ceci serait peut-être pour la suite du stage.

Annexes

Bibliographie/Webographie

Tutoriel Flask : https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world

Documentation Geonature : https://docs.geonature.fr/

Documentation Flask : https://flask.palletsprojects.com/en/2.3.x/

Documentation ODK : https://docs.getodk.org/

GitHub module flore prioritaire : https://github.com/PnX-SI/gn_module_flore_prioritaire

GitHub module monitoring : https://github.com/PnX-SI/gn_module_monitoring

GitHub ODK2GN : https://github.com/PnX-SI/odk2gn

GitHub ODK2GN dans son état actuel : https://github.com/Xav18/odk2gn/tree/tests

Annexe 1 : Une partie de l'XLSForm du suivi de chiroptères. Le formulaire dans sa définition actuelle ainsi que celle pour le module flore prioritaire seront joints dans le dossier zip.

		type	name	label	hint
		start	start		
		end	end		
		today	day		
		deviceid	deviceid		
Les champs dont le "name" sont de cette couleur SONT OBLIGATOIRES afin de faciliter l'import dans GeoNature. Toutefois ils peuvent être désactivés du formulaire en renseignant la colonne "appearance" avec la valeur "hidden"					
Page de présentation du protocole avec logo		note	presentation	Titre présentation	**Description du protocole** Emplacement pour décrire succinctement le protocole au besoin. Possibilité d'ajouter des liens vers un siteUn exemple de lien [ICI](https://geonature.fr). Cette application a été développée par une communauté bienveillante issue des PNx et de l'OFB. Avec les contributions de la **communauté ODK** sur [le forum d'ODK](https://forum.getodk.org).
Permet de choisir la méthode de sélection du site à renseigner		select_one list_method_site	method_site_choice	Choix de votre site par liste ou sur une carte	
	1	**begin group**	**visit_point**		
Choix du site d'après une liste avec autocomplétion active à partir de 3 caractères saisis		select_one_from_file gn_sites.csv	site_list	Choisir un site dans la liste	
Choix du site sur carte avec centrage de la carte sur la zone où se situe l'observateur si le GPS est activé		select_one_from_file gn_sites.csv	site_map	Choisir un site sur la carte	
	/1	**end group**			
Identifiant du site		calculate	id_base_site	Ajouter un point d'écoute	
Nom du site		calculate	base_site_name	Ajouter un point d'écoute	
	2	**begin group**	**visit_1**	**Visite**	
Date et heure du début des observations		date time	visit_date_min	Date et heure de début :	
Date et heure de la fin des observations		date time	visit_date_max	Date et heure de fin :	
Boucle pour renseigner le(s) observateur(s)	**3**	**begin repeat**	**observers**	**Observat.eur.rice.s**	
Identifiant de l'observateur issu d'un choix sur liste avec autocomplétion à partir de 3 caractères saisis		select_one_from_file gn_observateurs.csv	id_role	Observat.eur.rice.s	
	/3	**end repeat**			
	3.1	**begin repeat**	**observers_txt**	**Observat.eur.rice.s extérieurs**	
Choix d'un observateur externe		text	observers_ext	Observat.eur.rice.s extérieurs	
	/3.1	**end repeat**			
	/2	**end group**			
		integer	sheet_id	sheet_id	
Import d'une photo prise avec l'appareil du téléphone ou d'une image stockée dans le téléphone propre à la visite		image	medias_visit	Photo de la visite	
	4	**begin group**	**dataset**		
		select_one_from_file gn_jdds.csv	id_dataset	Choisir jeu de données	
		select_multiple guano	guano_presency	Cocher si présent	
		text	guano_comment	Renseignement sur le guano présent (taille, fraîcheur, qualité)	
Commentaires propre à la visite Champ multiligne		text	comments_visit	Commentaire (autres indices, présence d'autres espèces, etc..)	
	/4	**end group**			
présence de chiroptères		select_multiple chiro	no_data	Cocher si présence de chiroptères	
Boucle pour renseigner l(es) observation(s)	**8**	**begin repeat**	**observations**	**Espèce(s) observées**	
	9	**begin group**	**sp_choice**	**Choisir l'espèce**	
Choix de l'espèce d'après une liste avec autocomplétion active à partir de 3 caractères saisis		select_one_from_file gn_taxons.csv	cd_nom	Sélectionnez l'espèce (auto) :	
	/4	**end group**			
Filtrage de gn nomenclature pour avoir les différents choix		select_one_from_file gn_nomenclatures.csv	id_nomenclature_behaviour	Comportement	
Filtrage de gn nomenclature pour avoir les différents choix		select_one_from_file gn_nomenclatures.csv	id_nomenclature_bio_condition	Etat biologique	
Filtrage de gn nomenclature pour avoir les différents choix		select_one_from_file gn_nomenclatures.csv	id_nomenclature_bio_status	Statut biologique	
Filtrage de gn nomenclature pour avoir les différents choix		select_one_from_file gn_nomenclatures.csv	id_nomenclature_obs_technique	Technique d'observation	
Filtrage de gn nomenclature pour avoir les différents choix		select_one_from_file gn_nomenclatures.csv	id_nomenclature_life_stage	Stade de vie	
Filtrage de gn nomenclature pour avoir les différents choix		select_one_from_file gn_nomenclatures.csv	id_nomenclature_sex	Sexe	
Nom de l'espèce		calculate	sp_label		
	10	**begin group**	**sp_nbre**	**Comptage**	
		integer	count_min	Dénombrement (minimum)	
		integer	count_max	Dénombrement (maximum)	
Commentaires propres à l'observation Champ multiligne		text	comments_observation	Commentaire(s)	
	/10	**end group**			
Import d'une photo prise avec l'appareil du téléphone ou d'une image stockée dans le téléphone propre à chaque observation		image	medias_observation	Veuillez prendre une photo de l'observation	
	/8	**end repeat**			