**Projet SaaS CVM 2022**

**CVMBIZ.inc**

**But général du projet**

Produire un logiciel de gestion de projet avec une approche SaaS (Software as a Service), partant d’une application fonctionnelle minimale.

Ce type de logiciel se divise en deux sections.

L’**application de service**, résidant sur la l’ordinateur client, séparée en modules distincts fournissant des services précis. Ces modules sont pour desservir des « locateurs ».

L’**application de gestion SaaS**, résidant sur le serveur, permettant de gérer les demandes des locateurs : inscription initiale, gestion des usagers et de leurs permissions, obtention des modules spécialisés (gestion des projet, facturation, contrat, etc).

**Contexte : Gestion Communication – GestMedia**

Les services du SaaS sont offerts pour les organisations œuvrant dans des projets axés sur des mandats de production de matériels multimédia (spot et/ou matériel publicitaire, matériel didactique, etc), et/ou des mandats de gestion d’événements (campagne publicitaire, événement culturel, etc).

**Opérations**

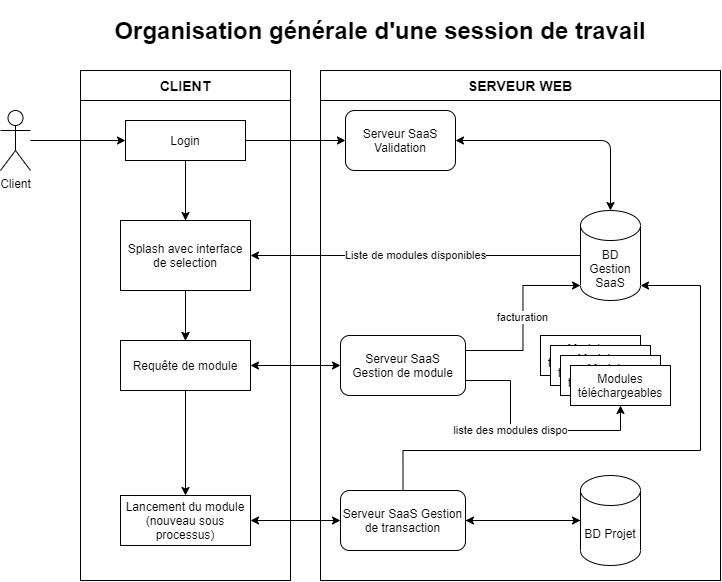
Les “locateurs” utilisent l’Application

1. En inscrivant leur organisation auprès du serveur (à faire)
2. En se loggant au serveur avec identifiant et mot de passe (à compléter)
3. Le locateur-admin détermine des rôles et leur permission d’Accès à un sous ensemble de modules du SaaS
4. Le locateur-admin inscrit les personnes de son organisation ayant identifiant, mot de passe et rôle
5. Les locateurs téléchargent et utilisent les modules au besoin, les données étant envoyées au serveur pour sauvegarde dans la base de données du SaaS

Le “développeur-fournisseur” du SaaS

1. Développe les divers modules offrant une gamme de services pour les compagnies oeuvrant en multimédia
2. Gère les données de gestion des locateurs (organisations, roles et permissions, gestion des logins, etc)
3. Transige les données des locateurs dans une base de données résidant sur le serveur
4. Inscrit ces transactions pour fin de facturation

**Vision**



**Intérêts des logiciels SaaS**

**Pour le fournisseur**

* Modalité de tarification flexible, au choix du fournisseur
  + par bloc d’utilisateur
  + par utilisateur
  + par stockage de données
  + par transaction
  + Abonnement mensuel
  + Fremium
    - etc
* Distribution dynamique des applications
  + L'utilisateur peut requérir uniquement certains des modules, à sa convenance
    - qui sont transférés au client à partir du serveur pour être exécuté localement
      * Ces modules peuvent être conservés “en cache” afin d’éviter des téléchargements inutiles
        + Ceci dépendra de la stratégie de tarification et de mise à jour des modules par le fournisseur
      * Les modules peuvent être effacés à la fin de leur utilisation
* Développement progressif
  + Le fournisseur peut rendre disponibles des modules supplémentaires de manière progressive fournissant une offre supplémentaire de services
    - Il faut néanmoins bien faire l’arrimage avec les informations à gérer provenant parfois d’autres modules
* Mise à jour constante
  + La mise à jour des modules peut être effectués de manière similaire, avec la possibilité d’accepter ou non ces mises à jour
    - Encore une fois cela dépendra de la tarification mise en place
* Gestion des licences
  + droit des utilisateurs et leur inscription au service
    - La comptabilisation peut aussi se faire de diverses manières
      * par modules
      * par transaction
  + ce qui requièrent minimalement un administrateur principal
  + ce qui requiert un système de d’identification
* Inscription initiale du client
  + Historique de ses usages et performances sur la plateforme (info-nuagique)
* Relative indépendance des technologies
  + Le serveur de BD peut changer de plateforme sans modifier l’usage par le client (passer de SQLite à Oracle, par exemple)

**Pour le client**

* Utilise seulement les modules qui lui sont utiles
  + et en conséquence ne débourse que pour une partie de l’offre
* Le logiciel reste à jour sans problème pour ses gestionnaires de technologie interne
* Le client peut gérer les rôles de ses employés et fournir les droits d’accès aux modules requis
  + Ainsi il peut encore mieux découper les coûts d’exploitation de la plateforme.

**Configuration**

* Le client pourra gérer ses usagers et les services dont ils disposeront
* Certaines préférences pourraient être offertes au client pour personnaliser l’usage des modules
  + Mise à jour automatique ou sur-demande
  + Ajout et retrait de services
* Personnalisation de certains modules et des interfaces attenantes
  + Il faut remarquer que les fonctions de personnalisation de ce genre peuvent avoir un impact sur les mises à jour prévu par le fournisseur

**Point de départ**

Le système est déjà partiellement constitué suivant le principe de distribution logiciel de l’approche SaaS, dans une infrastructure client/serveur. L’approche retenue ici est un peu hybride, en ce que le traitement ne se fait pas dans l’infrastructure nuagique du fournisseur mais bien sur la station de travail du locateur.

Si les données sont préservées sur les infrastructures du fournisseur, les modules de traitement, eux, sont obtenus à partir de l’infrastructure du fournisseur et téléchargés au locateur-client pour effectuer localement le traitement avant de renvoyer les résultats à la base de données client sur le serveur.

**Détail**

* L’application cliente de base comprend
  + l’interface d'identification (login)
  + L’interface d’inscription initiale du client (si le client n’est pas encore connu par le fournisseur)
    - Cette interface doit aussi permettre de choisir les fonctionnalités initiales que le fournisseur leur offrira
    - Exemple
      * Un module d’identification de client
      * avec un module de prise en mandat
  + L’interface d’entrée au logiciel, permettant de choisir l’application (module de service) désirée
    - Dans l’exemple précédent
      * Module Client
      * Module Mandat
    - **NOTE**
      * Dans l’exemple précédent, le module pourrait requérir que le client existe déjà dans la base de données Client, ou que le mandat puisse permettre d’inscrire un client et soit le garder localement avec l’information du mandat, soit d’inscrire, à partir du mandat un fiche pour ce client
* Modules de service (applicatifs pour un sujet spécifique)
* Outils de développement (pour configurer le logiciel lui-même)
  + Gestion interne (license, rôles, modules, etc)

Exemple d’utilisation

* Le client d’origine, avec des droits d’administration, se connecte au serveur SaaS
  + C’est à l'administrateur de déterminer quels rôles obtiennent le droit à quel service
  + L'administrateur peut aussi retirer ces permissions
* l’utilisateur doit
  + s’identifier dans un premier temps pour que le système connaisse son rôle
  + à partir de l’interface d’entrée il choisit quel module il veut utiliser
* Le serveur indique au client dans l’interface initiale quels sont les modules qu’il peut utiliser
* Le client requiert un module d’application
* Le serveur retourne les fichiers pertinents et les informations de connections requises, notamment concernant l’emplacement des BDs
* L’application demandée par le client démarre, se connecte à sa base de données et affiche l’interface de travail au client.
* Lorsque le client termine l’usage de l’application celle-ci est supprimée du client. (voir NOTES ci-bas)
* Les données sont préservées dans les bases de données “info-nuagiques”.

NOTES

L’application pourrait être préservée en “cache”, et n’être remplacée que par une version plus récente, évitant ainsi des téléchargements inutiles

* 1. Cette option pourrait être un choix de configuration du client
  2. Cette approche requiert cependant un système de vérification de numéros de version
  3. Une autre option pourrait permettre au client d’utiliser une version spécifique d’un logiciel même si de nouvelles versions sont disponibles, ce qui nécessiterait l’archivage des versions…

Les BDs peuvent ainsi voyager sur les serveurs physiques différents, identifiables par leur adresse IP sans requérir de changement du client.

Les transaction/communications entre le client et le serveur se font en xmlrpc (note

**État actuel et réalisations à poursuivre**

Le logiciel client ainsi que le serveur SaaS, permettant de retourner les services disponibles, et de répondre aux requêtes d’applications demandées est actuellement implanté.

**Client (partiellement implanté)**

* Notre logiciel client qui se connectera au serveur SaaS
* la connexion initiale requiert une adresse IP et un nom d'usager pour l'instant
* sur connexion, les applications disponibles sont présentées (sql et projet pour l'instant)
* la requête de service verra le transfert de l’application vers le client ainsi que le démarrage de l’application s’effectuer.

**IMPORTANT**: les applications sont démarrées dans leur propre processus en mémoire de l’ordi, et n’ont pas de connexion avec le client - seulement avec le serveur, avec qui ils doivent établir cette connexion à partir des infos fournies par le client lors du lancement de l’application.

C’est ce logiciel-client qui conservera les informations de sessions de travail local: par exemple, l’usager actuel, le projet avec lequel travailler, les modules ouverts lors de cette session, etc

**Serveur**

Le serveur sert de porte d'entrée aux requêtes de services.

Il fournit au client la liste des modules accessibles lors de la connexion.

Il fournit le module requis afin qu'il soit accessible au client

Les modules fournis vont se connecter

**Fonctionnement présent du code d’amorçage**

Les modules proposés sont inscrit dans le code du serveur: voir self.modulesdisponibles de la classe ModeleService

Le serveur retourne la liste des clés de ce dictionnaire

Les entrées de ce dictionnaire donnent le nom d’un dossier dans le dossier gp\_modules, qui se trouve au même endroit que GestMedia\_serveur.py.

NOTE: À changer pour la lecture du dossier de modules disponibles

**Procédure de lancement d’un module**

* Le client identifie un module dans la liste qui s’offre dans la fenêtre affichée.
* Il requiert le module
* Le serveur analyse le nom dans la fonction requetemodule
* Si le nom de module transmis correspond à une des clés du dictionnaire modulesdisponibles
  + Il trouve le dossier courant (d’exécution)
  + Si il existe un dossier “/gp\_modules/” dans le dossier courant
    - Si il y existe un dossier du nom du module recherché, tel que retourné par le dictionnaire modulesdisponibles (RAPPEL: le nom de module transmis correspond aux clés de ce dictionnaire), donc via la clé, on obtient le nom réel du dossier contenant tous les fichiers composant ce module.

**NOTE** cette indirection permet jouer avec la structure interne sans influencer ce que le client voit.

* + - * Pour tous les fichiers et dossiers contenu dans le dossier du module, on les identifie soit comme fichier soit comme dossier, on ajoute cet identifiant, et le nom du fichier/dossier en une liste de 2 élément (ex: [“fichier”,”gp\_projet.py”])
      * On ajoute cette paire à la liste “listefichiers”
      * On retourne cette liste au clients
* Le client reçoit la liste des fichiers correspondants au module requis.
* Si la reponse est positive (on a reçu une liste de fichiers)
  + On crée un dossier pour ce module
  + Pour chaque paire correspondant à “fichier”, le client requiert spécifiquement ce fichier au serveur
    - Le client ouvre un fichier en mode “écriture binaire”
    - Le client écrit la section “data” de la réponse dans ce fichier
    - Le client ferme ce fichier
  + On lance un process Python avec Popen qui veut démarrer un fichier du nom du module plus l’extension ”.py”

**Problèmes divers**

* Les modules ne consistent qu’en fichiers pour l’instant, on ne fait pas le travail récursif pour regarder à travers tous les sous-dossiers et obtenir ces contenus.Le login est automatique - aucune vérification du coté serveur
* Il écrase silencieusement une installation déjà existante d’un module
* Le serveur devrait lire le contenu de son dossier modules pour savoir ce qui est disponible et non qu’on soit obligé de modifier le code du serveur…
* Aussi l’idée d’un nom court couplé d’un préfixe complique peut-être inutilement le travail d’identification et de repérage des modules demandés
* Les applications modulaires “projet” et “sql” sont seulement des squelettes d’applications
  + Dans ce sens, l’application possède plusieurs recettes mais pas vraiment de fonctionnalité
* Il n’y a encore aucune base de données de connectés
* Comme chaque module est indépendant de l’existence des autres, ils doivent être autonome, c’est à dire aptes à fonctionner seul - donc les éléments comme les lisez-moi.txt, l’aide enligne, etc doit être inclus avec chaque module.
* Comme mentionné dans le texte, il faut choisir une approche concernant les mises à jour: elles peuvent être automatique, ou au choix, avec contraintes ou non (genre, si vous faites cette mise à jour, les documents ne seront plus accessibles aux anciennes versions… Ensuite, il faut systématiquement appliquer cette approche pour les modules.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Addendum**

Gestion agile (voir chiffrier)

***Analyse, modélisation et planification globale***

***Mandat***

Module servant à inscrire le mandat: plein texte (importable/exportable) et grille d'analyse textuelle

Cette étape consiste à l'analyse du document texte qui représente le mandat fournit par le client. Il s'agit d'analyser le texte écrit en langage ordinaire, pour en faire ressortir les éléments prépondérants qui devraient correspondre à des parties du logiciel escompté. L'analyse consiste à répertorier les mots essentiels et éliminer le superflu, du point de vue de ce que le logiciel devrait pouvoir faire comme tâche. Ces mots sont répartis dans un tableau à deux axes, en colonnes et rangées.

**Le tableau d’identification des mots importants**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Nom | Verbe | Adjectif |
| explicite |  |  |  |
| implicite |  |  |  |
| suppl. |  |  |  |

**Attention** on n’a pas besoin de faire des phrases, justes inscrire les mots dans leur colonnes et rangées respectives

***Les colonnes***

Les **noms** représentent des objets du domaine, c'est à dire les choses qui font ou sur lesquelles portent les informations à traiter.

Les **verbes** représentent les actions qui transforment l'information se rapportant aux choses du domaine.

Les **adjectifs** qualifient les choses et les transformations d'informations.

***Les rangées***

Ce qui est **explicite**: les mots tels qu'on les retrouvent dans le texte - ils représentent l'expression direct de la commande du mandataire

Ce qui est **implicite**: parfois, ce qui est requis explicitement assumera l'existence d'actions, de choses qui ne sont pas mentionnées mais qui s'avèrent requis pour effectuer certaines actions.

Ce qui est **supplémentaire**: représente diverses possibilité d'actions (donc de traitement par l'ordinateur) qui n'ont pas été envisagées par le mandataire.

Par exemple, un informaticien, grâce à son expérience, peut interpréter les données pour fournir des informations nouvelles qui peuvent être intéressantes mais qui n'ont pas été demandées parce que les traitements non informatisés en seraient trop onéreux, alors que cela devient (relativement) simple de le faire grâce à l'ordinateur.

**Cas d'usage**

Les cas d'usage représentent la liste des opérations qu'un utilisateur du logiciel peut vouloir faire, chaque opération représentant un tout ayant un début, une suite d'étapes, et une conclusion, c'est à dire un point où on peut dire que l'opération fût effectuée, que le traitement demandé est complet.

**Les rôles**

Diverses personnalités “virtuelles” ayant des accès et/ou permissions différentes eu égard aux cas d’usage, à savoir qui peut utiliser quoi. Employé de bureau, soutien administratif, adjoint, p.d.g. …, s’il y a disparité d’usage, alors il faudra voir l’importance de la connection usager-machine (le login :-) )

**Scénarios d'utilisation**

Les scénarios (scénarii, en latin) d'utilisation décrivent l'ensemble des étapes qui se trouvent dans un cas d'usage. Chaque cas d’usage doit donc être modélisé au complet en scénario, incluant autant les étapes faites par l'utilisateur (choisir un item de menu, cliquer sur un bouton, insérer du texte, etc) que les actions menées par l'ordinateur (afficher une page-écran, calculer un résultat, lire un fichier, etc). Il s'agit d'un enchaînement de fonctions menant à la réalisation du cas d'usage. On ne rentre pas dans les fonctions individuelles, il n'y a pas de pseudo-code encore à ce niveau (on pourrait voir ça comme des appels à de pseudo-fonctions)...

***Liste séquentielle des tâches impliquées dans un cas d’usage***

|  |  |  |
| --- | --- | --- |
| Cas d’usage : Lancement avec login | | |
| Humain | Station (ordi local) | serveur |
| Démarre l’application client |  |  |
|  | Initialise le programme en mémoire |  |
|  | Affiche la page de connection, incluant champ nom et mot de passe et IP du serveur |  |
| Inscris son nom |  |  |
| Inscrit son mot de passe |  |  |
| Inscrit le IP |  |  |
| Demande la connection (bouton) |  |  |
|  | Lecture des champs |  |
|  | Validation d’info |  |
|  | Requete au serveur si ok |  |
|  | … | Verification par le serveur ... |

etc, etc

**Maquettes**

Les maquettes d'interface sont des représentations graphiques des pages-écran du logiciel à développer. On peut utiliser les cas d'usage et/ou les scénarios d'utilisation comme point de départ pour regrouper les objets d'interface (champs texte, boutons, menus, etc) qui servent leur exécution.

Dans un deuxième passage on modifie ces maquettes afin qu'elles reflètent mieux notre compréhension globale du système par restructuration et réagencement.

Exemple 1 : deux fenêtres qui ne se distinguent que par les étiquettes des champs textes, pourraient se voir regroupé en une seule qui aurait alors un état d'où les étiquettes seraient modifiées et les fonctions assignées aux boutons pointeraient sur d'autres fonctions de traitement.

Exemple 2 : deux fenêtres (A et B) possèdent 50% d'objets d'interface identiques, mais 50% d'objets différents - au lieu de deux fenêtres, on pourrait alors prévoir une fenêtre avec une section toujours présente (celle comportant les objets partagés), et deux onglets, l'un affichant les 50% d'objets de la fenêtre A, l'autre ceux de la fenêtre B.

**Modèle de données**

Définition des tables de base de données, c'est à dire les fichiers contenants les données accessibles par le programme.

Généralement construit sous la forme d'un ou de plusieurs tables à deux dimensions, chaque table conservera les données de plusieurs instances d'entités d'un même type appartenant au domaine d'application.

La définition d'une table se fait en indiquant une suite d'attributs dont on spécifiera le type (texte, entier, réel, etc) des données qu'on s'attend à recevoir comme information.

L'utilisation d'une table se fait en insérant une ligne de données, représentant les valeurs qu'une entité particulière possède pour les attributs définis dans la définition de table. Chaque ligne représente une et une seule entité. On peut considérer que le mot entité représente essentiellement la même chose qu'un objet en Programmation Orientée Objet, sans les fonctions.

D'autres part, les tables peuvent être reliées entre elles afin de fournir un mécanisme permettant de fournir une liste de valeurs pour un attribut particulier d'une entité. Par exemple, un client peut avoir plusieurs transactions avec un fournisseur et il serait difficile de manipuler un attribut "transaction" qui contiendrait toutes les transactions de ce client. Plutôt, on entend alors qu'une transaction est un sorte d'entité qui aura sa propre table, où les données spécifiques à chaque transaction seront notées et dont un attribut particulier permettra de connecter une transaction spécifique à un client spécifique.

Exemple de modèle de table

|  |  |  |  |
| --- | --- | --- | --- |
|  | type | cle | nulle |
| nom |  |  |  |
| prenom |  |  |  |
| adresse |  |  |  |
| courriel |  |  |  |

Etc, etc

**CRC (Définition des classes)**

Les cartes CRC représentent une manière de définir les classes dont fera usage notre application. Chaque carte comporte trois sections (C+R+C)

Classe : représente le nom de la classe, et inclus aussi le nom du propriétaire de cette classe dans l'équipe de programmation. Ce propriétaire n'est pas tenu de programmer tout ou même une partie de cette classe mais il est l'arbitre du dernier recours la concernant ainsi que celui qui sera imputable de sa réalisation.

Responsabilité : ce dont cette classe est responsable en terme de traitement, d'action ayant une participation aux scénarios d'utilisation. Il est possible qu'une classe dispose aussi d'autres méthodes, à usage interne en quelque sorte, c'est à dire utiliser par les méthodes principales...

Collaboration : Ensemble des autres classes avec qui les objets de notre classe pourraient être amené à communiquer afin de fournir des services, ou d'en recevoir. Cette liste permet de savoir avec qui dans l'équipe nous devrons discuter le cas échéant, pour que les objets aient une bonne communication (se passent les bonnes données, leur type, etc).

Exemple d’info à établir sur l’architecture des classes qui rempliront les exigences de traitement identifiées dans les scénarios d’utilisation

|  |  |  |
| --- | --- | --- |
| Classe: (propriétaire): | | Collaboration |
| Responsabilité  fonction | paramètre |

**Planification globale**

**Time-boxing**

Une stratégie particulièrement efficace pour réaliser un logiciel est l'approche par "time-boxing", qui limite les délais de manière absolue. Time-boxing veut dire un boîte de temps, à savoir que comme une boîte à une extrémité, elle a aussi une longueur définie et stricte.

Si votre objet est plus grand que la boîte, il n'entrera pas dans la boîte, à moins de pouvoir couper votre objet et ainsi mettre les morceaux côte-à-côtes.

Dans un projet 'time-boxé', si les tâches débordent des délais, il faut trouver une façon de les effectuer en parallèle, à l'intérieur de la même boîte de temps. Ou il faut couper les bouts qui dépassent... Alors si on doit couper, il est préférable de couper là où ça fait le moins mal - c'est à dire couper les fonctions les moins utiles, nécessaires, intéressantes du projet logiciel en question.

**Plus utile en premier, moins utile en dernier**

Comme nous l'avons mentionné plus haut, pour respecter l'échéancier, certaines tâches pourraient se voir supprimer en fin de course. S'il s'agit des fonctionnalités moins utiles, l'échec de réalisation sera moins cuisant.

La question est donc de savoir distinguer ce qui est "plus utile" de ce qu'il l'est moins. L'utilité est un concept nuancé: s'agit-il de ce qui est important, de ce qui est nécessaire, de ce qui est requis, de ce qui est préférable ou préféré? Est-ce qu'une fonction centrale très intéressante est plus utile que la fonction qui la nourrit de données?

**Séquence et parallélisation**

Sans faire de répartition des tâches à ce stade, il est néanmoins précieux de pouvoir identifier les séquences de développement. Une séquence de développement représente la suite des fonctions à développer tel que pour pouvoir utiliser une fonction celle qui précède est requise.

Voir les diverses séquences est une aide précieuse pour déterminer l'ordre d'exécution des tâches mais aussi pour permettre l'organisation d'une équipe de travail, en fournissant des séquences à des sous-groupes de programmeurs.

\*\*\*\*\*\*\*\*\*\*\*\*

**La gestion de l’implantation**

**Définition des SPRINTs**

Les SPRINTs sont des périodes prévues de développement axées sur les livrables à rendre en fin de SPRINT.

Ils sont normalement divisé en 3 sections

**Planification détaillée**

Cette section sert à déterminer les attributions de tâches et de responsabilités aux membres de l’équipes. C’est la responsabilité de chaque membre de pouvoir indiquer combien de temps il compte allouer à la réalisation d’une tâche.

Il faut également être attentif à l’ordre des tâches, certaines requérant possiblement l’achèvement d’autres tâches… ce qui exclut un parfait parallélisme.

Il est évident que cette section bénéficierait d’une interface permettant la création de graphique de type PERT ou Gantt pour aider à visualiser, ou encore le modèle Kanban (un peu à la Trello)

**Gestion des SCRUMs**

Pour chaque période retenue (bloc de cours, jour, etc) , obtenir les réponses des membres concernant leurs réalisations, les tâches en cours et les problèmes rencontrés. Rien comme un liste de case à cocher de tâches pour se mettre en train aller à l’essentiel.

**Debriefing**

Cette section sert, à la fin du Sprint, à remettre les pendules du projet à l’heure concernant les prévisions et les réalisations effectives (livrables fonctionnels pour le client). Des décisions sont prises pour transférer les tâches incomplètes vers le prochain Sprint, ou les abroger si la situation s’avère moins nécessaire voir intéressante à poursuivre.

\*\*\*\*\*\*\*\*\*\*\*\*\*

**Environnement**

**Développement**

Python, flask

SQLite (SGBD)

Eclipse (bon pour debogguer)

**Gestion**

*Suivant vos préférences d’équipe*

GIT, Google Apps, OneDrive apps

***Important*** - ajoutez moi comme membre de vos équipes, de préférence avec des droits :-)