MEMOIRE D’UNE APPLICATION



ANGLADE JESSICA

Sommaire

Prologue : Le projet et sa créatrice

Chapitre I : Le cahier des charges ou l’analyse du projet

Chapitre II : Les spécifications fonctionnelles

Sous-partie 1 : Le cas d'utilisation

Sous-partie 2 : Le diagramme d'activité

Sous-partie 3 : Le diagramme de séquence

Sous-partie 4 : Le maquettage

Chapitre III : La conception d'un Système d'Information

Sous-partie 1 : Le MCD

Sous-partie 2 : Le MLD

Sous-partie 3 : Les requêtes SQL

Chapitre IV : L’arborescence

Chapitre V : Les outils techniques

Chapitre VI : Les fonctionnalités

Epilogue : La Conclusion

Annexe

**Prologue : The project and his creator**

*Once upon a time a woman who loved stories. She read a lot of books, played video games and watched movies and TV series. One day, she invented a story. It was a simple and clumsy story, but she enjoyed it very much. She continued and wrote more. Over time, she created many characters. But every creation, she found it more and more difficult not to forget some details or even not to mix everything up, so she used note cards where she wrote down information about each of her characters: their name, their appearance, their past, ... After a while, it was so hard to find each character in all her note cards.*

*While changing career for health reasons, she followed a training where she learned how to develop android applications. The idea of creating a database application to store the note cards was born. But why limit only to characters ? She decided to extend her note cards to other categories: universe, item, faune, place... Moreover, this would allow a better organization.*

*And then her application could be used by other people like her, passionate about writing. Yes, she would make an application for writers of all kinds, professional or hobbyist. They will be able to create and modify their note card at any time in any place thanks to their cell phone. Because imagination can come anywhere at any time.*

*This woman is me and I will tell you about the creation of my application: Assis'Tales.*

Chapitre I : Le cahier des charges ou l’analyse du projet

Tout début de projet demande à ce qu’on étudie et analyse ce dernier profondément. J’ai donc d’abord bien affiné l’idée, défini ce que je voulais réellement faire ou ne pas faire, ce dont le projet avait besoin. Pour bien affiner l’idée, j’ai dû étudier le marché actuelle, les cibles que je voulais atteindre. Grâce à tout cela, j’ai pu mettre au point un cahier des charges à suivre. Un cahier des charges est un document à respecter lors de la conception d’un projet et qui est habituellement contractuelle entre un client et son prestataire. Celui sera allégé d’un bon nombre d’informations puisqu’il n’y a pas de client indépendant. Le cahier des charges sera découpé en différents points. Je mettrai en annexe le cahier des charges complet réaliser avec un faux client.

1. Les objectifs

Je vais d’abord commencer par les différents objectifs de l’application :

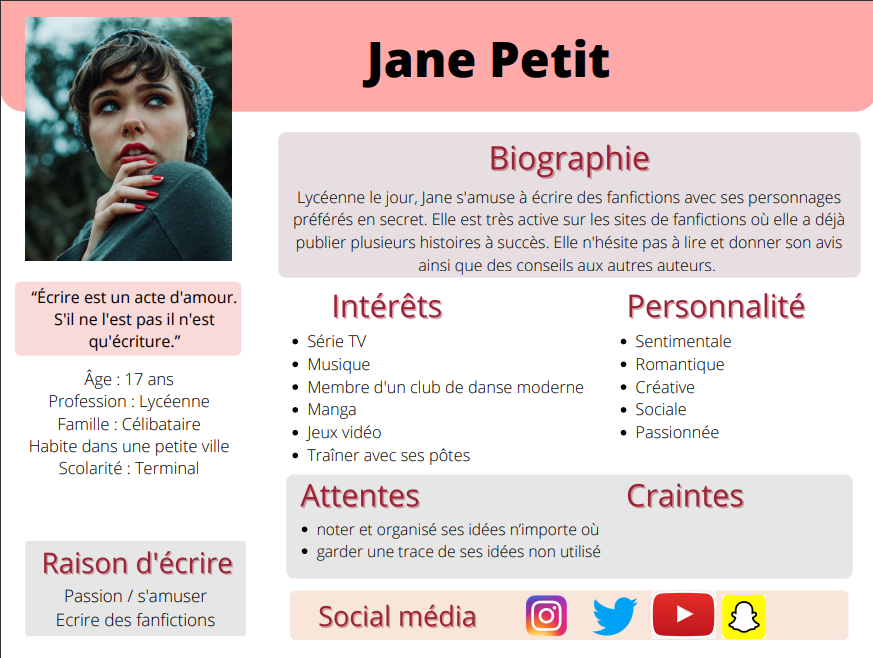
* Elle doit permettre aux auteurs de gérer plus facilement leurs projets d’histoire.
* Elle doit mettre à disposition des outils permettant aux auteurs d’organiser et de structurer leurs récits ainsi que les éléments (personnages, lieux, …) qui les façonnent à l’aide de fiche.
* Elle doit également créer des vocations dans l’écriture que ce soit en amateur ou en professionnel.
* Elle doit enfin stimuler l’imagination en permettant une large possibilité de remplissage de donnée. Elle doit être le moins restrictive possible.

1. Les cibles

Je passe ensuite aux cibles qui pourront utiliser l’application. Pour cela, j’ai créé 5 personas différent. Un persona est, dans le marketing, un personnage imaginaire représentant un groupe ou un segment cible utilisé dans le développement d’un nouveau produit ou service. Elle permet de faire ressortir certaines caractéristiques vers laquelle accès sa politique marketing. Par exemple, une application pour enfants ne sera pas créée de la même manière, que ce soit graphiquement ou en utilisation, qu’une application destinée aux adultes.

Voici donc 2 des personas que j’ai créé :





De cette étude de la cible et des personas, il en est ressorti que :

* Les utilisateurs pourront être des professionnels ou des amateurs
* Les utilisateurs pourront l’utiliser écrire différents types d’ouvrages comme par exemple : des romans, des scénarios, des scripts pour le jeu de rôles, des fanfictions (fiction écrit par des fans ayant lieux soit dans un univers et/ou avec des personnages d’une œuvre existant)
* Les utilisateurs peuvent être de tout âge, avoir n’importe quels métiers, d’en n’importe quelles situations personnelles (célibataire, marié, …).
* Les utilisateurs sont des personnes passionnées par l’écriture.
* Les attentes des utilisateurs sont de pouvoir noter ou consulter leurs idées n’importe où et n’importe quand, de pouvoir s’organiser efficacement et de pouvoir retrouver facilement les informations dont ils ont besoins.

1. Les concurrents

Pour étudier les concurrents que l’application pourrait avoir, je suis allé chercher sur le play store de google. J’y ai trouvé 4 concurrents direct. Les quelques autres sont plus accès sur le scripting de jeux de rôle. Pour pouvoir les analyser efficacement j’ai créé un tableau comparatif :

|  |  |  |  |
| --- | --- | --- | --- |
| Nom | Langues | Points forts | Points faibles |
| Character Story Planner | Anglais | - fiche complète  - 6 différentes catégories  - modification en catégorie dans les fiches  - lecture des fiches pratique | - pas très intuitif  - pas de liens entre toutes les fiches et pas rétroactive |
| Character Story planner 2 | Anglais | - fiche complète  - différentes catégories  - possibilité de télécharger des personnalisations de fiche  -importation/exportation de la base de données  -plusieurs liens entre les fiches  -modification en catégorie dans les fiches  - lecture des fiche pratique | - impossibilité de créer une fiche sans créer un projet  - liens non rétroactive,  - relations personnages dans catégorie à part |
| Writer Tools | Multilingue dont le français | - fiche complète  - notation d’idée  - écrire des chapitres  - un thésaurus  - lecture des fiches pratique  - multilingue | -demande une connexion à un compte  - pas de possibilité de passer à un projet en hors ligne  -sauvegarde en ligne exclusivement  - modification en complète |
| Mystory.today | Anglais | - peut écrire les chapitres  - exportation en format document | - obligation de s’inscrire pour sauvegarder  - création d’un projet pour avoir fiche  - fiche courte |

Ce tableau de comparaison m’a appris plusieurs choses :

* Qu’ils sont presque tous en anglais
* Que certain permet d’écrire des chapitres
* Que la moitié oblige de créer un compte et être en ligne pour enregistrer
* Que les fiches sont plutôt complètes
* Qu’ils n’ont pas tous de lien entre les fiches.

1. Analyse marketing

Pour l’analyse marketing, j’ai réalisé un SWOT qui permet d’identifier les forces et faiblesses du projet ainsi que ses opportunités et les menaces qu’elle pourrait avoir.

|  |  |
| --- | --- |
| **Forces**  - application en français  - plusieurs catégories  - personnalisation des fiches  - lien entre les fiches  - exportation en **format** document  - 2 modes de fiche (simplifié et complète) | **Faiblesses**  - première prise en main complexe  - pas de possibilité de travaillé un projet en équipe  - pas de possibilité d’écrire les chapitres directement  - la catégorie « idées » peut devenir très vite conséquente  - compte de sauvegarde en ligne payant |
| **Opportunités**  -ouverture vers divers domaine d’écriture (scripting jeux de rôle en plateau, fanfiction, roman, scénario,…)  - création d’une communauté  - possibilité d’un partenariat avec des sites de fanfiction, ... | **Menaces**  - cible une catégorie de personnes spécifiques  - désintérêt |

1. Les besoins

Les besoins principaux pour ce projet sont de mettre en place différentes fonctionnalités en ordre d’importance :

* Création des fiches (personnage, lieux, …)
* Possibilité de les intégrer dans un projet ou non
* Possibilité de mettre des liens entre les fiches (relation entre les personnages, personnage habitant dans un lieu, …)
* Possibilité de masquer des infos dans la fiche
* Création de 2 modes de fiches (simplifié et complète)
* Possibilité de créer des fiches personnalisées par défaut
* Création de compte avec sauvegarde en ligne (payant)
* Possibilité d’exportation en format document

1. Les contraintes

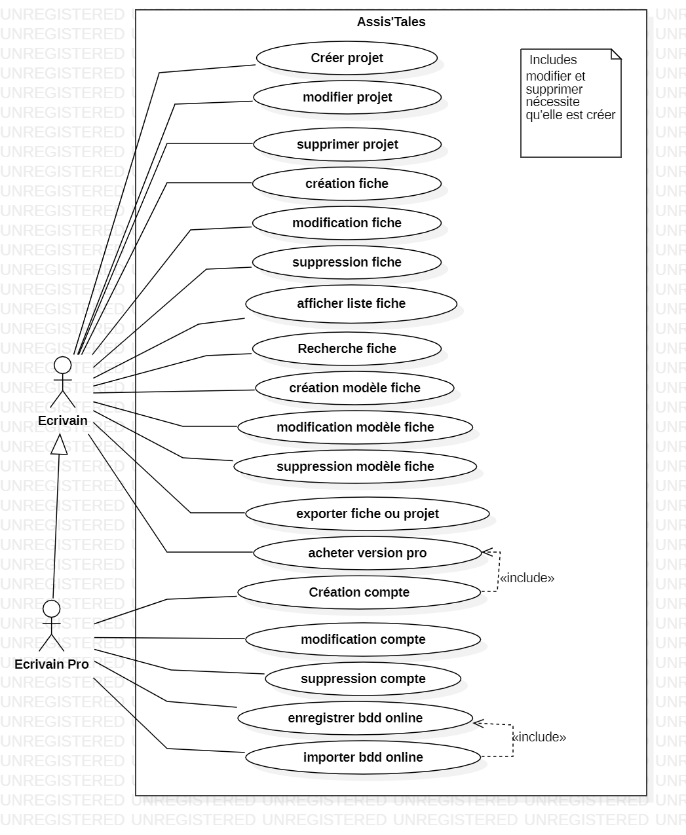
Il n’y a pas de contrainte technique, de coût ou de temps particulier. Les contraintes légales et réglementaire sont celle définit par le gouvernement français.

Chapitre II : Les spécifications fonctionnelles

Après avoir analysé le projet, je dû m’atteler à la conception de l’application. J’ai commencé par les diagrammes UML.

Sous-partie 1 : Le cas d'utilisation

Le premier diagramme que j’ai fait est le cas d’utilisation. Il permet de représenter le comportement fonctionnel de l’application.



Dans un premier temps, j’ai donc listé les différents cas d’utilisation de l’application. Par exemple, il y a la création de projet, la modification de projet et ainsi de suite.

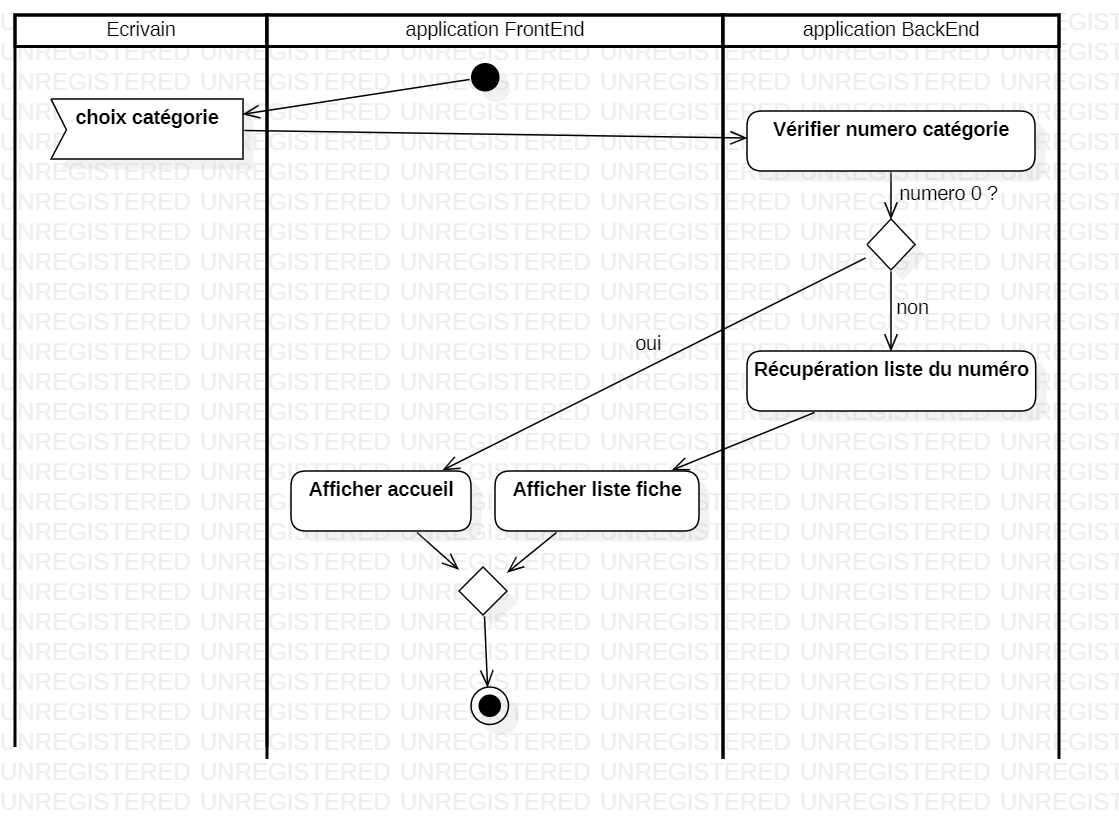
J’ai ensuite défini les différents acteurs, les utilisateurs, de l’application. Ici nous en avons deux que j’ai nommé « écrivain » et « écrivain pro »

Après, j’ai associé à l’acteur « écrivain » les différents cas d’utilisation à laquelle il a accès. L’acteur « écrivain pro » hérite de l’acteur « écrivain », c’est-à-dire qui a accès aux mêmes cas d’utilisation que lui. Je lui ajoute ensuite les cas d’utilisations dont lui seul à accès.

Certaines activités sont liées entre elles. Par exemple, la création d’un compte inclus, « include », que l’activité acheté la version pro ai été faite.

Sous-partie 2 : Le diagramme d'activité

Ensuite je réalise le diagramme d’activité de l’activité afficher la liste des fiches pour une catégorie. Ce diagramme permet de voir le comportement de l’activité.

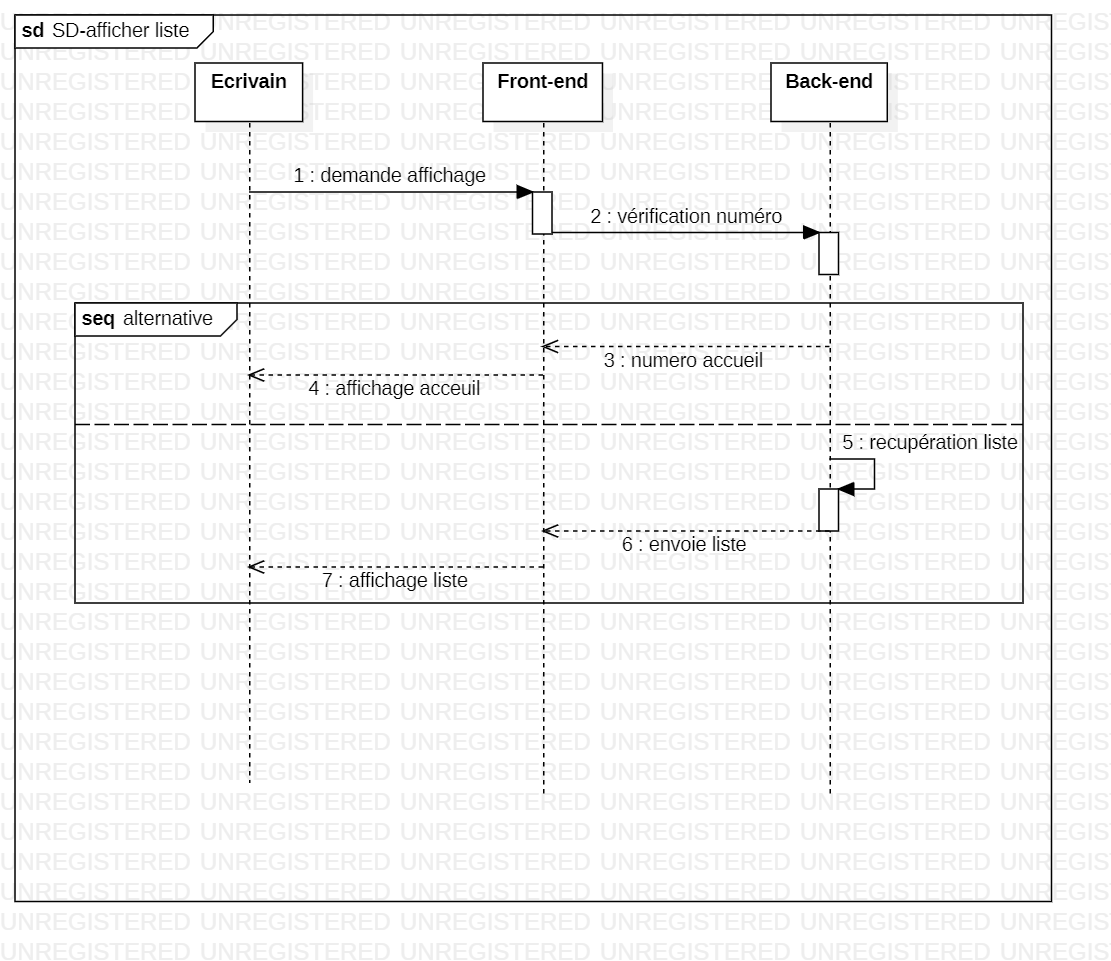


Je créais en premier les acteurs. Il y a tout d’abord l’écrivain, l’utilisateur. Ensuite comme l’application est hors ligne, elle a une base donnée intégrer. Je vais donc différencier la partie Front-end de l’application de la partie Back-end.

Le nœud initial est instancié par la partie font-end. L’acteur « écrivain » choisi une catégorie. La partie back-end vérifie le numéro de la catégorie. J’arrive alors sur un nœud de décision. Si le numéro est 0 alors il affichera l’accueil sinon il va récupérer dans la base de données la liste des fiches du numéro de la catégorie. Il l’enverra alors à la partie front-end pour qu’elle l’affiche. J’ai mis un nœud de merge car que ce soit l’affichage de l’accueil ou de la liste, j’arrive au nœud final.

Sous-partie 3 : Le diagramme de séquence

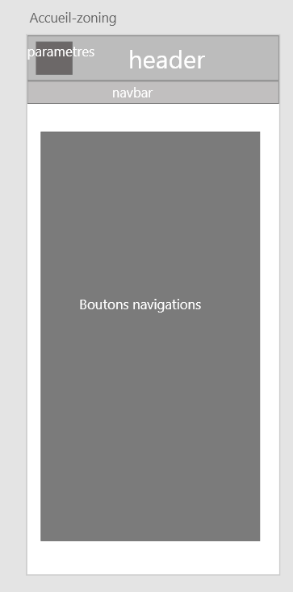
Je poursuis par la réalisation du diagramme de séquence de la même activité. Ce diagramme permet de voir les interactions entre les objets.



Je commence par mettre les acteurs, identique au diagramme d’activité. En premier lieu l’écrivain envoie en message une demande affichage de catégorie au front-end. Celui-ci envoie en message une vérification du numéro. J’arrive alors sur deux possibilités alternatives. Si le numéro correspond à celui de l’accueil, le back-end envoie en message de réponse que c’est l’accueil puis le front-end affiche l’accueil en réponse à l’écrivain. Sinon le back-end récupère la liste de nom de la catégorie et l’envoie en réponse au front-end qui va ensuite l’afficher à l’écrivain.

Sous-partie 4 : Le maquettage

Je passe ensuite sur le maquettage de l’application. Je commence par la page d’accueil.



Tout d’abord le zoning qui montre les blocs apparaissant sur la page. Il y a un header avec un bloc paramétre dessus sur la gauche. En dessous du header, une barre de navigation. Viens ensuite un bloc pour les bouttons de navigation.



Ensuite je réalise le wireframe qui montre un peu plus en détail l’écran. On peut voir que le header est une image avec le nom de l’application en bas à droite. Le bloc paramètre est un icone. La barre de navigation est composer uniquement d’icone. Le bloc de boutons de navigation est composé de plusieurs bouttons avec une icone dessus et son nom.



Enfin je créais le Mockup qui est la version visuel le plus abouti, l’application devra ressembler à ça.

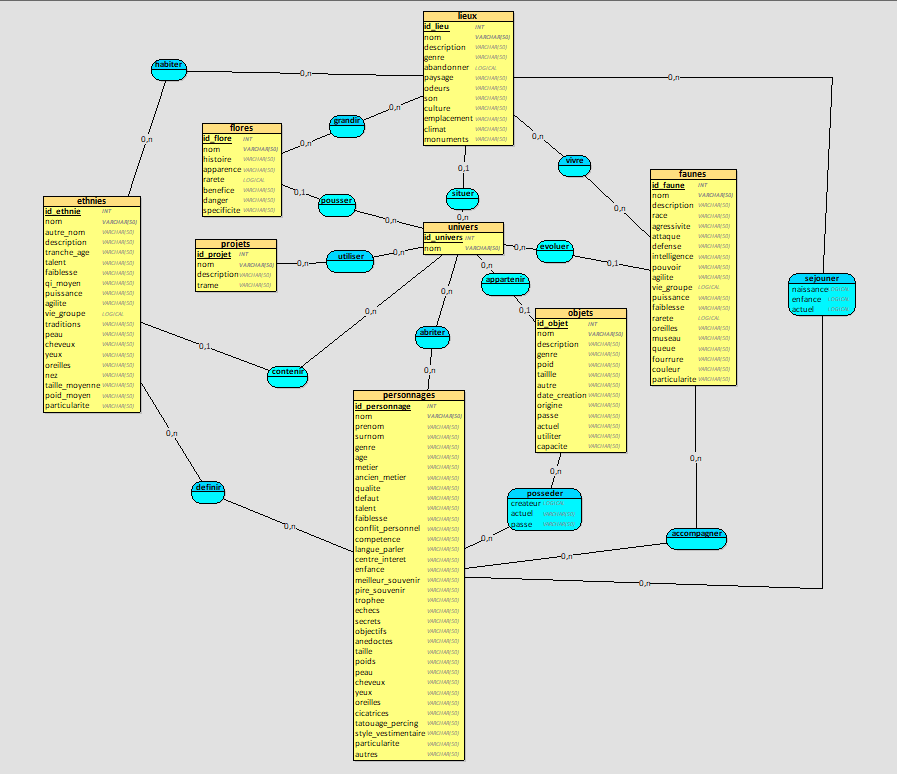


J’en ai fais de même pour la liste des projets. Le header et la barre de navigation est identique à celui de l’accueil. Ensuite, j’ai mis le titre et un bouton ajouter. Enfin je place la liste des noms de fiche qui sera variable en fonction du nombre d’enregistrement dans la base de données.

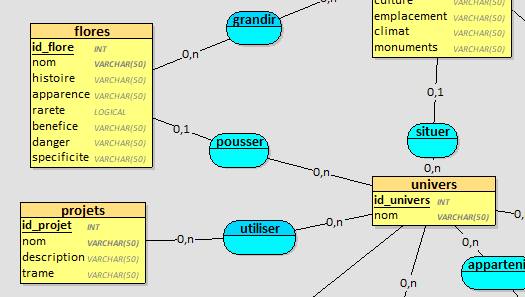
Chapitre III : La conception d'un Système d'Information

Après les diagrammes UML et le maquettage, je construis la structure de la base de données en commençant par le Modèle Conceptuel de Donnée.

Sous-partie 1 : Le MCD



Le MCD étant très grand (8 entités), je ne vais présenter qu’une partie.



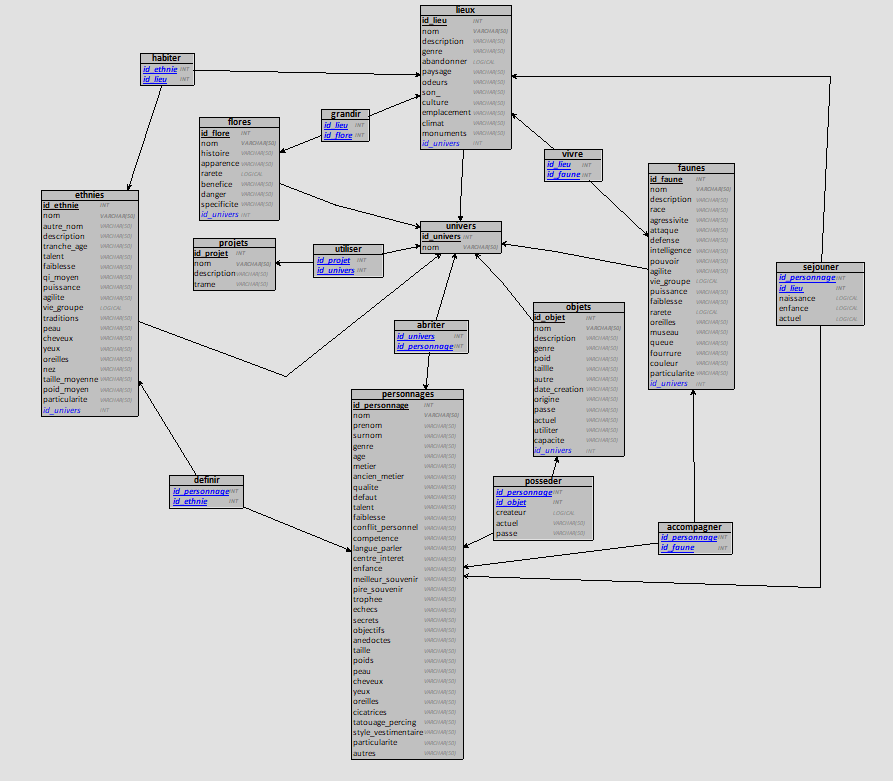
D’abord, je créais les différentes entités. Je leur mets une propriété « id\_nomEntité » en clé primaire. Une clé primaire doit toujours être unique. J’ajoute ensuite les autres propriétés. Après je réalise les différentes associations entre mes entités.

Ici, il y a deux types de relation :

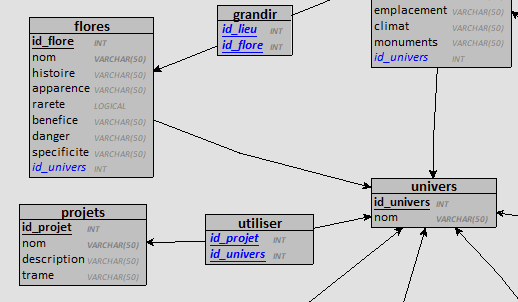
* La première est de type « 1,n », c’est-à-dire que les cardinalités sont de type (0,1) d’un côté et (0,n) de l’autre, entre flores et univers associé par le verbe pousser. Je peux lire les cardinalités de cette façon : « une flore peut pousser dans aucun ou un univers. » et « Un univers peut voir pousser aucun ou plusieurs flores. »
* La deuxième est de type « n,n », c’est-à-dire que les cardinalités sont de type (0,1) d’un côté et (0,n) de l’autre, entre projets et univers associé par le verbes utiliser. Je peux lire les cardinalités de cette façon : « un projet peut utiliser aucun ou plusieurs univers. » et «Un univers peut être utilisé par aucun ou plusieurs projets. »

Comme le MCD est fait, je peux passer au Modèle Logique de Donnée

Sous-partie 2 : Le MLD



Tout comme le MCD, je vais présenter qu’une partie du MLD



Le MLD reprend les mêmes entités/tables et propriétés/attributs que le MCD.

Ensuite pour la relation « 1,n », je rajoute en clé étrangère sur flores la clé primaire de univers, « id-univers ».

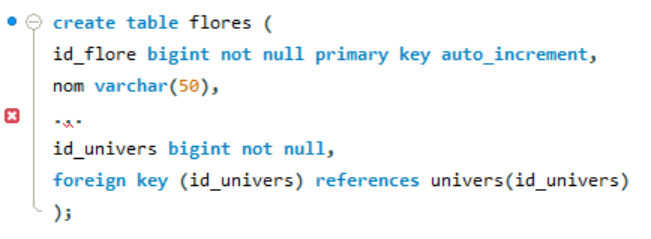
Pour la relation « n,n », je créé la table d’association « utiliser » à laquelle j’ajoute deux clés primaires qui sont également des clés étrangères et qui correspondent aux clés primaires des deux tables liés. Là, il s’agit « id\_projet » et «id\_univers ».

Maintenant que j’ai le MCD et MLD, je peux passer aux requêtes SQL pour créer et gérer la base de données.

Sous-partie 3 : Les requêtes SQL

Même si pour l’application, je vais utiliser principalement une base de données intégré à l’application. J’ai tout de même créé une base de données externe pour une future fonctionnalité qui sera mis en place pour des sauvegardes en ligne.

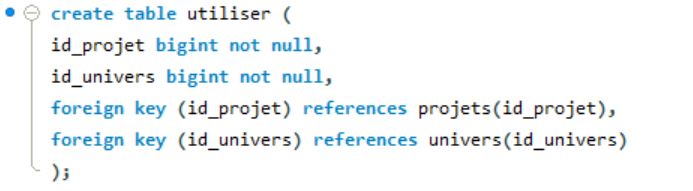
Tout d’abord, je créais la base de données avec . Ensuite, je vais créer les tables. Voici la table flores :



J’ai mis « id\_flore » en clé primaire grâce à « primary key ». Je précise également son type « bigint », qu’elle ne peut pas être null avec « not null » et qu’elle ajoute automatiquement un au précédent avec « auto\_increment ».

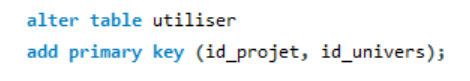
Je créé ensuite les autres champs puis « id\_univers » que j’instancie en clé étrangère ensuite avec « foreign key » qui se réfère à l’id\_univers de la table univers.

Je créais aussi les tables d’associations nécessaires. Voici la table « utiliser » :



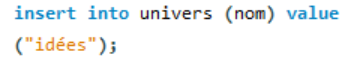
Je mets bien les deux clés primaires des tables liés en clés étrangères. Mais j’ai oublié sur celui-ci de les mettre en clé primaire.

Je vais donc faire un alter table :



Un alter table permet de faire des modifications sur les champs de la table. Là, j’ajoute des clés primaires avec des attributs. Mais je pourrais modifier le nom d’un champ, ajouter ou supprimer un champ, etc.

Pour les tests, je vais créer des enregistrements. En voici un pour la table « univers » :



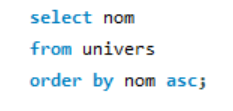
On utilise « insert into » et ajoute le nom de la table et son champ à remplir et continu en précisant la valeur à mettre.

J’ai fait une erreur sur l’un des enregistrements, je vais la modifier avec « update ».



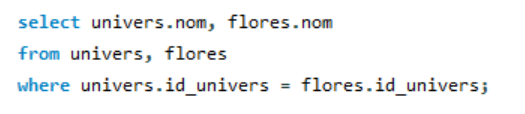
« set » permet de mettre la nouvelle valeur au champ voulu et « where » pour préciser quel enregistrement est à modifier.

Maintenant que j’ai les enregistrements, je vais créer la requête permettant de récupérer tous les noms de la table univers afin de pouvoir les afficher.

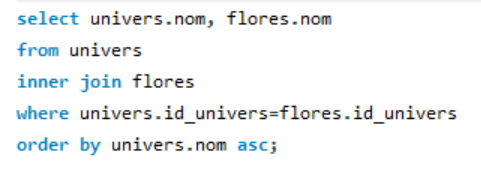


J’ai ajouté « order by » pour qu’il soit rangé dans l’ordre alphabétique.

Je pourrais faire des requêtes plus poussées en utilisant « where » afin de trouver un enregistrement en particulier ou ayant un attribut spécifique. Je pourrais également chercher liens entre les tables. Par exemple, je voudrais connaitre les flores liées aux univers j’utiliserais cette requête :



Je pourrais aussi faire cette requête d’une manière différente en utilisant une jointure grâce à « inner join », j’y ajoute également de le ranger par ordre alphabétique du nom de l’univers :

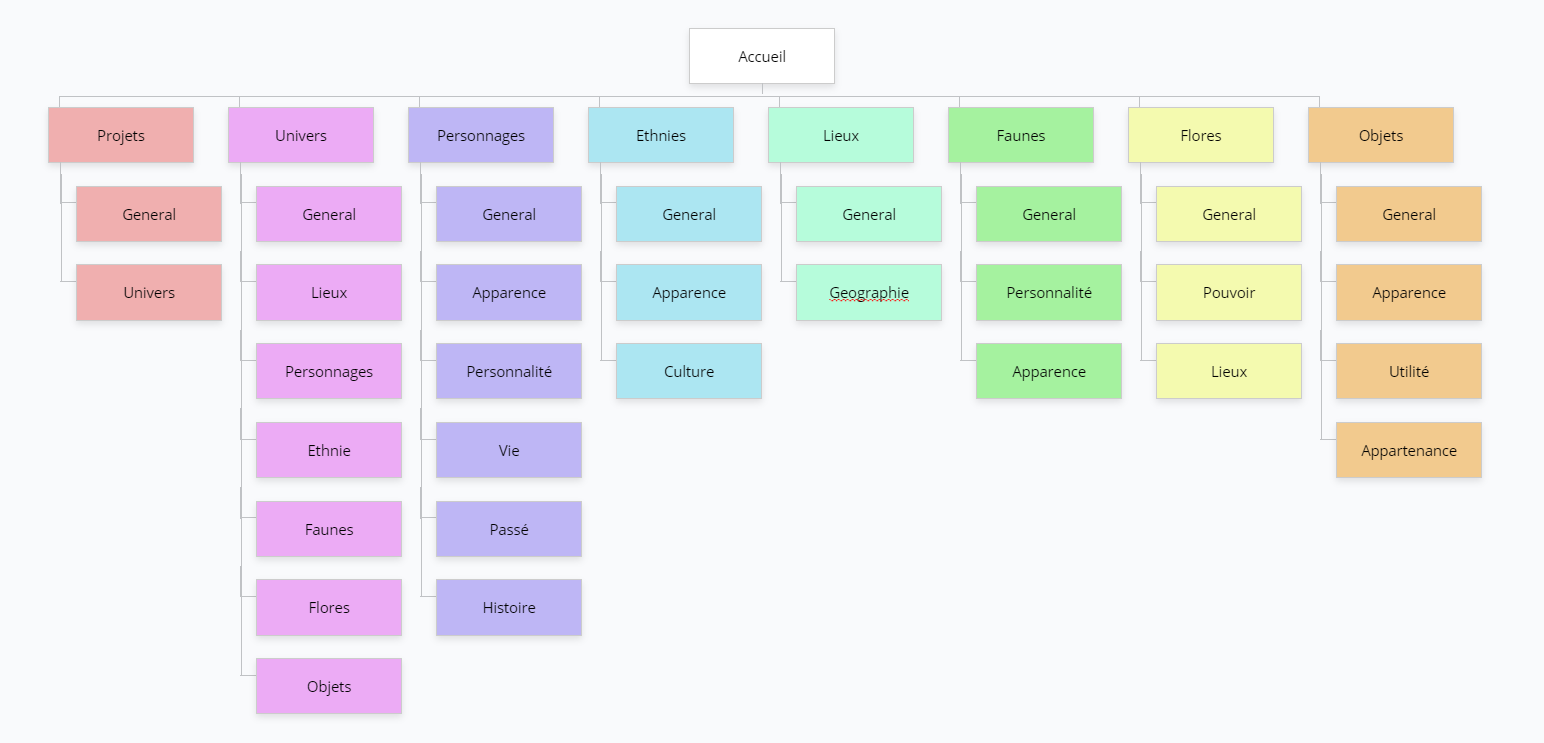


Maintenant que j’ai ma base de données, je peux réaliser l’arborescence de navigation de l’application.

Chapitre IV : L’arborescence

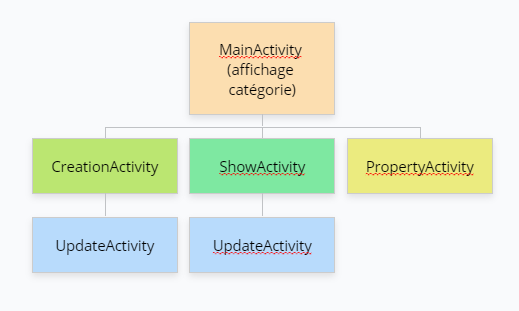
Pour l’application, j’ai créé deux arborescences.

La première est l’arborescence de navigation :



Nous avons d’abord les 8 catégories puis une fois qu’on n’a choisi une fiche ou qu’on n’en a créé une, j’ai dispatché les différentes informations dans des sous catégories afin de facilité le remplissage et la recherche d’information.

La seconde est l’arborescence des activités de l’application :



Nous avons d’abord le MainActivity qui permet de naviguer dans les différentes catégories. De là, on peut soit créer une fiche et démarrer CreationActivity qui une fois créer dirige vers UpdateActivity. Soit on peut afficher une fiche et demarrer ShowActivity qui nous permet de soit la supprimer ou de la modifier qui ouvrira UpdateActivity. De MainActivity, on pourra aussi ouvrir PropertyActivity où il y aura toutes les propriétés.

Chapitre V : Les outils techniques

Mes compagnons lors de la conception de la l’application sont :



Word : Il est la référence en traitement de texte. Il a été très utile pour élaborer le cahier des charges et tous les documents annexes.



StarUML : Je l’ai utilisé pour créer tous les diagrammes UML. Je l’ai choisi car il m’a été conseillé lors de la formation.



AdobeXD : Je l’ai utilisé pour réaliser la maquette de l’application. Je l’ai choisi car il est plutôt intuitif et permet de faire beaucoup de chose.



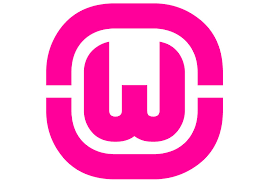
Looping : Je l’ai utilisé pour créer le MCD et le MLD. Je l’ai choisi car il m’a été conseillé lors de la formation.



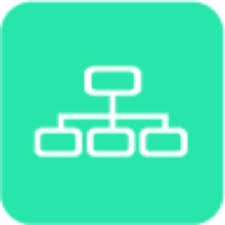
SQL : Il est le langage utilisé pour la base de données.



MySQL Workbrench : Je l’ai utilisé pour créer la base de données et réalisé les tests de requête sql lié.



WAMPServer : Je l’ai utilisé pour héberger la base de données en local.



Gloomaps : C’est le site que j’ai utilisé pour réaliser les arborescences. Je l’ai choisi car il permet facilement de réaliser des arborescences.

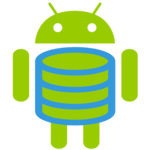
Mes compagnons pour la programmation de l’application sont :



Kotlin : C’est le langage que j’ai utilisé pour réaliser l’application. Je l’ai choisi car il est plus souple que Java et que je le préfère à celui-ci.

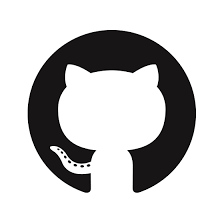


Android Studio : C’est le logiciel que j’ai utilisé pour réaliser l’application. Je l’ai choisi car c’est le plus pratique a utilisé et celui où je suis le plus à l’aise.



Android Room : C’est la base de données intégrée à Android.

Tout du long du projet, mon compagnon a été :



Github : C’est la plateforme d’hébergement que j’ai utilisé pour sauvegarder mon projet et versionner le projet.

Chapitre VI : Les fonctionnalités

Je vais présenter la fonctionnalité d’afficher les listes des enregistrements de la base de données dans leur catégorie.

Dans un premier temps, j’ai créé les différents modèles des objets que je vais utiliser. J’ai décidé de mettre le nom de table et les attributs en anglais dans la programmation pour obéir aux règles de programmation.

Prenons le modele Projet comme exemple. Je créais d’abord une classe ProjectBean. J’utilise le mot « Bean » pour bien identifié qu’il s’agit d’un modèle et prend la table Projet en référence.

Voici le code de la classe :

@Entity(  
 tableName = "project"  
)

class ProjectBean(  
 @PrimaryKey(autoGenerate = true)   
 val id\_project: Long = 0,

val name: String,  
 val description: String? = null,  
 val storyline: String? = null  
  
)

J’ai précisé avant la classe qu’il s’agit d’une entité dont le nom de la table se nomme project.

J’ai mis la variable « id\_project » en Long et à 0 pour qu’il sache à quel nombre commencer. Ensuite J’ai précisé au-dessus qu’il s’agissait d’une clé primaire et que celle-ci est généré automatiquement.

Ensuite, je créais l’interface de ce modèle que je vais placer dans la même classe et que je vais nommer ProjectDao pour les différencier et parce que nous allons ajouter le « @Dao » qui nous permet de la lié à la base de données. Voici le code :

@Dao  
interface ProjectDao {  
 @Query("SELECT \* FROM project ORDER BY name ASC")  
 fun getAll(): List<ProjectBean>  
  
 @Insert  
 fun insert(project: ProjectBean): Long  
  
 @Update  
 fun update(project: ProjectBean)  
  
 @Delete  
 fun delete(project: ProjectBean)  
  
 @Query("SELECT name FROM project ORDER BY name ASC")  
 fun getName(): List<String>  
}

A l’intérieur de cette interface, j’y met les différentes requêtes qui pourrait mettre utile. Je peux d’abord en faire des personnalisés comme la première et la dernière avec le « @query ()» en ajoutant la requête sql à l’intérieur. Mais j’utiliserais principalement les mots clé de l’ORM. L’ORM est un ensemble de classe permettant de manipuler plus facilement la base de données comme s’il était des objets. Dans ce code, nous en avons 3. Par exemple insert permet d’ajouter un enregistrement à la base de données.

Maintenant que j’ai créé les modèles et leurs interfaces, je vais pouvoir créer la base de données local dans room. Je créais une classe nommée SheetDatabase. Voici le code :

@Database(  
 entities = [ProjectBean::class, …, ItemBean::class],  
 version = 1,  
 exportSchema = false  
)  
abstract class SheetDatabase : RoomDatabase() {  
 abstract fun projectDao(): ProjectBean.ProjectDao  
 …  
 abstract fun itemDao(): ItemBean.ItemDao  
  
 companion object {  
 private var INSTANCE: SheetDatabase? = null  
 fun getDatabase(context: Context) =  
 INSTANCE ?: Room.databaseBuilder(context, SheetDatabase::class.*java*, "sheet\_database")  
 .fallbackToDestructiveMigration().build().*also* **{** INSTANCE = **it }** }  
}

Dans un premier temps, j’ai précisé que SheetDatabase est de type RoomDatabase(). Et que cette classe est abstraite. C’est-à-dire qu’elle est une classe qui sert de base à d’autre table dérivée qui hériteront de ses méthodes. Elle n’est pas instanciable.

J’ajoute ensuite le « @Database où j’y met en entité tous les modèles que j’ai créé qui deviendront les tables de la base de données.

Dans la classe SheetDatabase, j’y ajoute les méthodes reliées aux interfaces afin que la base de données soit utilisable.

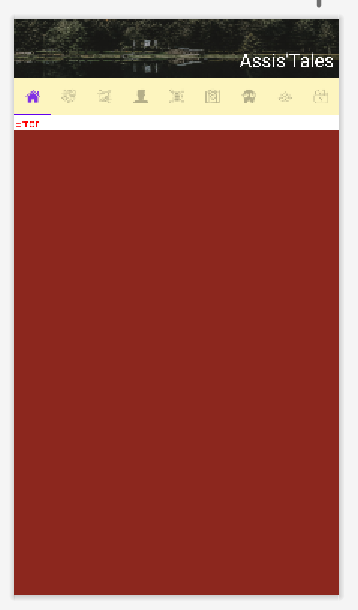
Ensuite, j’utilise le mot-clé companion object afin de pouvoir créer la méthode qui permettra de créer et d’instancié la base de données.

Je vais maintenant, relié la base de données à l’application en modifiant le fichier AssistalesApp.kt dont voici le code :

lateinit var *applicationInstance*: AssistalesApp  
  
class AssistalesApp : Application() {  
 val database by *lazy* **{** SheetDatabase.getDatabase(this) **}** override fun onCreate() {  
 super.onCreate()  
 *applicationInstance* = this  
 }  
}

J’ai donc ajouté la variable database ansi que le mot clé « lazy » qui permet de retarder l’instanciation de la variable jusqu’à ce qu’elle soit utilisée pour la première fois. Cette variable sera instanciée par la méthode getDatabase de SheetDatabase.

Maintenant que la base de données et créer, je vais mettre en place l’activité Main. Je créais en premier lieu l’xml, les éléments graphiques de la page d’activité.



Ici j’ai mis un texteView avec une image en background.

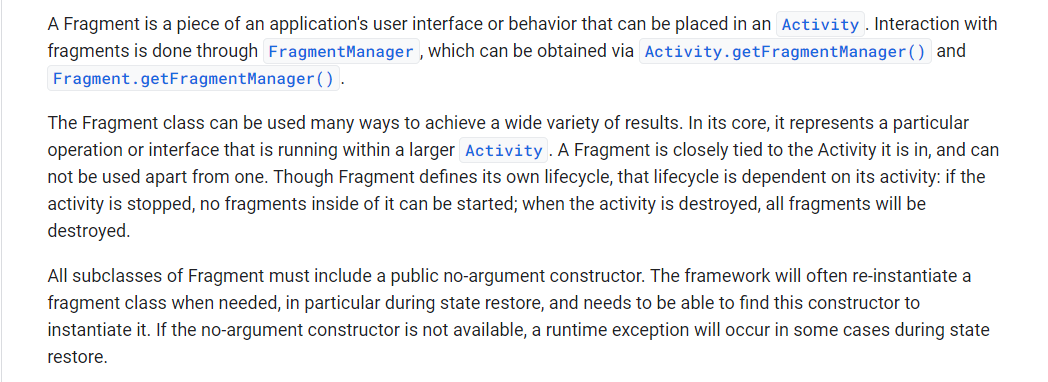
Ici, j’ai mis TabLayout afin de pouvoir naviguer sur les différentes catégories

Ici, j’ai mis un texteView en cas d’erreur. Il sera invisible la plupart du temps.

Ici, j’ai mis un FrameLayout qui me permettre que faire apparaitre l’accueil et les listes.

Le tout est dans un LinearLayout orienté à la verticale.

In order to display the lists of every category in the same Activity, I had to use Fragment. To understand how to use it, I read the android documentation about Fragment in english (https://developer.android.com/reference/kotlin/android/app/Fragment). So I translate it. Here are the first paragraphs and their translation:



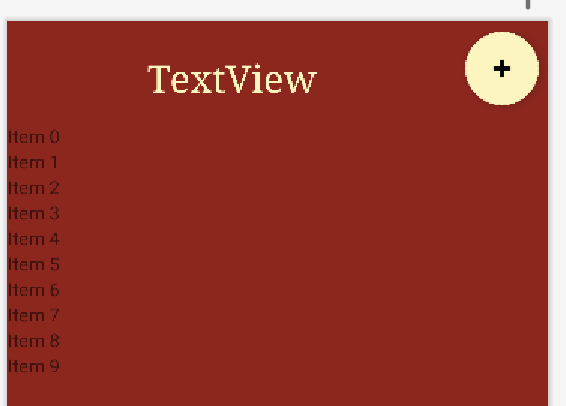
Translation :

 Un fragment est un élément de l’interface utilisateur de l’application ou de son comportement qui peut être placé dans un fichier « Activity ». L’interaction avec les fragments se fait par le biais de « FragmentManager », qui peut être obtenu par les méthodes « Activity.getFragmentManager() » et « Fragment.getFragmentManager() »

La classe Fragment peut être utilisé de plusieurs manière pour obtenir de nombreux résultat varié. Dans son noyau, il représente une opération particulière ou une interface qui est exécuté dans une Activité plus large. Un fragment est strictement lié à l’activité qui le possède et ne peut donc pas être utilisé seul. Bien qu’un fragment définisse son propre cycle de vie, ce cycle de vie est dépendant de son activité : si l’activité est arrêtée, aucun fragment à l’intérieur ne peut être démarré ; quand une activité est détruite, tous les fragments sont aussi détruits.

Toutes les sous-classes de Fragment doivent inclure un constructeur public sans argument. Le framework peut souvent re-instancié une classe fragment quand c’est nécessaire, en particulier pendant l’état de restauration, et doit pouvoir retrouver le constructeur pour l’instancier. Si le constructeur sans argument n’est pas disponible, une exception d’exécution pourra se produire dans certain cas durant l’état de restauration.

Après avoir lu, la documentation sur les Fragments j’ai commencé à créer les deux xml des fragments qui s’afficheront dans le FrameLayout du xml de l’activité main.



J’ai d’abord créé un LinearLayout orienté à la verticale qui englobera tous les éléments.

Dans un LinearLayout orienté à l’horizontal, j’ai mis un TextView qui donnera le titre de la catégorie et un FloatingActionButton.

En dessous, j’ai ajouté un ScrollView afin de pouvoir afficher la liste qui sera plus ou moins.

J’ai également dû créer un autre xml pour l’apparence graphique de la liste.



Tout comme les deux autres d’avant, j’ai d’abord mis un LinearLayout vertical puis j’y est placé un CardView composé d’un linearLayout horizontal avec un textView.

Maintenant que nos éléments graphiques sont créés, je vais passer au code.

Afin de faciliter l’accès aux éléments du xml, je vais mettre en place le View binding.

Je dois d’abord configurer le projet afin qu’il y ai accès. Dans le fichier build.gradle du module, je dois ajouter ses lignes dans la partie android{}

buildFeatures **{** viewBinding true  
 **}**

Ensuite dans le MainActivity, je vais créer une variable binding :

lateinit var binding: ActivityMainBinding

Ici, j’ai rajouté le mot-clé lateinit pour retarder l’initiation de la variable qui sera initialiser plus tard dans le code.

Ensuite je vais remplacer dans la méthode oncreate la ligne :

setContentView(R.layout.*activity\_main*)

qui a été généré automatique par :

binding = ActivityMainBinding.inflate(*layoutInflater*)  
setContentView(binding.*root*)

Je vais maintenant mettre en place les fragments. Pour l’instant, je vais d’abord faire juste afficher le Fragment d’accueil. Je mettrais en place les boutons un peu plus tard.

Il y a d’abord la méthode onCreate :

override fun onCreate(savedInstanceState: Bundle?) {  
 super.onCreate(savedInstanceState)  
  
}

Cette méthode est appelée à faire la création initiale du fragment.

Ensuite il y a la méthode onCreateView :

override fun onCreateView(  
 inflater: LayoutInflater, container: ViewGroup?,  
 savedInstanceState: Bundle?  
): View? {  
 *// Inflate the layout for this fragment* return inflater.inflate(R.layout.*fragment\_home*, container, false)  
}

Cette méthode créée et renvoie les vus associé au Fragment. Il permet de le relié au fichier xml afin d’avoir accès et d’afficher les éléments graphiques.

Ensuite je m’occupe du fragment qui affichera les listes des catégories. Dans ce fragment, j’ai un recyclerView alors je dois d’abord mettre en place un adapter. Je créais donc la classe SheetAdapter. Voici le code de la classe :

class SheetAdapter(val data: List<\*>?, val recepteur: OnSheetAdapterListener?) :  
 RecyclerView.Adapter<SheetAdapter.ViewHolder>() {  
  
 class ViewHolder(val bind: RowSheetBinding) : RecyclerView.ViewHolder(bind.*root*)  
  
 override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder =  
 ViewHolder(RowSheetBinding.inflate(LayoutInflater.from(parent.*context*)))  
  
 override fun onBindViewHolder(holder: ViewHolder, position: Int) {  
 var datum: Object = data!!.get(position) as Object  
 holder.bind.tvNameSheet.setText(datum.toString())  
 holder.bind.rootSheet.setOnClickListener(View.OnClickListener() **{** fun onClick(view: View?) {  
 if (recepteur != null) {  
 recepteur!!.onSheetAdapterClic(datum)  
 }  
 }  
 **}**)  
 }  
  
 override fun getItemCount(): Int {  
 return data!!.size  
 }  
  
 *//----------------------------Interface (emetteur)-------------------//* interface OnSheetAdapterListener {  
 fun onSheetAdapterClic(`object`: Any?)  
 }  
}

J’ai d’abord mis deux paramètres à la classe :

* une variable data qui est de type liste d’objet pour que je puisse lui passer des listes de différents types.
* Une variable recepteur de type OnSheetAdapterListener afin de récupérer la donnée lors d’un clic sur l’un des objets.

Je précise ensuite que la classe est un adapter de RecyclerView qui utilisera le ViewHolder de la classe pour l’affichage. Je créais donc à l’intérieur la classe ViewHolder qui prend un binding en paramètre. Je créais ensuite la méthode qui permettra de créer ce ViewHolder et de l’instancier. Après, je créé la méthode qui permettra d’entrer les données dans le View. Je créais donc une variable datum de type object qui ira récupérer un à un les objets contenus dans la liste obtenu en paramètre dans la classe Adapter.

Je passe ensuite dans l’élément graphique TextView qui a pour identifiant «tvNameSheet » le texte retourner par la méthode toString de l’objet contenu dans la variable Datum. Ne voulant afficher que le nom de l’objet, je suis allé surcharger la méthode toString dans les modèles :

override fun toString(): String {  
 return name  
}

Je mets ensuite la possibilité de clic sur l’élément graphique qui a pour identifiant « rootSheet ». Pour cela il doit lire la donnée puis la renvoyer mais je ne peux pas émettre de donnée directement. Je dois donc créer une interface afin de le faire qui aura une méthode qui recevra en paramètre un objet de type inconnu afin qu’il puisse être utilisé dans toute les listes.

Je vais donc retourner sur la classe fragments des listes. Je passe d’abord les différentes variables dont je vais avoir besoin pour la suite :

private lateinit var recyclerView: RecyclerView  
private lateinit var tvTitle: TextView  
private lateinit var fabAdd: FloatingActionButton  
private var data = ArrayList<Any>()  
private var adapter = SheetAdapter(data, this)  
private var title: String? = null  
private var mType: Int = 0

Je créais en premier les différents éléments de la page qui seront par la suite instancié avec les éléments graphiques. Puisque dans les fragments, je ne peux pas utiliser le binding, je suis donc obligé de les instancier manuellement. Ensuite il y a la variable data qui sera une arrayList, c’est-à-dire une liste dont la taille peut être extensible, d’objet. Ensuite vient la variable adapter qui est instancier de l’adapter que je viens de créer avec pour paramètre la variable data que je viens de créer et en recepteur le Fragment utilisé. Ensuite je créais également une variable title de type String que je mets en null pour l’instant et une variable mType de type Int qui me servira à identifier quelle catégorie a été choisi. Je mets toutes les variables en private afin qu’elles soient protégées et qu’elles ne puissent être utilisé que dans cette classe.

Je vais maintenant créer un constructeur afin de remplir la variable data de la bonne liste et d’afficher le bon titre. Voici le constructeur :

constructor(pType: Int) : super() {  
 mType = pType  
 *thread* **{** if (pType == 1) {  
 data.addAll(*applicationInstance*.database.projectDao().getAll())  
 title = "Projet"  
 …  
 } else if (pType == 8) {  
 data.addAll(*applicationInstance*.database.itemDao().getAll())  
 title = "Objet"  
 }  
 **}**}

Je passe en paramètre du constructeur la variable pType de type Int afin de pouvoir connaître le numéro de catégorie ensuite je passe ce paramètre dans la variable mType afin de pouvoir connaitre l’information plus tard.

Je mets ensuite en place un thread. Un thread est un processus qui s’exécute en parallèle des autres processus. A l’intérieur, j’y place une condition, si le paramètre pType est égale a à certain nombre, il va ajouter à la variable data tous les enregistrements de la table choisit de la base de données. J’instancie ensuite le bon titre à la variable Title. Je le fais donc pour chaque catégorie.

Ensuite, il y a la méthode onCreate qui est identique à celle de l’accueil puis je modifier le onCreateView :

override fun onCreateView(  
 inflater: LayoutInflater, container: ViewGroup?,  
 savedInstanceState: Bundle?  
): View? {  
 var view: View = inflater.inflate(R.layout.*fragment\_sheet*, container, false)  
 tvTitle = view.findViewById(R.id.*tv\_title*)  
 tvTitle.setText(title)  
 fabAdd = view.findViewById(R.id.*fab\_add*)  
 fabAdd.setOnClickListener **{** view **->** val intent = Intent(*activity*, CreationActivity::class.*java*)  
 intent.putExtra("type", mType)  
 startActivity(intent)  
 **}** recyclerView = view.findViewById(R.id.*rv\_array*)  
 recyclerView.*adapter* = adapter  
 recyclerView.*layoutManager* = GridLayoutManager(view.*context*, 1)  
  
 return view  
}

Contrairement à tout à l’heure, je ne renvoie pas directement la vue puisque je veux qu’elle se modifie en fonction de la catégorie. Je mets donc la vue dans une variable que j’ai nommé view. Ensuite j’instancie la variable tvTitle avec l’élément graphique du fichier xml lui correspondant. Nous pouvons donc maintenant la modifier grâce à .setText(). Je lui passe ensuite la variable title que j’ai instancié dans le constructeur.

Je m’occupe ensuite du bouton d’ajout qui se trouve sur la page. D’abord je l’instancie avec son élément graphique correspondant puis j’utilise la méthode setOnClickListener afin de lui dire ce qu’il doit faire avec un clic sur le bouton. Au clic, il doit démarrer l’activité de création mais je dois aussi lui passer l’information sur la catégorie où elle est afin de pouvoir enregistrer la nouvelle fiche dans la bonne table. Donc je créais une variable intent qui sera instancié de Intent, un objet qui représente une intention de l’application à faire quelque chose. J’ajoute ensuite la variable mType dans l’intent grâce à la méthode putExtra(). Puis je peux enfin démarrer l’activité avec la méthode StartActivity() en lui passant l’intention en paramètre.

Je peux enfin passer au RecyclerView. Je commence par l’instancier avec son élément graphique puis je lui précise son adapter et pour finir ce qui va le manager que je mets en grille, il va donc passer chaque objet de la liste un par un.

Pour finir, je peux retourner la vue.

Je créais ensuite la méthode qui va recevoir l’objet que l’adapter nous renvoie en cas de clic sur l’un des objets :

override fun onSheetAdapterClic(`object`: Any?) {  
 if (`object` is FloraBean) {  
 val floraBean: FloraBean? = `object` as FloraBean? }…

} else if (`object` is ItemBean) {  
 val itemBean: ItemBean? = `object` as ItemBean?  
 }  
}

Là, je reçois donc un objet inconnu, je dois donc l’identifier pour pouvoir ensuite l’afficher. J’utilise donc une condition pour identifier cet objet et ensuite je pourrais la transmettre à l’activité suivante. La prochaine activité n’étant pas encore terminé, cette méthode est donc incomplète.

Maintenant que les fragments sont terminés, nous allons pouvoir revenir sur l’activité afin de les afficher. Pour commencer, je vais mettre en place des variables qui pourront être utilisé dans toute la classe :

var fragment: Fragment? = null  
var fragmentManager: FragmentManager? = null  
var fragmentTransaction: FragmentTransaction? = null  
var mType: Int = 0

Nous avons d’abord toutes les variables utiles pour la création des fragments que l’on instancie en null. Il y a aussi la variable mType afin qu’une fois que l’écrivain a fini de créer une nouvelle fiche, il revienne directement sur la catégorie qu’il avait choisi.

Je retourne donc dans la méthode onCreate. J’y ajoute :

mType = *intent*.getIntExtra("type", 0)

Ici j’instancie mType de l’information se trouvant de l’Extra de l’intent que j’ai ajouté grâce à la méthode .putExtra. Si elle n’existe pas, la valeur par défaut sera 0 qui correspond à l’accueil.

Je peux maintenant commencer à instancier les variables des fragments :

fragmentManager = *supportFragmentManager*fragmentTransaction = fragmentManager!!.beginTransaction()

Ensuite je vais pouvoir afficher mes fragments en fonction du numéro dans la variable mType. Pour éviter de répéter le même code plusieurs fois, je vais créer avant deux méthodes qui me permettrons d’instancier la variable fragment du bon fragment à afficher, de modifier l’image en background du header et aussi de sélectionner le bon bouton de la barre de navigation :

fun showFragmentHome() {  
 fragment = HomeFragment()  
 binding!!.tvHead.setBackgroundResource(R.mipmap.*ic\_banner\_home\_foreground*)  
}

La première méthode est celui qui va permettre d’afficher la page d’accueil. J’utilise donc la méthode setBackgroundResource pour pouvoir modifier le fond du header.

fun showFragment(pType: Int, pImg: Int) {  
 fragment = SheetFragment(pType)  
 binding.tlMain.selectTab(binding.tlMain.getTabAt(pType))  
 binding!!.tvHead.setBackgroundResource(pImg)  
}

La deuxième méthode est celle qui affichera les listes. Je mets en paramètre le numéro de la catégorie et l’adresse de l’image. Ici, j’utilise la méthode .selectTab de la classe TabLayout afin de sélectionner le bon icon puis j’utilise la méthode .getTabAt avec le numéro de catégorie afin qu’il sache laquelle il s’agit.

Je retourne donc dans la méthode onCreate et y ajoute :

if (mType == 0) {  
 showFragmentHome()  
} else if (mType == 1) {  
 showFragment(mType, R.mipmap.*ic\_banner\_project\_foreground*)  
} …  
} else if (mType == 8) {  
 showFragment(mType, R.mipmap.*ic\_banner\_item\_foreground*)  
}

Je créais alors une condition qui teste la variable mType. Si elle est égale à 0 alors elle lance la méthode showFragmentHome que j’ai créé précédemment sinon si elle est égale à 1 alors elle lance la méthode showFragment avec pour paramètre mType et l’adresse de l’image d’en-tête sinon elle passe au suivant jusqu’à qu’elle obtient le bon.

Maintenant je peux lancer le fragment :

fragmentTransaction!!.replace(R.id.*fl\_main*, fragment!!)  
fragmentTransaction!!.setTransition(FragmentTransaction.*TRANSIT\_FRAGMENT\_OPEN*)  
fragmentTransaction!!.commit()

La méthode .replace permet de remplacer l’élément graphique FrameLayout par le fragment.

La méthode .setTransition permet de réaliser une animation de transition.

Enfin la méthode .commit permet de valider la transaction.

Je vais maintenant pouvoir gérer la barre de navigation :

binding.tlMain.addOnTabSelectedListener(object : TabLayout.OnTabSelectedListener {  
 override fun onTabSelected(tab: TabLayout.Tab) {  
 when (tab.*position*) {  
 0 -> {  
 showFragmentHome()  
 }  
 1 -> {  
 showFragment(1, R.mipmap.*ic\_banner\_project\_foreground*)  
 }  
 …  
 }  
 8 -> {  
 showFragment(8, R.mipmap.*ic\_banner\_item\_foreground*)  
 }  
 }  
 val fm = *supportFragmentManager* val ft = fm.beginTransaction()  
 ft.replace(R.id.*fl\_main*, fragment!!)  
 ft.setTransition(FragmentTransaction.*TRANSIT\_FRAGMENT\_OPEN*)  
 ft.commit()  
 }

override fun onTabUnselected(tab: TabLayout.Tab?) {}  
  
 override fun onTabReselected(tab: TabLayout.Tab?) {}  
})

Je commence d’abord par le lecteur d’ajout de sélection de l’onglet du TabLayout puis je peux alors utiliser la méthode de lecture de l’onglet au clic. Dans cette méthode je créais une méthode permettant d’afficher ce qui a été sélectionné. Je créais un switch en fonction de la position dans la table où il va utiliser les méthodes de sélection d’affichage des fragments. Je dois donc ensuit relancer l’affichage du fragment de la catégorie sélectionner.

Il y a ensuite deux méthodes qui doivent apparaître mais que je ne remplirais pas puisque l’application ne doit rien faire dans ses cas là.

Epilogue : La Conclusion

L’application n’est pas encore fonctionnelle et je dois encore mettre en place de nombreuses fonctionnalités (modification, affichage de la fiche, …) avant la finaliser. Je compte la poursuivre afin de gagner en expérience et en pratique.

Ce projet m’a appris beaucoup de chose au niveau technique : le langage kotlin, l’utilisation d’Android Studio, les Fragments, … Mais il m’a surtout appris à faire des recherches, consulter la documentation et à tester au fur et à mesure de la programmation afin de mieux identifier les erreurs que je pouvais rencontrer. J’ai également appris à optimiser mon code afin qu’il soit moins lourd et plus efficace. Ce qui m’a posé le plus de difficulté, ce fut de bien comprendre et de mettre en place les fragments ainsi que l’utilisation d’objet générique.

Je tiens à remercie toutes l’équipe de l’adrar pour tout ce qu’ils m’ont appris, leur patience et leurs soutiens lors de cette formation.

Et comme la si bien dit Saint-Exupéry dans le petit prince :

*« La fin d'une chose marque le commencement d'une nouvelle ».*

Annexe

Cahier des charges

1. Introduction
   1. Présentation

Fictioncollab est une association auteurs de tous genres qui a été créer en 2015 ayant pour objectifs de faire se rencontrer les écrivains, de mettre en place des événement autour des mots, de réaliser des ateliers d’écriture et de lecture, d’aider et de conseiller les jeunes et les nouveaux auteurs et d’encourager à la littérature. Suite à de nombreuses discussions avec les nouveaux auteurs et les retours des ateliers, ils ont l’idée de mettre à disposition une application mobile d’aide à l’écriture d’histoire.

* 1. Intervenants

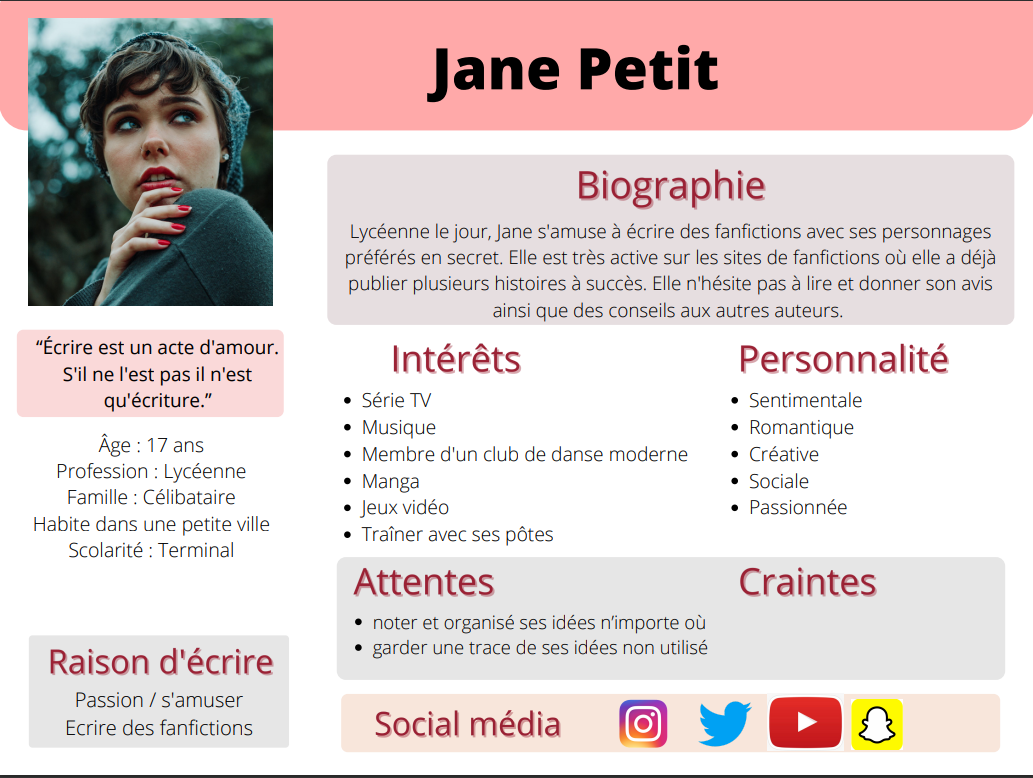
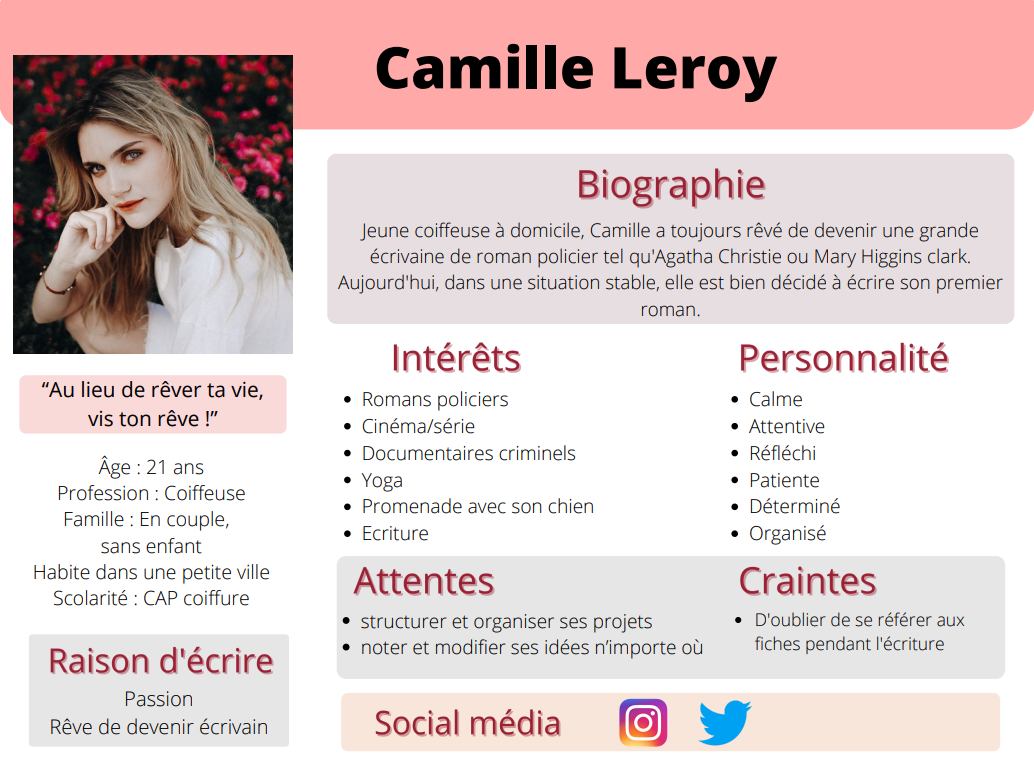
Le client, Fictioncollab, représenté par Lilia McDonel, présidente de l’association.

Le prestataire représenté par Jessica Anglade, developpeuse.

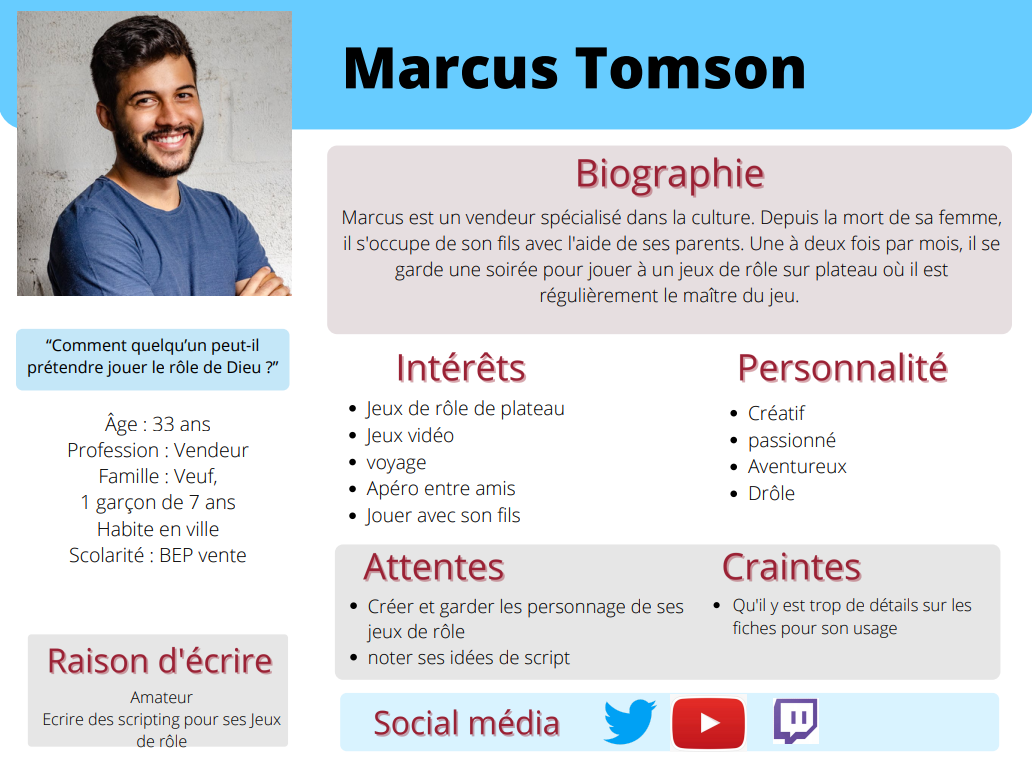
1. Contexte
   1. Objectifs

Les objectifs de l’application :

* permettre aux auteurs de gérer leurs projets d’histoire
* mettre à disposition des outils permettant aux auteurs d’organiser et de structurer leurs récit ainsi que les éléments (personnages, lieux, …) qui les façonnent.
* créer des vocations dans l’écriture que ce soit en amateur ou en professionnel
* stimuler l’imagination
  1. Cible







De cette étude de la cible et des personas, il en est ressorti que :

* Les utilisateurs pourront être des professionnels ou des amateurs
* Les utilisateurs pourront l’utiliser écrire différents types d’ouvrages comme par exemple : des romans, des scénarios, des scripts pour le jeu de rôles, des fanfictions (fiction écrit par des fans ayant lieux soit dans un univers et/ou avec des personnages d’une œuvre existant)
* Les utilisateurs peuvent être de tout âge, avoir n’importe quels métiers, d’en n’importe quelles situations personnelles (célibataire, marié, …).
* Les utilisateurs sont des personnes passionnées par l’écriture.
* Les attentes des utilisateurs sont de pouvoir noter ou consulter leurs idées n’importe où et n’importe quand, de pouvoir s’organiser efficacement et de pouvoir retrouver facilement les informations dont ils ont besoins.
  1. Concurrence

Les concurrents sont :

|  |  |  |  |
| --- | --- | --- | --- |
| Nom | Langues | Points forts | Points faibles |
| Character Story Planner | Anglais, | - fiches complètes  - 6 différentes catégories  - modification en catégorie dans les fiches  - lecture des fiches pratique | - pas très intuitif  -pas de liens entre toutes les fiches et pas rétroactive |
| Character Story planner 2 | anglais | - fiche complète  - différentes catégories  - possibilité de télécharger des personnalisation  -importation/exportation de la base de données  -plusieurs liens entre les fiches  -modification en catégorie dans les fiches  - lecture des fiche pratique | - impossibilité de créer une fiche sans créer un projet  - liens non rétroactive,  - relations personnages dans catégorie à part |
| Writer Tools | Plusieurs langue dont le français | - fiche complète  - notation d’idée  - écrire des chapitres  - un thésaurus  - lecture des fiches pratique  - multilingue | -demande une connexion à un compte  - pas de possibilité de passer à un projet en hors ligne  -sauvegarde en ligne exclusivement  - modification en complète |
| Mystory.today | anglais | - peut écrire les chapitres  - exportation en format document | - obligation de s’inscrire pour sauvegarder  - création d’un projet pour avoir fiche  - fiche courte |

* 1. Analyse marketing

SWOT :

|  |  |
| --- | --- |
| **Forces**  - application en français  - plusieurs catégories  - personnalisation des fiches  - lien entre les fiches  - exportation en format document  - 2 modes de fiche (simplifié et complète) | **Faiblesses**  - première prise en main complexe  - pas de possibilité de travaillé un projet en équipe  - pas de possibilité d’écrire les chapitres directement  - la catégorie « idées » peut devenir très vite conséquente  - compte de sauvegarde en ligne payant |
| **Opportunités**  -ouverture vers divers domaine d’écriture (scripting jeux de rôle en plateau, fanfiction, roman, scénario,…)  - création d’une communauté  - possibilité d’un partenariat avec des sites de fanfiction, ... | **Menaces**  - cible une catégorie de personnes spécifiques  - désintérêt |

1. Définir les besoins
   1. Énoncé du besoin

Le client devra fournir le contenu des fiches. Il devra également acheté le nom de domaine, un abonnement d’hébergement.

Le prestataire devra fournir une formation poussé à l’utilisation de l’outils.

Le prestataire réglera les problèmes de bugs rencontrer par les utilisateurs dans la limite de 4 mois après le lancement de l’application.

La maintenance, la gestion des comptes, des paiements et le SAV sera à la charge du client après le lancement de l’application.

Le prestataire peut présenter l’application à ses futurs clients.

* 1. Les fonctionnalités

Les fonctionnalités qui seront mise en place en ordre d’importance :

* Création des fiches (personnage, lieux,…)
* Possibilité de les intégrer dans un projet ou non
* Possibilité de mettre des liens entre les fiches (relation entre les personnage, personnage habitant dans un lieu, …)
* possibilité de masquer des infos dans la fiche
* création de 2 modes de fiches (simplifié et complète)
* possibilité de créer des fiches personnalisés par défaut
* création de compte avec sauvegarde en ligne (payant)
* possibilité d’exportation en format document

1. Contraintes
   1. Contraintes techniques

Le prestataire devra proposer 2 logos différents et la charte graphique qui en découlera.

Le prestataire choisira les logiciels adéquate à la création de l’application sans surcoût au devis.

* 1. Contraintes légales et réglementaires

- Le Client détient le droit exclusif d’exploitation du produit dans toutes ses versions. Dans le respect de l’oeuvre, le Client a l’impossibilité de transformer, recadrer, modifier du produit sans l’accord écrit du Prestataire. Tous autres cessions de droits non spécifiées sont nulles et non avenues.

- Le Prestataire décline toute responsabilité en cas de non-respect des légendes fournies. Dans le cas où l’article d’accompagnement ou la légende seraient éronnés et/ou diffamatoires, la responsabilité du diffuseur, de l’éditeur et de l’auteur de l’écrit est en cause, à l’exclusion de celle du Prestataire.

* 1. Contraintes de coûts

L’achat d’image n’étant pas prit en compte, il sera à la charge du client suivant sa demande.

* 1. Contrainte de délais

Le projet débute le 20 avril 2021 et l’application sortira le 22 avril 2022. Durant ce lapse de temps, le prestataire devra soumettre les éléments graphiques à validation avant le 10 mai, soumettre la maquette à validation avant le 31 mai. Les tests finaux devront commencer au plus tard, le 20 décembre 2020.

1. Déroulement du projet
   1. Livrables

Le prestataire devra livrer :

* le logo et la charte graphique
* la maquette
* la base de données
* l’application
* fonctionnalités