Solo Leveling Roleplay

« Réalisation d’un bot discord sur le thème du manhwa Solo Leveling pour une communauté de roleplay écrit. »

Par Vasco Valadares Semana – En 2023 – Projet de Cours

Table des matières

Introduction

Ce projet s'intitule « Solo Leveling Roleplay », il consistait en la création d'un bot discord sur le thème du manhwa Solo Leveling, ainsi que d'une communauté sur le réseau social Discord où les membres pourraient créer un personnage, le développer via la complétion de donjons, de quêtes, l'affrontement entre joueurs et contre des monstres, etc !

Informations

Contexte :

Ce projet a été réalisé de ma propre initiative. Je m'étais depuis peu intéressé au roleplay écrit sur le réseau social Discord et avait remarqué que peu de personnes utilisaient les fonctionnalités du réseau pour optimiser l'expérience des roleplayers. J'ai donc décidé de mettre mes compétences préalablement acquises lors du développement d'un autre bot discord (projet « Tools Project ») dans la création de celui-ci.

Objectifs :

Le but de ce projet était de créer un MMORPG interactif, reposant sur de la génération d'images et de texte pour divertir, simplifier et automatiser l'aventure roleplay des différents membres de la communauté. Il devait notamment permettre de renseigner des « Maîtres du Jeu », qui auraient accès à des commandes de modération (modification des informations des joueurs, création d'évènements, gestion des « PNJ », etc.)

Fonctionnalités Principales :

- 500+ Items à utiliser dans le jeu et plein d'autres prévus ;

- 100+ Commandes à utiliser dans le jeu, toutes plus utiles que les autres ;
- Plein de donjons à explorer avec des récompenses à gagner ;
- Une multitude de PNJs à rencontrer avec des histoires uniques à raconter ;
- Création presque infinie de compétences mettant en valeur l'imagination des joueurs ;
- Vrais systèmes économiques pour penser à deux fois avant de dépenser ;
- Une multitude de récompenses à gagner pour encourager les joueurs à progresser.

Développement

Technologies Utilisées :

- J'ai choisi d'utiliser **JavaScript** comme langage de programmation pour le projet. Ce choix vient du fait que parmi tous les autres langages, la librairie discord.js est la plus maintenue et avancée au niveau de ses fonctionnalités. De plus, j'avais déjà de l'expérience avec cette librairie et voulait perfectionner ma maîtrise de ce langage.

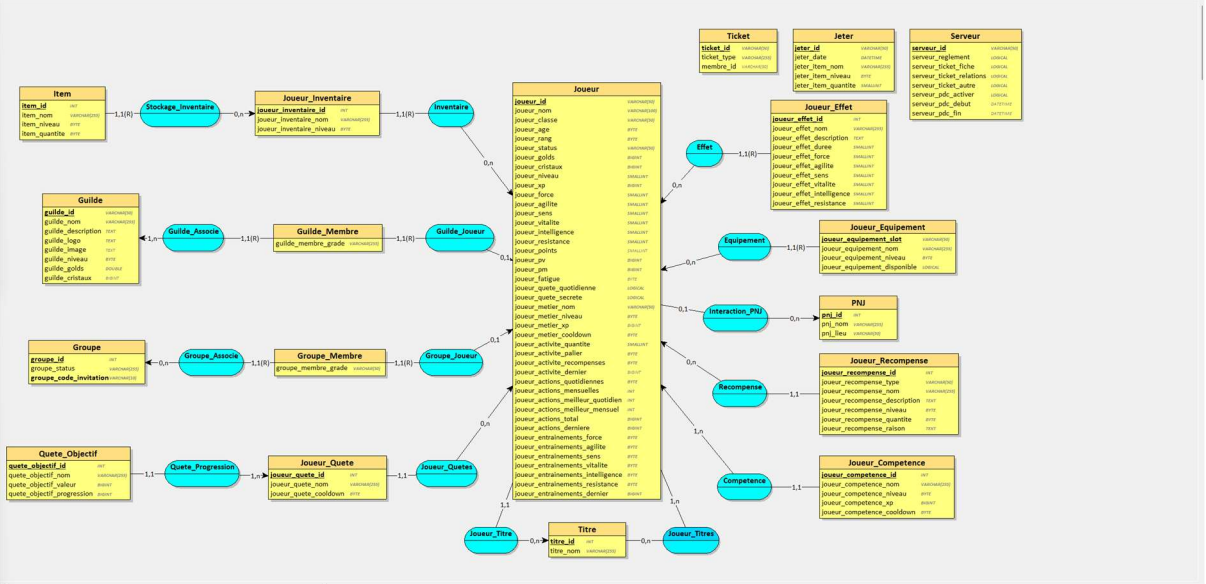
- J'ai utilisé **postgresql** comme base de données. De un, car j'avais déjà de l'expérience avec MongoDB et que je voulais apprendre les bases de données relationnelles ; de deux, car SQL est requis pour le BTS ; et de trois, car je pouvais héberger gratuitement une base de données postgresql sur un peu plus d'un an.

- J'ai utilisé un ORM pour faciliter l'implémentation de postgresql à JavaScript. L'outil que j'ai choisi était **Sequelize** car il m'avait été conseillé par un autre développeur.

- Pour la génération d'images, j'ai utilisé le module **canvas** pour NodeJS car il semblait être maintenu et facile d'utilisation.

- Pour l'hébergement de la base de données et du bot lui-même, j'ai décidé d'utiliser **Heroku**. Ce choix est dû à l'offre « github students » qui nous donne 460 euros de crédits dans toute la plateforme, ce qui était suffisant pour héberger confortablement la base de données ainsi que le bot durant près d'un an.

Modèle Conceptuel de Données :



Structure du Programme :

J’ai organisé mon projet en plusieurs parties :

```
node_modules/       // Modules NodeJS
src/                // Dossier de développement principal
   ├── commandes/   // Les différentes commandes
   ├── database/    // Les modèles de données, la connexion, etc
   ├── evenements/  // Les différents évènements discord
   ├── ressources/  // Les ressources du projet, images
   ├── utils/       // Les fonctions générales du projet
   ├── config.js    // Mapping javascript des variables d'environnement
   └── index.js     // Fichier principal du projet : initialisation du bot + bdd
.env                // Variables d'environnement
.gitignore          // Fichier gitignore pour le versioning
eslint.config.mjs   // Configuration d'ESLint
package.json        // Modules utilisés
Procfile            // Configuration d'Heroku
README.md
```

Détails du Code :

1. Gestion des Commandes :
   Afin de créer et gérer les commandes facilement et efficacement, les commandes sont regroupées sous la forme de fichiers, eux-mêmes organisés sous la forme de dossiers

présents dans le dossier « commandes ». Chaque commande peut alors être développée, puis déployée facilement.

2. Exemple de Commande :

La commande **commencer** est une des commandes principales, celle-ci permet la création des personnages. Pour ce faire, les membres doivent renseigner plusieurs informations à leur sujet :

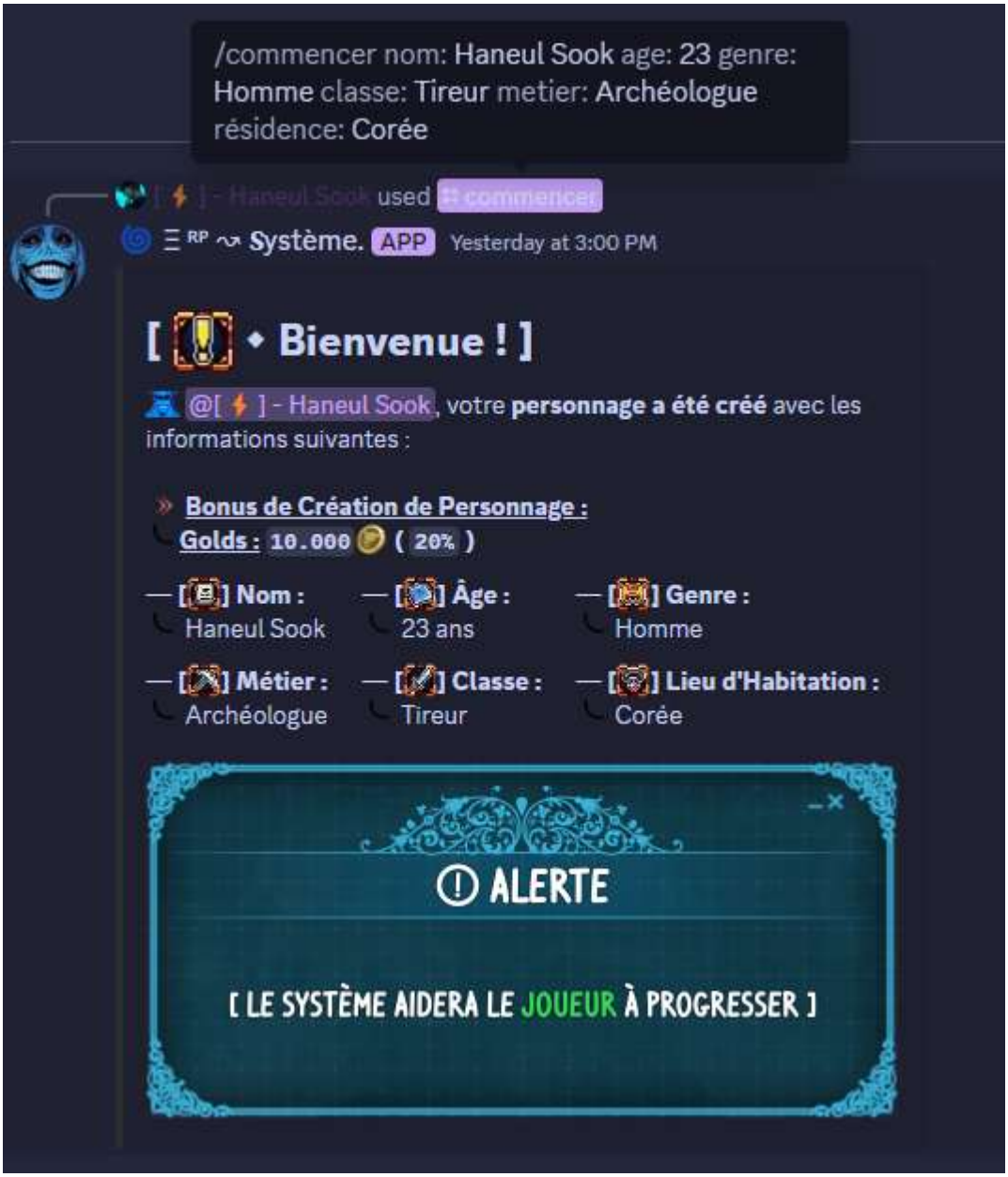
- Le **nom** du personnage *(champ de type VARCHAR(50) dans la base de données) ;*
- L'âge du personnage *(champ de type SMALLINT dans la bdd)* ;
- Le **genre** du personnage *(géré par un rôle dans le serveur discord) ;*
- *La **classe** du personnage (champ de type VARCHAR(50), avec un nombre d'options prédéfinies)*, le **métier** du personnage *(champ de type VARCHAR(50), avec un nombre d'options prédéfinies)*, et enfin le **pays de résidence** du personnage *(géré par un rôle dans le serveur discord)*.

Ensuite, le bot attribue automatiquement quelques informations, de départ, telles que :

- Le **niveau** du personnage (*champ de type SMALLINT dans la bdd, tous les joueurs commencent au niveau 1)* ;
- L'**expérience** du personnage *(champ de type BIGINT dans la bdd, tous les joueurs commencent à 0 d'expérience) ;*
- Le **statut** du personnage *(champ de type VARCHAR(50) dans la bdd, avec un nombre d'options prédéfinies, tous les joueurs commencent avec le statut « en vie »)* ;
- etc.

Les erreurs sont gérées et un message est envoyé au joueur lorsqu'une erreur se produit, en décrivant le problème source de façon humanisée et compréhensible par tous types d'utilisateurs. Si le problème est technique, le joueur est suggéré de contacter un administrateur/maître du jeu.

Message envoyé lors de la création du personnage :



3. Gestion des Événements :
   La gestion des événements se fait de façon similaire aux commandes. Chaque événement peut ainsi être lancé automatiquement au démarrage du bot.

4. Exemple d'Événement :
   Un des événements principaux est celui d'accueil de nouveaux membres : lorsqu'un utilisateur de discord rejoint le serveur, l'événement « onMemberAdd » est lancé. Le bot envoie donc un message de bienvenue au joueur lorsque cet événement est appelé.

5. Gestion des Ressources :

   Les ressources sont les images, textes, valeurs, etc, qui ne changent pas et peuvent être réutilisées dans différentes parties du code. Parmi ces ressources se trouvent notamment :

   - Les **items**, définis sous forme de classes. Voici la classe parent, qui est ensuite utilisée pour créer différentes variantes d'items :

```
-  /**
-   * @param {string} nom Le nom de l'item.
-   * @param {string} description Une courte description de l'item.
-   * @param {string} image Le chemin d'accès à l'image de l'item.
-   * @param {string} type Le type de l'item.
-   * @param {keyof typeof ItemCategorie} type La catégorie de
   l'item.
-   * @param {number} niveau_max Le niveau maximal de l'item.
-   */
-  class Item {
-    nom = undefined;
-    description = undefined;
-    image = undefined;
-    type = undefined;
-    categorie = undefined;
-    niveau_max = undefined;
-
-    /**
-     * @param {string} nom
-     */
-    setNom(nom) {
-      this.nom = nom;
-      return this;
-    }
-
-    /**
-     * @param {string} description
-     */
-    setDescription(description) {
-      this.description = description;
-      return this;
-    }
-
-    /**
-     * @param {string} image
-     */
-    setImage(image) {
-      this.image = image;
-      return this;
-    }
-
```

```
-     /**
-      * @param {keyof typeof ItemType} type
-      */
-     setType(type) {
-       if (!Object.values(ItemType).includes(type)) return this;
-       this.type = type;
-       return this;
-     }
-
-     /**
-      * @param {keyof typeof ItemCategorie} categorie
-      */
-     setCategorie(categorie) {
-       if (!Object.values(ItemCategorie).includes(categorie))
   return this;
-       this.categorie = categorie;
-       return this;
-     }
-
-     /**
-      * @param {number} niveau
-      */
-     setNiveauMax(niveau) {
-       if (niveau > NiveauMax) this.niveau_max = NiveauMax;
-       else this.niveau_max = niveau;
-       return this;
-     }
-   }
```

Les classes suivantes héritent de celle-ci :

- **Consommable :** item ayant la particularité de pouvoir être consommé par le joueur, lors de la consommation, un événement définit est déclenché ;
- **Equipable :** item ayant la particularité de pouvoir être équipé par le joueur, il octroie au joueur des attributs supplémentaires ;
- **Equipable->Arme :** classe héritant de **Equipable**, ces items ont des événements déclenchés lorsque le joueur attaque ;
- Tout autre type d'items est une instance de la classe **Item** elle-même.

- Les **PNJs** (personnages non joueurs), qui sont eux aussi définis sous forme de classes : une classe **PNJ**, qui définit les informations sur le PNJ lui-même ; et une classe **Dialogue** qui définit les dialogues du PNJ.
- Les **Titres**, eux aussi définis sous forme de classes.
- Les **Quêtes**, définies sous forme de classes, avec des objectifs à atteindre et une façon de les incrémenter en utilisant des événements.
- …