

DOSSIER PROJET

BASE DE DONNEES DU JEU *CHRONICLES RUNNER*

<i>Nom de naissance</i>	▶ FOUCHER
<i>Nom d'usage</i>	▶ FOUCHER
<i>Prénom</i>	▶ Simon
<i>Adresse</i>	▶ 105 Grande rue 69600 Oullins

Titre professionnel visé

Concepteur développeur d'applications

COMPETENCES DU REFERENTIEL :

Activités types : Concevoir et développer la persistance des données en intégrant les recommandations de sécurité

- Concevoir une base de données
- Mettre en place une base de données
- Développer des composants dans le langage d'une base de données

Sommaire

Liste des compétences du référentiel	p. 1
--------------------------------------	------

Présentation du projet	p. 3
------------------------	------

▶ Resume/Résumé	p. 4
-----------------------	------

▶ Cahier des charges et expression des besoins	p. 5
--	------

Conception de la base de données	p. 6
----------------------------------	------

▶ Création d'un dictionnaire	p. 6
------------------------------------	------

▶ Réalisation d'un modèle conceptuel des données.....	p. 6
---	------

▶ Réalisation d'un modèle logique des données.....	p. 8
--	------

Mise en place de la base de données	p. 8
-------------------------------------	------

▶ Création de la BDD	p. 8
----------------------------	------

▶ Gestion des utilisateurs	p. 10
----------------------------------	-------

▶ Veille et recherches : Sécurisation des données utilisateurs	p. 12
--	-------

▶ Mise en place d'un backup	p. 13
-----------------------------------	-------

Développement des composants	p. 14
------------------------------	-------

▶ Déclencheurs/Triggers	p. 14
-------------------------------	-------

▶ Vues/Views	p. 15
--------------------	-------

▶ Procédures stockées/Stored Procedures	p. 16
---	-------

Situation de travail ayant nécessité de la recherche	p. 18
--	-------

Conclusion	p. 19
------------	-------

Présentation du projet

Resume

This document aims to present the work carried out on the Chronicles Runner database, a Unity3D-developed game. This game is unique in for its feature of placing persistent datas at the core of its concept. Merging the classic mechanics of infinite runner games with a collaborative « treasure hunt », the game offers to each player the opportunity to discover a « fragments », which has an unique existence in all the database, and to claim their discovery by becoming the only owner. Created by using SQLite, this database ensures a crucial functional foundation, including the management of available and collected fragments, displaying discovered fragments with date and owner's name, as well as other features such as player registration, best times tracking and access to the different game levels.

Résumé

Ce dossier présente le travail réalisé autour de la base de données de Chronicles Runner, un prototype de jeu développé sur Unity3D. Ce jeu se distingue par sa particularité de placer les données persistantes au cœur même de son concept. En fusionnant les mécanismes classiques des jeux infinite runners - où le personnage court sans fin en évitant les obstacles et en collectant des pièces - avec une chasse au trésor collaborative, le jeu offre à chaque joueur la possibilité de découvrir des « fragments » à l'existence unique et d'en revendiquer la découverte en devenant l'unique propriétaire de ces fragments au sein de toute la base de données. Réalisée en utilisant SQLite, cette dernière assure un socle fonctionnel crucial, incluant la gestion des fragments disponibles et collectés, l'affichage des fragments déjà découverts avec la date et le nom du propriétaire, ainsi que d'autres fonctionnalités telles que l'enregistrement des joueurs, le suivi des meilleurs temps de course et la régulation de l'accès aux différents niveaux de jeu.

Cahier des charges et expression des besoins

Contexte

Dans le cadre d'une formation de Développeur de jeux vidéo, nous avons été amené à développer un jeu de type infinite runner en C# et utilisant SQLite, via le moteur de rendu Unity3D. La recherche d'un concept original nous a amené à imaginer comment la base de données pouvait non seulement soutenir le gameplay, mais surtout en devenir un élément essentiel, en poussant les utilisateurs à « jouer » avec elle et en faisant en sorte qu'ils puissent chercher à découvrir tout ce qu'elle contient.

Nous avons donc imaginé le synopsis suivant : « Dans un futur lointain, le Temps a été détruit. Avec lui, l'entière de l'Histoire de l'Humanité a disparu et personne ne s'est ce qui s'est véritablement passé. Heureusement, des hommes et des femmes ont trouvé un moyen d'accéder à des brèches temporelles : en allant courir dans différentes époques, ils peuvent découvrir des Fragments d'Histoire, des extraits de texte qui leur permettra, ensemble de reconstituer l'Histoire du monde tel qu'il était. »

Derrière ce background se cache finalement une grande chasse au trésor : lorsqu'il court, un joueur peut découvrir une entrée encore non découverte de la base de données, un « fragment d'histoire » unique qu'il peut ramasser pour en devenir l'unique destinataire. Ce fragment apparaîtra alors aux yeux de tous comme un extrait de texte consultable accompagné du nom du joueur qui l'a découvert et de la date de la trouvaille.



Exemple de fragment d'histoire

Objectifs

Les objectifs de la base de données nécessaire à Chronicles Runner se définissent en quatre points :

1. L'objectif premier est la mise en place d'une base de données contenant une importante quantité d'extraits textuels uniques, possédables par un seul utilisateur, et donc rares.
2. Développer des composants sous la forme de déclencheurs et de vues facilitant le traitement de ces données.
3. Objectif induit par la notion de possession : établir le socle d'un système de comptes utilisateurs sécurisé, afin que chaque joueur identifié puisse retrouver les fragments qu'il possède.

4. Enfin, le quatrième objectif est de fournir à Unity3D les informations nécessaires aux mécaniques d'un infinite runner classique, comme garder une trace des meilleurs temps de courses et des niveaux débloqués par l'utilisateur.

Spécifications fonctionnelles

L'étude du gameplay souhaité, des features envisagées et des contraintes nous ont permis d'établir les spécificités fonctionnelles auxquelles notre base de données allait devoir répondre :

Stocker et gérer les utilisateurs avec système d'inscription et de connexion sécurisé :

Il faut pouvoir stocker les informations d'identifications de l'utilisateur, comme son pseudonyme, son mot de passe hashé et un salt, pour un hashage plus performant contre les rainbow tables.

Stocker et gérer les fragments d'histoire :

Nous devons également pouvoir stocker les fragments d'histoire, incluant leur titre, leur contenu textuel, l'époque dans laquelle ils se trouvent. Bien que non implémentée dans un premier temps, une feature de rareté est envisagée pour l'avenir du jeu. De plus, il faudra pouvoir associer chacun d'eux à un joueur, ainsi qu'à une date de découverte.

Stocker et gérer les époques :

La base de données doit stocker le nom des époques historiques visitables dans le jeu. Elle doit également pouvoir faire remonter des informations relatives aux nombres de fragments qui lui sont associés : combien y'en a-t-il, combien sont encore disponibles, combien ont été trouvés... ?

Permettre le suivi de progression dans le jeu :

La base de données doit contenir les informations permettant de déterminer si un utilisateur a accès ou non à une époque particulière et si oui, quel est le meilleur score qu'il y a fait.

Permettre la consultation des fragments découverts :

La base de données doit être capable de faire remonter tous les fragments d'histoire associés à un joueur ou à une époque donnée.

Fonctionner avec SQLite :

Choisie par les formateurs pour sa légèreté et son intégration rapide sur Unity3D parce qu'elle permet de faire du prototypage rapidement, la bibliothèque SQLite doit être utilisée pour concevoir notre base de données. Il s'agit d'une considération technique importante, car cela signifie que nous n'aurons par exemple pas accès aux procédures stockées ou à la gestion des droits. De plus, étant donné l'aspect collaboratif du jeu, cela signifie que nous devons garder en tête qu'une migration vers des systèmes comme MySQL sera plus que probable pour un passage en production.

Conception de la base de données

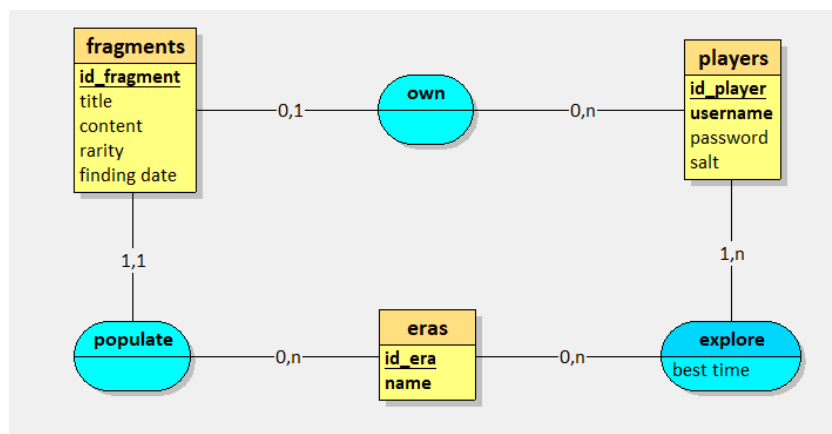
Création d'un dictionnaire

À partir des spécifications fonctionnelles, nous avons mis en place un dictionnaire des données afin d'identifier les données qui devront être conservées dans notre base et de faciliter la réalisation d'un modèle de conception des données.

Nom	Signification	Type	Taille
id_player	Identifiant du joueur	N	
username	Pseudonyme du joueur	AN	32
password	Mot de passe hashé du joueur	AN	500
salt	Salt attribué au joueur	AN	64
id_era	Identifiant de la période historique	N	
name	Nom de la période historique	A	25
id_fragment	Identifiant du fragment d'histoire	N	
title	Titre du fragment d'histoire	AN	128
content	Contenu du fragment d'histoire	AN	1000
rarity	Rareté du fragment d'histoire (1 banal, 10 légendaire)	N	2
finding date	Date où le fragment a été trouvé en timestamp Unix	N	10
best time	Meilleur temps du joueur en millisecondes	N	10

Modèle conceptuel des données

Une fois nos données identifiées, nous sommes en mesure d'aborder notre future base de données sous l'angle d'un schéma entité-association en réalisant un modèle conceptuel des données :



Fragments, players et eras deviennent nos trois entités, possédant des attributs qui leur sont propres. Les associations nous permettent d'établir des cardinalités entre-elles, qui nous serviront pour concevoir un modèle logique des données avant une insertion dans une SGBDR.

D'après ce que nous avons déterminé précédemment : un joueur (player) de notre jeu explore (explores) une période historique (era). Il met un certain temps à le faire. Le meilleur temps (besttime) dépendant à la fois du joueur à la fois de la période historique, nous choisissons de le mettre en attribut de l'association.

Un joueur peut explorer au minimum 1 période historique, au maximum n périodes historiques. Le choix de mettre la cardinalité minimum à 1 vient du fait que dans notre jeu, un nouveau joueur se voit automatiquement attribuer une période historique à l'inscription. Il n'y a donc pas la possibilité qu'un joueur se retrouve avec 0 période historique à explorer.

En revanche, une période historique peut être explorable par 0 joueur au minimum, n au maximum. Le choix de mettre la cardinalité minimale à 0 s'explique par exemple dans le cas de notre tout premier inscrit : à son arrivée, celui-ci aurait une seule période disponible et puisqu'il sera le seul sur le jeu, toutes les autres époques seraient alors explorables par 0 joueur.

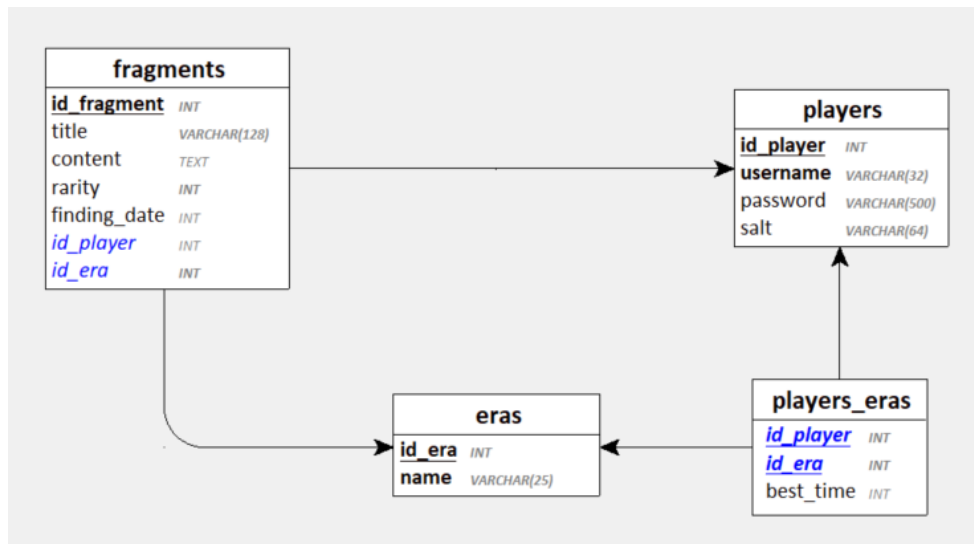
Durant ses explorations, un joueur peut trouver des fragments et les posséder. Un joueur possède (owns) donc au minimum 0 fragment, au maximum n fragments. De la même façon, un fragment peut être possédé par 0 joueur au minimum et 1 joueur au maximum. Le choix de mettre ces cardinalités s'explique par le fait que nos fragments sont des fragments uniques découvrables : un fragment ayant 0 joueur à l'avoir trouvé est un fragment trouvable qui peut apparaître à un joueur à tout moment. En revanche, chaque fragment ne peut avoir qu'un seul propriétaire en tout une fois qu'il a été trouvé.

Enfin, un fragment peuple (populates) une période historique. Il se trouve forcément dans une époque et ne peut apparaître que dans celle-ci, car l'histoire qu'il raconte est en lien avec elle. Nous sommes donc sur des cardinalités de 1,1. En revanche, une période historique peut être peuplée de 0 à n fragments. Nous avons en effet considéré que le jeu faisant apparaître des fragments de manière aléatoire aux joueurs, c'est-à-dire qu'il est possible de faire des parties sans qu'aucun fragment ne nous soit apparu, rien ne nous semblait obliger une période historique à avoir au minimum 1 fragment.

La date à laquelle le fragment a été trouvé (finding date) a été considérée comme un attribut à part entière du fragment : chaque fragment n'aura qu'une seule date à laquelle il a été trouvé et ne variera pas selon le joueur qui le consulte. De plus, lorsqu'un fragment n'a pas été trouvé, la finding date sera *null*, ce qui est un indicateur pour dire que le fragment est disponible.

Modèle logique des données

Notre MCD étant établi, nous pouvons en déduire un modèle logique des données en suivant les règles de normalisation :



Nos entités ayant toutes un identifiant unique (id_fragment, id_player et id_era) nous les faisons évoluer en clés primaires pour nos tables fragments, players et eras. L'association n : m de l'entité eras et players fait naître une table d'association players_eras qui a une clé primaire formée des clés étrangères id_player et id_era, référençant les clés primaires de players et eras. C'est elle qui contiendra la colonne best_time.

Les deux associations de type 1 : n font naître des clés étrangères côté 0,1 (fragments - players) et 1,1 (fragments - eras) : nous insérons dans fragments une clé étrangère référençant la clé primaire de players (id_player) et la clé étrangère référençant la clé primaire de eras (id_era).

La table fragments possède donc l'information de l'utilisateur qui l'a trouvé, s'il y en a un, et de l'époque où le fragment se trouve.

Mise en place de la base de données

Création de la base de données

La mise en place de la base de données a commencé par la réalisation du script de création des quatre tables identifiées par mon MLD :


```

1 CREATE TABLE players (
2     id_player INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL UNIQUE,
3     username VARCHAR(32) NOT NULL UNIQUE,
4     password VARCHAR(500) NOT NULL,
5     salt VARCHAR(64) NOT NULL
6 );
7
8 CREATE TABLE eras (
9     id_era INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL UNIQUE,
10    name VARCHAR(25) NOT NULL UNIQUE
11 );
12
13 CREATE TABLE fragments (
14     id_fragment INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL UNIQUE,
15     title VARCHAR(128) NOT NULL,
16     content TEXT NOT NULL,
17     rarity INTEGER NOT NULL DEFAULT 1,
18     finding_date INTEGER,
19     id_player INTEGER,
20     id_era INTEGER NOT NULL,
21     FOREIGN KEY(id_era) REFERENCES eras(id_era),
22     FOREIGN KEY(id_player) REFERENCES players(id_player)
23 );
24
25 CREATE TABLE players_eras (
26     id_player INTEGER NOT NULL,
27     id_era INTEGER NOT NULL,
28     best_time INTEGER,
29     CONSTRAINT id PRIMARY KEY(id_player,id_era),
30     FOREIGN KEY(id_player) REFERENCES players(id_player),
31     FOREIGN KEY(id_era) REFERENCES eras(id_era)
32 );

```

Les tables sont créées dans un ordre permettant l'intégration correcte des différentes clés étrangères. Les id constituent la clé primaire (avec auto-incrémentation), à l'exception de la table de liaison `players_eras` dont la clé primaire est composée des clés étrangères (`id_player`, `id_era`) référençant les tables `players` et `eras`.

Concernant la table **players**, nous ne voulons pas de compte utilisateur fantôme, l'intégralité des champs sont donc NOT NULL : un utilisateur doit obligatoirement avoir un username, un mot de passe et un salt. L'username est UNIQUE puisque nous ne souhaitons pas que deux joueurs aient le même pseudonyme. C'est un index.

La table **eras** est extrêmement simple et nous avons considéré qu'un VARCHAR(25) serait suffisant pour contenir le nom des époques historiques. Deux époques ne peuvent avoir le même nom, nous contraignons donc cette colonne avec UNIQUE et name devient un index également.

La table **fragments** est un petit peu plus complexe. Le titre et le contenu d'un fragment ne peuvent pas être NULL : un fragment doit raconter une histoire, sinon il n'a pas de raison d'être dans notre base de données. En revanche, `finding_date` et `id_player` peuvent l'être puisque les fragments peuvent n'avoir été découverts par personne. Ces colonnes seront NULL si le fragment est disponible et remplies si un joueur le découvre. Nous avons estimé qu'un VARCHAR(128) permettait déjà des titres très conséquents. Le contenu lui est en TEXT, car il s'agit d'un contenu écrit dont la taille peut varier significativement. Un fragment ayant forcément une rareté, le champ `rarity` est NOT NULL et nous avons défini un DEFAULT sur 1, qui est la rareté « ordinaire » du jeu. Enfin, les clés étrangères `id_era` et `id_player` liées aux tables `eras` et `players` ont été insérées conformément à notre MLD.

Concernant la **table de liaison players_eras**, nous avons le meilleur temps du joueur, `best_time`, un INTEGER qui peut être NULL dans le cas où le joueur a accès à l'époque historique, mais n'a pas encore couru dedans. Choisir NULL plutôt qu'un DEFAULT 0 reflète plus à nos yeux la réalité de la situation. Enfin, une clé primaire composée des clés étrangères insérées (`id_player`, `id_era`) et provenant des tables `players` et `eras` a été défini.

Gestion des utilisateurs

SQLite ne dispose pas de système de droits utilisateurs concernant l'accès à la BDD. Cependant, dans le cas où nous devrions effectuer une migration vers une base de données en ligne utilisant par exemple MySQL et PhpMyAdmin (voir partie conclusion), nous aurions créé trois users en plus de l'utilisateur root standard :

- Un user « `readOnly` » en lecture seule, permettant l'accès aux informations du jeu (fragments, époques etc) sans pouvoir y faire aucune modification.
- Un user « `game` », qui correspond aux droits de mon application Unity3D lorsqu'elle communique avec la BDD.
- Un user « `administrator` », possédant tous les droits sur la BDD de manière distante.

Voici comment nous aurions procédé (utilisant ici MySQL version 5.7) :

User « `readOnly` » :

```
CREATE USER 'readOnly'@'%' IDENTIFIED BY '0000';
GRANT SELECT ON fragments TO 'readOnly'@'%';
GRANT SELECT ON eras TO 'readOnly'@'%';
GRANT SELECT (id_player, username) ON players TO 'readOnly'@'%';
GRANT SELECT ON players_eras TO 'readOnly'@'%';
```

L'utilisateur classique peut accéder en lecture seule aux informations des fragments, des époques et aux id et noms des utilisateurs. Nous ne souhaitons pas qu'ils puissent voir les mots de passe et les salts même cryptés pour des raisons de sécurité.

User « `game` » :

```
CREATE USER 'game'@'%' IDENTIFIED BY 'gamePassword';
GRANT INSERT, UPDATE, SELECT, DROP ON players TO 'game'@'%';
GRANT SELECT ON fragments TO 'game'@'%';
GRANT INSERT, UPDATE (id_player, finding_date) ON fragments TO 'game'@'%';
GRANT SELECT ON eras TO 'game'@'%';
```

Le user `game` est l'utilisateur qui serait attribué à mon application Unity3D pour communiquer avec la BDD. Elle peut effectuer toutes les actions sur la table `player` car c'est elle qui gère le compte utilisateur :

inscription, connexion et demain pourrait permettre de modifier son mot de passe ou supprimer son compte. En revanche, elle ne dispose pas des droits de suppression sur les autres tables, ce qui permettra d'éviter les erreurs de requête qui entraîneraient par exemple la suppression d'un fragment. Les champs insérable et modifiable de la table fragment sont clairement définis : il s'agit uniquement des champs permettant d'indiquer qu'un fragment a été trouvé par un joueur (id_player et finding_date) L'application n'a aucune raison de venir modifier les contenus des fragments, car ils ne sont pas créés in game. De la même façon, la table eras n'est pas modifiable. En revanche tout select est possible sur fragments et eras.

User « administrator » :

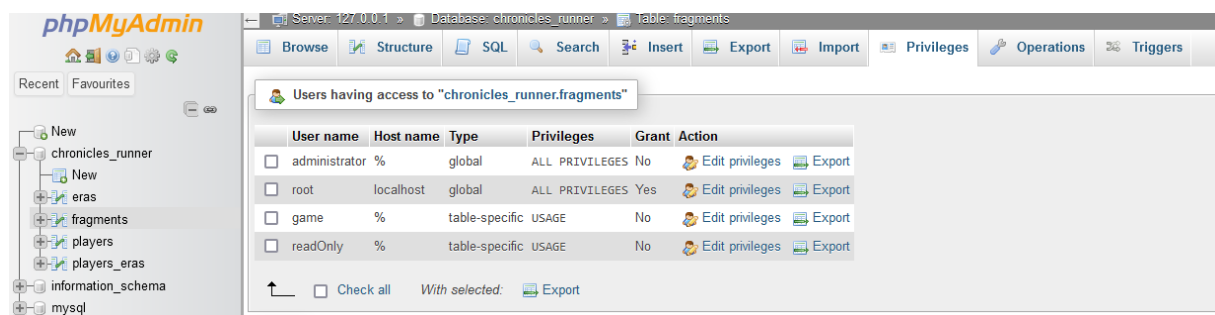
```
CREATE USER 'administrator'@'%' IDENTIFIED BY 'administratorStrongPassword';
GRANT ALL PRIVILEGES ON *.* TO 'administrator'@'%';
```

Reste la création d'un user administrateur distant, qui a beaucoup de droits et devra donc être utilisé avec parcimonie. Il doit utiliser un mot de passe fort.

User « root » :

```
CREATE USER 'root'@'localhost' IDENTIFIED BY 'rootPassword';
GRANT ALL PRIVILEGES ON *.* TO 'root'@'localhost';
```

Enfin, l'utilisateur root, qui a tous les droits. Il s'agit de l'administrateur système et il est en général défini automatiquement par le système lors d'une première installation en local.



User name	Host name	Type	Privileges	Grant	Action
<input type="checkbox"/> administrator	%	global	ALL PRIVILEGES	No	Edit privileges Export
<input type="checkbox"/> root	localhost	global	ALL PRIVILEGES	Yes	Edit privileges Export
<input type="checkbox"/> game	%	table-specific	USAGE	No	Edit privileges Export
<input type="checkbox"/> readOnly	%	table-specific	USAGE	No	Edit privileges Export

☐ Check all With selected: [Export](#)

Exemple de mise en place de ces users dans une database similaire créée en MySQL 5.7 a fin de démonstration.

Veille et recherche sur la sécurisation des identifiants et les vulnérabilités de sécurité

Puisque le jeu comporte une notion de compte utilisateur, il était très important pour moi de garantir un certain niveau de sécurité quant à leurs identifiants. Une bonne partie de mes recherches ont été consacrées à cela : après avoir consulté les recommandations en vigueur auprès de l'ANSSI et de la CNIL notamment (avec des mots clés comme « Recommandations ANSSI sécurité mots de passe », « CNIL mots de passe » ou « Password security official recommendations »), nous avons notamment veillé au respect des recommandations suivantes (« *Recommandations relatives à l'authentification multifacteur et aux mots de passe* », guide de l'ANSSI avec la participation de la CNIL) :

- Générer les éléments aléatoires avec un générateur de nombres aléatoires robuste (R5)
- Limiter la durée de validité d'une session authentifiée (R12)
- Ne pas donner l'information sur l'échec de l'authentification (R14)
- Imposer une longueur minimale pour les mots de passe (R21)
- Mise en œuvre de règles de complexité des mots de passe : imposer le recours aux caractères spéciaux et majuscules (R23)
- Utiliser un sel aléatoire long d'au moins 128 bits (R28)
- Utiliser une fonction de dérivation de mots de passe itérative pour conserver les mots de passe : PBKDF2 (R29 -)

Cela est passé par l'implémentation de quatre éléments côté applicatif :

1. Des regex pour pousser l'utilisateur à s'inscrire avec un mot de passe complexe.

```
1 référence
bool CheckUsernameFormat(string user)
{
    Regex validationRegex = new Regex(@"[a-zA-Z0-9]+$");
    return (validationRegex.IsMatch(user));
}

1 référence
bool CheckPasswordFormat(string pswd)
{
    Regex numericalCaseRegex = new Regex(@"^(?=.*[0-9])(?=.*[a-z])(?=.*[A-Z])(?!.* ).{8,}$");
    Regex specialCharCaseRegex = new Regex(@"^(?=.*[a-z])(?=.*[A-Z])(?=.*\W)(?!.* ).{8,}$");
    return (numericalCaseRegex.IsMatch(pswd) || specialCharCaseRegex.IsMatch(pswd));
}
```

2. La création d'un salt à l'inscription de l'utilisateur pour éviter les attaques avec des rainbow tables, en utilisant un système de génération de chiffre performant, ici `RandomNumberGenerator()` qui est une méthode de chiffres aléatoires adaptés à la cryptographie issue du namespace `System.Security.Cryptography` de Microsoft, sur 32 bytes, soit 256 bits.

```
3 références
public string CreateSalt()
{
    byte[] salt = new byte[32];

    using (RandomNumberGenerator rngCsp = RandomNumberGenerator.Create())
    {
        rngCsp.GetBytes(salt);
    }
    return Convert.ToBase64String(salt);
}
```

3. Le hashage de mot de passe : Le défi a donc été d'implémenter PBKDF2 dans mon jeu. Nous avons pu identifier grâce à nos recherches (mots clés « PBKDF2 C# implémentation », « PBKDF2 and Unity3D », « PBKDF2 .Net Microsoft ») une implémentation de PBKDF en C# directement depuis la documentation officielle de Microsoft : `Rfc2898DeriveBytes.PBKDF2`, qui était compatible avec mon jeu.

```

3 references
public string HashPassword(string password, string salt)
{
    var hashWithSalt = string.Format("{0}:{1}", password, salt);
    var saltBytes = Encoding.UTF8.GetBytes(hashWithSalt);
    using (var rfc2898DeriveBytes = new Rfc2898DeriveBytes(password, saltBytes, 1000))
        return Convert.ToBase64String(rfc2898DeriveBytes.GetBytes(256));
}

```

4. Mise en place de tokens : pour permettre une notion de « session » avec expiration et garantir que l'utilisateur est bien celui qu'il prétend être, nous avons eu recours à un système de jeton JWT (JSON WEB TOKEN), une référence en la matière qui disposait également d'une implémentation en C#.

```

1 référence
public string GenerateToken(int id, string username)
{
    int date = (int)DateTimeOffset.UtcNow.AddHours(2).ToUnixTimeSeconds();
    // Informations payload dans le token
    var payload = new Dictionary<string, object>
    {
        { "id", id },
        { "username", username },
        { "exp", date }
    };

    // Création de l'encodeur JWT
    IJwtAlgorithm algorithm = new HMACSHA256Algorithm();
    IJsonSerializer serializer = new JsonNetSerializer();
    IBase64UrlEncoder urlEncoder = new JwtBase64UrlEncoder();
    IJwtEncoder encoder = new JwtEncoder(algorithm, serializer, urlEncoder);

    // Génération du token
    string token = encoder.Encode(payload, DBConstant.JWTSecret);

    return token;
}

```

Ainsi, lorsque mon joueur s'inscrit, un salt lui est attribué et s'ajoute à son mot de passe avant le hashage. Lorsqu'il se connecte, ce même salt est utilisé pour hasher le mot de passe entré par l'utilisateur et vérifier la correspondance entre les deux hashages. Si c'est le cas, JWT prend le relais et crée un token stocké en local, jusqu'à ce que l'utilisateur quitte le jeu ou après un certain temps. Ce token est constitué de trois parties distinctes : un header, un payload et une signature. Dans le payload se trouvent les informations permettant d'identifier l'utilisateur, à savoir : son ID, son username et la date d'expiration du token.

Concernant ma base de données en tant que telle, nous nous sommes ensuite renseignés sur les vulnérabilités auxquelles une base de données SQLite s'exposait (avec des mots clés comme « SQLite vulnerabilities », « SQLite CVE », « How protect SQLite database ? ») et avons notamment eu recours à des requêtes préparées tout au long de notre code pour éviter les injections SQL en garantissant l'intégrité de nos requêtes et prévu la mise en place d'un système de backup.

Mise en place d'un backup

Notons que l'un des derniers points importants de la constitution de cette base de données et la mise en place d'un backup en cas de problèmes. Étant donné qu'il s'agit, à l'étape du prototype, d'une base de données SQLite située en local, la possibilité de backup est assurée par un script qui sauvegarde le fichier de BDD dans un dossier backup au lancement de l'application dans un dossier qui n'est pas celui de l'application. Ce dossier backup garde au maximum trois bases de données en mémoire.

```

Script Unity (1 référence de ressource) | 0 références
public class DatabaseBackup : MonoBehaviour
{
    private string databasePath;
    private string backupFolderPath;
    private int maxBackupCount = 3;

    Message Unity | 0 références
    private void Start()
    {
        databasePath = Application.streamingAssetsPath + "/Database/database.db";
        backupFolderPath = Application.persistentDataPath + "/Database_saves/";
        BackupDatabase();
    }

    1 référence
    private void BackupDatabase()
    {
        try
        {
            if (!Directory.Exists(backupFolderPath))
            {
                Directory.CreateDirectory(backupFolderPath);
            }

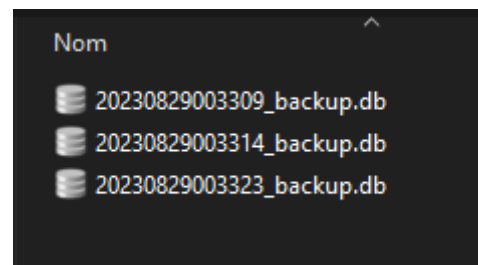
            string backupFileName = DateTime.Now.ToString("yyyyMMddHhmmss") + "_backup.db";
            string backupFilePath = Path.Combine(backupFolderPath, backupFileName);

            File.Copy(databasePath, backupFilePath);

            CleanupBackups();

            Debug.Log("Backup successful: " + backupFileName);
        } catch (Exception e)
        {
            Debug.LogError("Backup failed: " + e.Message);
        }
    }
}

```



Extrait du script permettant le backup de la base de données

Développement des composants

Déclencheurs ou Triggers

Deux triggers ont été mis en place pour assurer le bon fonctionnement du jeu :

Un trigger permettant d'attribuer automatiquement une période historique explorable à un nouvel inscrit, afin qu'un joueur ait obligatoirement au moins une époque dans laquelle courir :

```

CREATE TRIGGER t_assign_era_to_new_player AFTER INSERT ON players
BEGIN
    INSERT INTO players_eras (id_player, id_era, best_time)
    VALUES (NEW.id_player, (SELECT id_era FROM eras ORDER BY RANDOM() LIMIT 1), NULL);
END;

```

Et un trigger permettant de supprimer proprement un joueur de la base de données en « libérant » ses fragments et effaçant ses traces de la table players_eras.

```

CREATE TRIGGER t_delete_player BEFORE DELETE ON players
BEGIN
    DELETE FROM players_eras WHERE id_player = OLD.id_player;
    UPDATE fragments SET id_player = NULL, finding_date = NULL WHERE id_player = OLD.id_player;
END;

```

Conformément au gameplay du jeu, nous avons envisagé de mettre en place un trigger pour attribuer une nouvelle époque explorable à un joueur lorsqu'il a au moins 2 fragments dans chaque époque actuellement explorable pour lui. Cependant, nous avons mis cette hypothèse de côté pour une meilleure flexibilité, estimant qu'il était préférable d'avoir cette logique côté application, afin qu'un game designer puisse faire évoluer plus facilement les conditions de déblocage d'une époque historique au besoin.

Vues ou views

Dans le cadre de notre jeu, les vues sont très intéressantes pour faciliter le traitement des fragments d'histoire et leur rapport avec les joueurs ou avec une époque. Plusieurs vues ont été établies pour notre jeu : une vue permettant de lister tous les fragments par joueur, une vue permettant de lister toutes les périodes historiques explorables pour joueur, une vue permettant d'afficher tous les fragments ayant été trouvés dans une époque et une vue permettant d'avoir toutes les informations nécessaires pour l'affichage in game d'un fragment.

À titre d'exemple, voici la vue permettant de lister tous les fragments d'un joueur, concaténés dans une colonne appelée « list_fragments ».

```
CREATE VIEW v_fragments_by_player AS
SELECT players.id_player, players.username,
       GROUP_CONCAT(fragments.id_fragment) AS list_fragments
FROM players
LEFT JOIN fragments ON players.id_player = fragments.id_player
GROUP BY players.id_player;
```

	id_player	username	list_fragments
1	3	Simon	29,33,40,114,118,164,165
2	4	Tom	NULL
3	5	GamingCampus	113,121
4	6	Mimi	NULL

Et la vue permettant d'afficher les informations nécessaires pour l'affichage in game d'un fragment :

```
CREATE VIEW v_fragment_info AS
SELECT fragments.id_fragment, fragments.title, fragments.content, fragments.rarity, fragments.finding_date,
       players.id_player, players.username, eras.id_era, eras.name
FROM fragments
JOIN players ON fragments.id_player = players.id_player
JOIN eras ON fragments.id_era = eras.id_era;
```

	id_fragment	title	content	rarity	finding_date	id_player	username	id_era	name
1	29	Histoires dans la Pierre	"C'est grâce à ces Indiens que Jackson découvrit ...	1	1692657671	3	Simon	2	far west
2	33	Flamme de Vérité	"Les vents du Far West murmuraient les histoire...	1	1692657671	3	Simon	2	far west
3	40	Héritage Envolé	"Et ainsi, les cristaux du Far West furent dispers...	1	1692657666	3	Simon	2	far west
4	42	L'Enigme de la Lame d'Héritage	"Les gravures énigmatiques ornant une antique ...	1	1693261111	23	Harfang	3	médiéval...
5	52	Éclats d'une Symphonie Inachevée	"Des fragments cristallins captaient la tourmente ...	1	1693261111	23	Harfang	3	médiéval...

En l'absence de procédure stockée en raison de l'utilisation de SQLite, cette vue est principalement utilisée pour afficher plus rapidement tous les fragments d'un joueur à une époque donnée à l'aide d'un WHERE.

```

1 référence
public List<DBFragment> GetAllCurrentUserFragments()
{
    int userID = sessionManager.UserData.CurrentUserId;
    List<DBFragment> allFragments = db.Query<DBFragment>("SELECT * FROM v_fragment_info WHERE id_player = ? ", userID);
    return allFragments;
}

```

Procédures stockées ou Stored procedures

Comme indiqué précédemment, l'utilisation de SQLite ne nous a pas permis d'utiliser les procédures stockées durant ce projet. Néanmoins, il est certain qu'elles auraient été utiles, notamment pour trier les fragments que l'on souhaite afficher dans le jeu. Puisque nous envisageons un jour un portage vers MySQL, nous sommes en mesure de nous projeter sur les procédures stockées qui pourraient être intéressantes.

Afficher les fragments pour une époque et un joueur donnés dans un ordre voulu :

```

DELIMITER //
CREATE PROCEDURE GetFragmentsByEraAndUser(
    IN eraID INT,
    IN userID INT,
    IN orderBy VARCHAR(10)
)
BEGIN
    IF orderBy = 'asc' THEN
        SELECT * FROM fragments
        WHERE id_era = eraID AND id_player = userID
        ORDER BY finding_date ASC;
    ELSEIF orderBy = 'desc' THEN
        SELECT * FROM fragments
        WHERE id_era = eraID AND id_player = userID
        ORDER BY finding_date DESC;
    ELSEIF orderBy = 'random' THEN
        SELECT * FROM fragments
        WHERE id_era = eraID AND id_player = userID
        ORDER BY RAND();
    END IF;
END //
DELIMITER ;

```

Cette procédure permettrait d'afficher des fragments de manière randomisés ou non à une époque et un utilisateur donné, par ordre croissant ou décroissant. Aujourd'hui, cela est géré directement dans Unity3D. Avec la procédure stockée, nous pourrions par exemple sélectionner tous les fragments de l'utilisateur id 26 dans l'époque id 6 en affichant les derniers trouvés en premier, ou le faire dans un ordre aléatoire simplement en changeant un paramètre (« desc » et « random ») :

Showing rows 0 - 2 (3 total. Query took 0.0005 seconds.) CALL GetFragmentsByEraAndUser(6, 26, "desc"); [Edit inline] [Edit] [Create PHP code]					
Show all Number of rows: 25 Filter rows: Search this table					
Extra options					
id_fragment	title	content	rarity	finding_date	id_player id_era
11	Les Toiles Magiques d'Hiroshi	Parmi les histoires entrelacées, celle d'un peintre...	1	1693261542	26 6
12	Les Destins Entremêlés	Les époques s'entrelaçaient, les histoires fusionn...	1	1693261401	26 6
10	Lueur Enigmatique	"Le souffle du passé était palpable. Les images an...	1	1692655963	26 6

Showing rows 0 - 2 (3 total. Query took 0.0011 seconds.) CALL GetFragmentsByEraAndUser(6, 26, "random"); [Edit inline] [Edit] [Create PHP code]					
Show all Number of rows: 25 Filter rows: Search this table					
Extra options					
id_fragment	title	content	rarity	finding_date	id_player id_era
10	Lueur Enigmatique	"Le souffle du passé était palpable. Les images an...	1	1692655963	26 6
12	Les Destins Entremêlés	Les époques s'entrelaçaient, les histoires fusionn...	1	1693261401	26 6
11	Les Toiles Magiques d'Hiroshi	Parmi les histoires entrelacées, celle d'un peintre...	1	1693261542	26 6

Afficher le meilleur joueur dans chaque époque :

```
DELIMITER //
CREATE PROCEDURE GetTopPlayersByEra(
    IN eraID INT,
    IN topNB INT
)
BEGIN
    SELECT players.id_player, players.username, players_eras.best_time,
    COUNT(fragments.id_fragment) AS num_fragments,
    (COUNT(fragments.id_fragment) / GREATEST(players_eras.best_time, 1)) AS score
    FROM players
    JOIN players_eras ON players.id_player = players_eras.id_player AND players_eras.id_era = eraID
    LEFT JOIN fragments ON players.id_player = fragments.id_player AND fragments.id_era = eraID
    GROUP BY players.id_player, players.username, players_eras.best_time
    ORDER BY score DESC
    LIMIT topNB;
END //
DELIMITER ;
```

Cette procédure stockée nous aurait également été utile pour faire un classement plus complexe que celui actuellement en place (meilleur joueur par nombre de fragments) en affichant le joueur ayant la meilleure « moyenne » entre son meilleur temps dans l'époque et le nombre de fragments qu'il y a récolté.

Showing rows 0 - 2 (3 total, Query took 0.0008 seconds.)				
CALL GetTopPlayersByEra(6,3);				
[Edit inline] [Edit] [Create PHP code]				
Show all Number of rows: 25 Filter rows: Search this table				
Extra options				
id_player	username	best_time	num_fragments	score
26	Jack	45	3	0.0667
22	Caro	134	4	0.0299
21	Arnaud	32	0	0.0000

Showing rows 0 - 2 (3 total, Query took 0.0004 seconds.)				
CALL GetTopPlayersByEra(6,3);				
[Edit inline] [Edit] [Create PHP code]				
Show all Number of rows: 25 Filter rows: Search this table				
Extra options				
id_player	username	best_time	num_fragments	score
22	Caro	134	4	0.0299
26	Jack	105	3	0.0286
21	Arnaud	32	0	0.0000

Exemple d'utilisation de la procédure stockée où l'on voit que selon la différence de temps entre Caro et Jack, le classement diffère, car le score global évolue.

Situation de travail ayant nécessité des recherches

Comme indiqué précédemment, une grande partie de nos recherches ont été consacrées à des questions de sécurité avec le hashage PBKDF2 et les tokens JWT (voir partie concernée sur la sécurité pour plus de détails).

Néanmoins, d'autres recherches ont été nécessaires tout au long du projet. Durant la phase de conception, nous avons dû nous assurer des différents types pris en compte par SQLite, car nous savions qu'ils étaient assez restreints et voulions choisir les bons types dès le départ. Nous avons également dû consulter la documentation de SQLite lorsque certaines requêtes devenaient un peu complexes, notamment lorsque nous avons besoin d'y ajouter de la randomisation, afin de nous assurer de l'ordre dans lesquelles faire les choses ou pour connaître l'ordre exact pour insérer des sous-requêtes dans certaines requêtes un peu complexes. De ce point de vue, le site officiel de SQLite a été d'un grand secours.

Un exemple concret serait l'intégration de PBKDF2 dans notre projet : nous avons commencé par nous informer en nous rendant sur les sites de l'ANSSI et du CNIL qui permettent d'obtenir les meilleures recommandations. A la lecture de leur document, nous avons estimé que PBKDF2 ou Bcrypt semblaient être des pistes intéressantes. Après avoir découvert au travers de différents forums anglophones que PBKDF2 était plus facile à implémenter et plus conseillé pour du C#, une seconde phase a consisté à tenter de comprendre ce dont il s'agissait réellement, en passant par différents sites comme le site de Microsoft ou des blogs spécialisés. Ce sont par ces derniers que nous avons essayé de comprendre dans un troisième temps comment cela devait s'implémenter dans notre application, puis nous avons poussé cette compréhension à l'étude de codes déjà implémentés dans des projets jugés comme étant de confiance, comme des projets d'Azure ou de Microsoft via GitHub. Nous avons également consulté d'autres projets GitHub et fait un regroupement de toutes ces informations pour déterminer la bonne façon de le faire, puis nous l'avons implémenté dans notre projet.



Conclusion

La base de données que nous avons imaginée est désormais opérationnelle et remplit parfaitement son rôle au sein du prototype de jeu que nous avons réalisé. Pour le moment, notre objectif est d'effectuer des tests utilisateurs sur le prototype existant afin de réorienter le jeu au besoin et, dans ce contexte, SQLite est plutôt pertinent en raison de sa flexibilité. Néanmoins, il s'agit d'une base de données locale qui est donc propre au poste qui l'utilise : elle n'est pas mise à jour par les actions des autres joueurs distants. Or, puisqu'il s'agit d'un jeu collaboratif souhaitant proposer aux joueurs de trouver des « trésors cachés » avant les autres, cela semble plus que nécessaire. Par ailleurs, dans un jeu de cette nature, il est assez aisé d'imaginer devoir gérer à terme une très importante quantité de fragments. Cela nous amène à dire que, sur du long terme, le recours à une base de données en local via SQLite n'est pas l'idéal. Nous envisageons donc d'effectuer une transition vers MySQL pour sa mise en production. Cette transition devra passer par :

1. Une analyse des spécificités de MySQL par rapport à SQLite.
2. L'adaptation éventuelle des types de données en veillant à leur intégrité (MySQL ayant plus de types et étant plus contraignant).
3. La création de la base de données sur MySQL via PhpMyAdmin.
4. La migration des données via export/import CSV.
5. Une transposition des composants SQLite dans MySQL en adaptant les scripts de création de vues et de triggers et en ajoutant des procédures stockées).
6. Une adaptation de la syntaxe côté Unity3D prise en compte des procédures stockées et intégration du dialogue avec PHP pour pouvoir échanger avec le serveur distant.
7. Des tests du jeu pour vérifier que tout est en place avant de le distribuer.

Une fois connecté à une base de données en ligne, *Chronicles Runner* pourra prendre une tout autre dimension et atteindre son véritable potentiel.

