ATELIER PROFESSIONNEL

MEDIATEK DOCUMENTS

COMPTE-RENDU RÉALISÉ PAR:
DROAI LOUIZA
BTS SIO-SLAM 2ÈME ANNÉE

CONTEXTE

MediaTek86, responsable de la gestion des médiathèques de la Vienne ainsi que de la coordination des emprunts de livres, DVD et CD, s'engage également dans le développement de la médiathèque numérique pour toutes les médiathèques du département. Pour assurer une gestion efficace du catalogue et des commandes de documents, MediaTek86 a délégué la conception d'une application de gestion à InfoTechServices 86.

MISSION

L'objectif de cet atelier est de faire progresser une application de bureau développée en C#, qui exploite une API REST en PHP pour accéder à une base de données relationnelle MySQL. Cette application vise à gérer les documents et les commandes des médiathèques affiliées à MediaTek86. Un premier développeur a déjà contribué à la création de la base de données et au développement de certaines fonctionnalités de l'application.

Cette application de bureau est destinée à être installée sur plusieurs postes dans les médiathèques, tous accédant à la même base de données. L'objectif est d'étendre ses fonctionnalités existantes pour inclure la gestion du catalogue des médiathèques ainsi que des commandes de documents.

SOMMAIRE

MISSION 1: GÉRER LES DOCUMENTS

A-FONCTIONNALITÉ “AJOUTER”

1-FONCTIONNALITÉ “AJOUTER” -ONGLET LIVRES
2-FONCTIONNALITÉ “AJOUTER” -ONGLET DVD
3-FONCTIONNALITÉ “AJOUTER” -ONGLET REVUES

B-FONCTIONNALITÉ “MODIFIER”

1-FONCTIONNALITÉ “MODIFIER”- ONGLET LIVRES
2-FONCTIONNALITÉ “MODIFIER”- ONGLET DVD
3-FONCTIONNALITÉ “MODIFIER”- ONGLET REVUES

C-FONCTIONNALITÉ “SUPPRIMER”

1-FONCTIONNALITÉ “SUPPRIMER”- ONGLET LIVRES
2-FONCTIONNALITÉ “SUPPRIMER”- ONGLET DVD
3-FONCTIONNALITÉ “SUPPRIMER”- ONGLET REVUES

MISSION 2: GÉRER LES COMMANDES

TACHE 1: GÉRER LES COMMANDES DE LIVRES OU DE DVD

TACHE 2: GÉRER LES COMMANDES DE REVUES

MISSION 3: GÉRER L’ÉTAT DES EXEMPLAIRES

A. ETAT EXEMPLAIRES - ONGLET LIVRES

B. ETAT EXEMPLAIRES - ONGLET DVD

C. ETAT EXEMPLAIRES - ONGLET REVUES

MISSION 4: METTRE EN PLACE DES AUTHENTIFICATIONS

A. AJOUTER UNE TABLE “USER” -BDD

B. FONCTIONNALITÉ D’AUTHENTIFICATION

MISSION 5: ASSURER LA SÉCURITÉ,LA QUALITÉ ET INTÉGRER LES LOGS

TACHE 1: CORRIGER DES PROBLÈMES DE SÉCURITÉ

TACHE 2: CONTRPLER LA QUALITÉ

TACHE 3: INTÉGRER DES LOGS

MISSION 6: TESTER ET DOCUMENTER

TACHE 1: GÉRER LES TESTS

TACHE 2: DOCUMENTATIONS TECHNIQUES

TACHE 3: DOCUMENTATION UTILISATEUR

MISSION 7: DÉPLOYER ET GÉRER LES SAUVEGARDES DE DONNÉES

TACHE 1: DÉPLOYER LE PROJET

TACHE 2: GÉRER LES SAUVEGARDES DES DONNÉES

BILAN

LANGAGES ET OUTILS

LANGAGES DE PROGRAMMATION

C#
PHP
GHERKIN

SERVEUR

WAMPSERVER:
APACHE
MYSQL

FRAMEWORK

.NET FRAMEWORK

TEST

POSTMAN
SPECFLOW

VERSIONNING

GITHUB

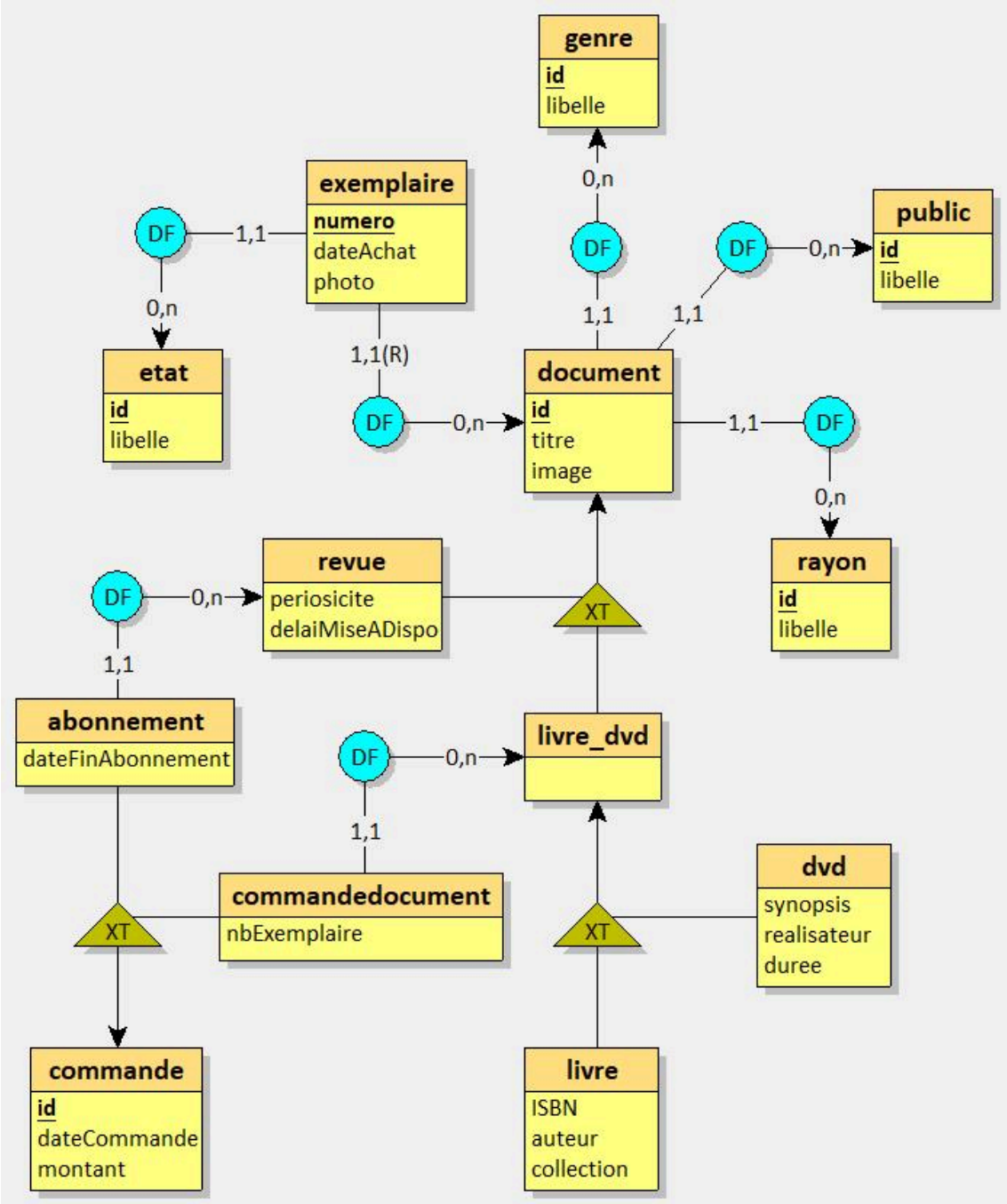
IDE

NETBEANS
VISUAL STUDIO 2019

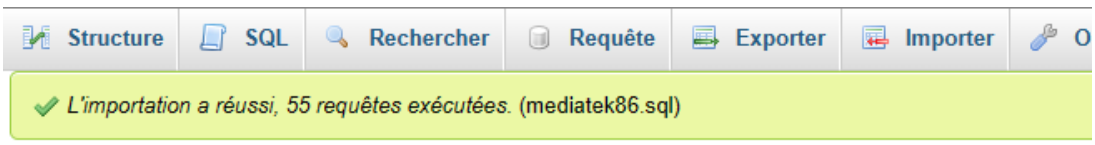
Hébergeur

HOSTINGER

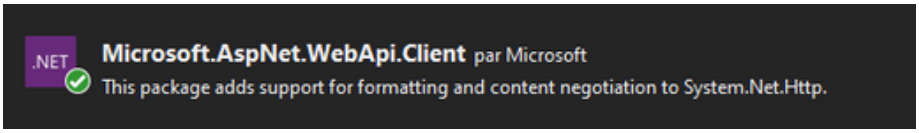
MISSION 0 : PRÉPARER L'ENVIRONNEMENT DE TRAVAIL

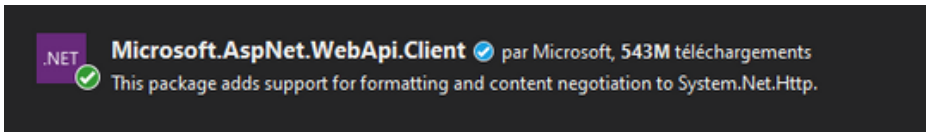


Import de la base de données avec Wamp et PhpMyAdmin.

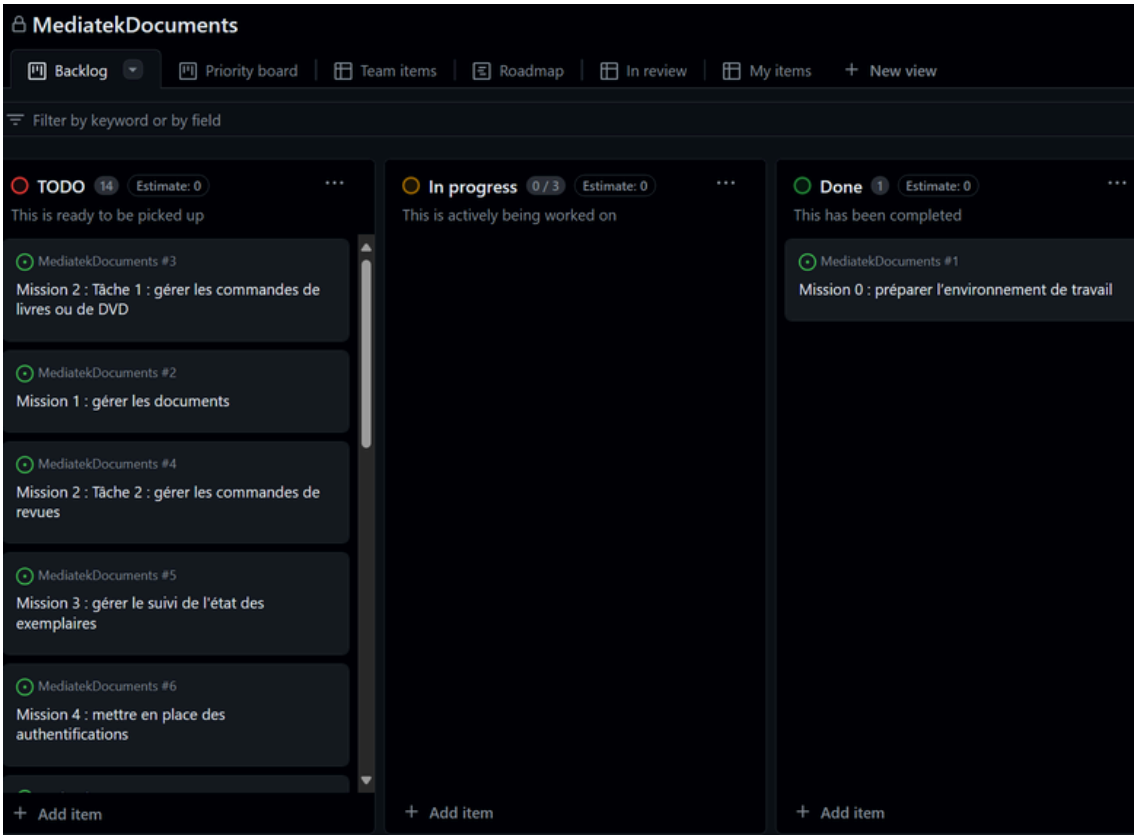


Installation des nugets nécessaire à l'application.





Création du dépôt distant sur Github, mise en place du projet Kanban.

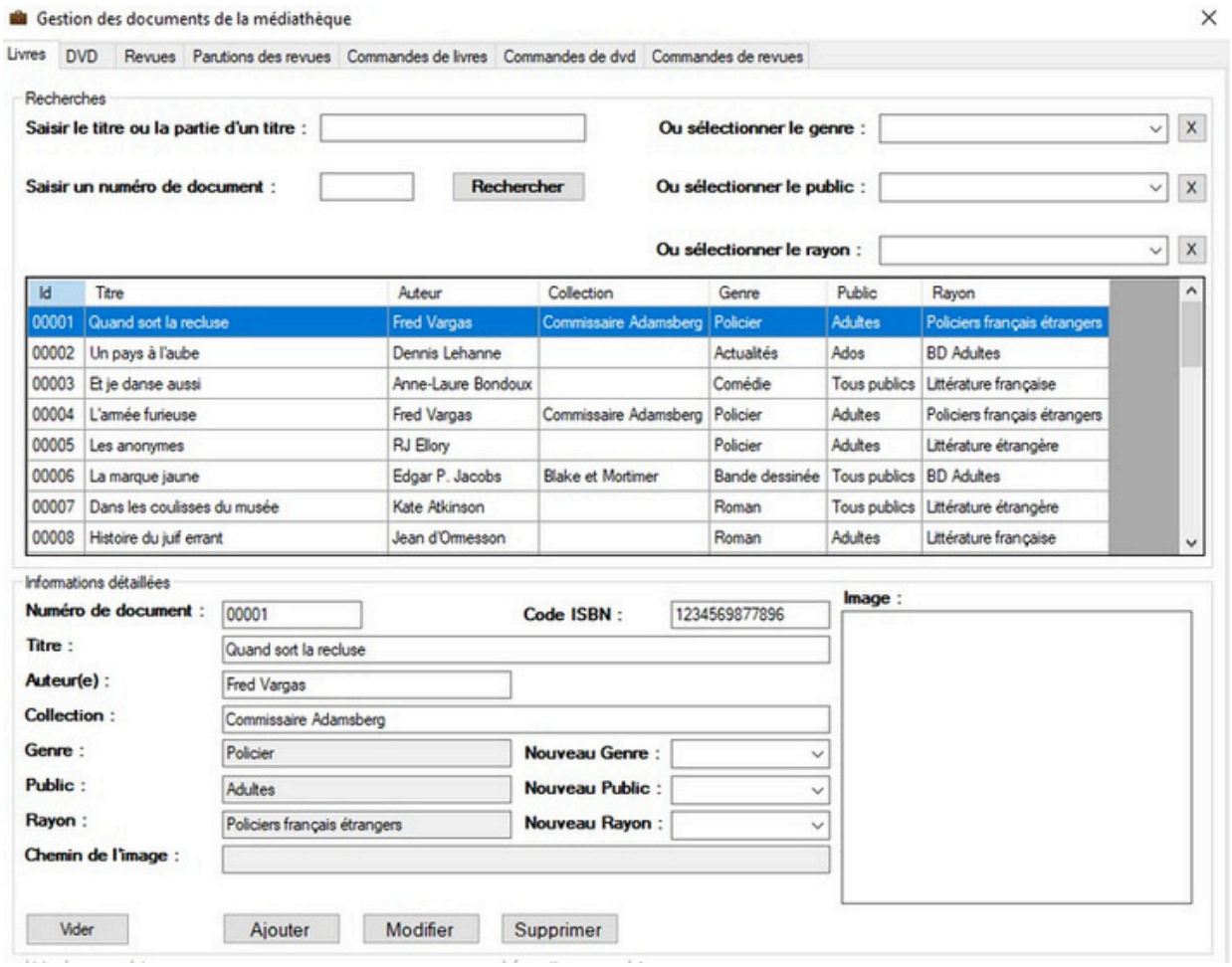


MISSION 1 : GERER LES DOCUMENTS

1-FONCTIONNALITÉ “AJOUTER” -ONGLET LIVRES

Je débute en créant les boutons "Ajouter", "Modifier" et "Supprimer" dans la vue des onglets "livres", "dvd" et "revues". Pour cela, je me dirige vers la vue "FrmMediatek.cs" et je choisis l'option "button" dans la boîte à outils de Visual Studio. Ensuite, j'insère trois boutons correspondant à chaque action, positionnés sous les détails de chaque type de document dans les onglets respectifs.

Pendant ce processus, j'ajuste également le paramètre "ReadOnly" des champs pour permettre à l'utilisateur d'entrer des valeurs en le passant à "false". De plus, je rajoute trois comboBox afin que l'utilisateur puisse sélectionner un nouveau genre, public et rayon, pour les associer à un document qu'il souhaite ajouter ou modifier.



MÉTHODE- CREERDOCUMENT

L'objectif est d'ajouter la fonctionnalité "ajouter" dans l'onglet "livres" de l'application. Cela nécessite l'utilisation de l'API REST PHP avec la méthode "Insert One" accessible via la méthode "POST" en C#. Les données saisies par l'utilisateur sont envoyées au format JSON à l'API.

Dans la classe "Access.cs", une fonction nommée "CreerDocument" est créée pour ajouter les informations d'un document à la base de données. Les données sont sérialisées en JSON et envoyées à l'API, qui les ajoute à la table "document" en utilisant la route "document/" du point de terminaison.

```
public bool CreerDocument(string Id, string Titre, string Image, string IdRayon, string IdPublic, string IdGenre)
{
    String jsonCreerDocument = "{ \"id\" : \"" + Id + "\", \"titre\" : \"" + Titre + "\", \"image\" : \"" + Image + "\", \"idRayon\" : \"" + IdRayon + "\", \"idPublic\" : \"" + IdPublic + "\", \"idGenre\" : \"" + IdGenre + "\"}";
    Console.WriteLine("jsonCreerDocument" + jsonCreerDocument);
    try
    {
        // récupération soit d'une liste vide (requête ok) soit de null (erreur)
        List<Document> liste = TraitementRecup<Document>(POST, "document/" + jsonCreerDocument);
        return (liste != null);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
    return false;
}
```

Dans le contrôleur "FrmMediatekController.cs", une méthode "CreerDocument" est créée pour ajouter un document à la table "document". Cette méthode utilise la fonction correspondante de la classe "Access.cs" et retourne un booléen pour indiquer le succès de l'opération.

Pour étendre cette fonctionnalité et permettre l'ajout d'un nouveau "livre" à la table "livre", une méthode "CreerLivre" est mise en place dans la classe "Access.cs". Cette méthode sérialise les données en JSON et les envoie via une requête "POST" vers le point de terminaison "livre/" de l'API.

MÉTHODE- CREERLIVRE

```
public bool CreerLivre(string Id, string Isbn, string Auteur, string Collection)
{
    String jsonCreerLivre = "{ \"id\" : \"" + Id + "\", \"isbn\" : \"" + Isbn + "\", \"auteur\" : \"" + Auteur + "\", \"collection\" : \"" + Collection + "\"}";
    Console.WriteLine("jsonCreerLivre" + jsonCreerLivre);
    try
    {
        // récupération soit d'une liste vide (requête ok) soit de null (erreur)
        List<Document> liste = TraitementRecup<Document>(POST, "livre/" + jsonCreerLivre);
        return (liste != null);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
    return false;
}
```

Une fois réalisée, je dois appeler cette fonction dans le contrôleur, j'insère donc la méthode "CreerLivre(string Id, string Isbn, string Auteur, string Collection)", à la classe "FrmMediatekController.cs".

```
public bool CreerLivre(string Id, string Isbn, string Auteur, string Collection)
{
    return access.CreerLivre(Id, Isbn, Auteur, Collection);
}
```

Dans la vue "FrmMediatek.cs", j'ai développé une fonctionnalité pour faciliter l'ajout simultané d'un document et d'un livre lorsqu'un utilisateur clique sur le bouton "Ajouter".

- Lorsque l'utilisateur inscrit le numéro du document dans le champ "txbLivresNumero", le système capture les données saisies dans les champs "informations détaillées" pour créer un nouveau document et un nouveau livre avec un ID correspondant.

- Les méthodes "CreerDocument" et "CreerLivre" sont utilisées pour intégrer le nouveau document à la base de données.

- Cette fonctionnalité actualise l'affichage de la base de données avec les livres et leurs informations respectives.

Pour garantir la sécurité des saisies, j'ai incorporé des comboBox permettant de spécifier le genre, le public et le rayon du document.

- Des méthodes telles que "RemplirCbxAjoutModifGenreLivre()", "RemplirCbxAjoutModifPublicLivre()", et "RemplirCbxAjoutModifRayonLivre()" ont été élaborées pour alimenter les comboBox avec les catégories pertinentes, en utilisant la méthode "GetAll" pour récupérer les données nécessaires.

- Ces méthodes suivent une structure uniforme et modifient les noms des comboBox en fonction de leur usage pour assurer une cohérence dans l'interface utilisateur.

MÉTHODE - REMPLIRCBXAJOUTMODIFGENRELIVRE()

```
private string RemplirCbxAjoutModifGenreLivre()
{
      List lesGenresLivres = controller.GetAllGenres();
      foreach (Categorie genre in lesGenresLivres)
         {
           cbxAjoutModifGenreLivre.Items.Add(genre.Libelle);
         }
      if (cbxAjoutModifGenreLivre.Items.Count > 0)
         {
            cbxAjoutModifGenreLivre.SelectedIndex = 0;
         }
     return cbxAjoutModifGenreLivre.SelectedItem?.ToString();
}
```

Pour faciliter la récupération des IDs correspondant aux genres, publics et rayons sélectionnés par l'utilisateur, j'ai mis en place trois méthodes distinctes dans mon application. Chacune de ces méthodes, nommément "GetIdGenreDocument", "GetIdPublicDocument" et "GetIdRayonDocument", fonctionne de manière similaire. Elles parcourent la liste complète des catégories à l'aide de la méthode "GetAll" et utilisent une structure conditionnelle "if" pour identifier le libellé sélectionné dans les comboBox respectives. Une fois le libellé trouvé, la méthode retourne l'ID correspondant. Cette approche garantit une récupération précise des IDs associés aux différentes catégories de documents, assurant ainsi la cohérence et la fiabilité des données dans l'application.

MÉTHODE - GETIDGENREDOCUMENT

```
private string GetIdGenreDocument(string genre)
{
      List lesGenresDocument = controller.GetAllGenres();
       foreach (Categorie c in lesGenresDocument)
         {
            if (c.Libelle == genre)
            {
                return c.Id;
            }
         }
        return null;
}
```

Dans la procédure événementielle associée au clic sur le bouton "Ajouter" de l'onglet "livres", j'ai implémenté plusieurs mesures de sécurité supplémentaires. Ces mesures garantissent que l'utilisateur saisit tous les champs obligatoires pour ajouter un livre, et empêchent l'ajout d'un livre avec un ID déjà existant dans la base de données, grâce à une condition "if(idLivreNonExistant)".
De plus, j'ai inclus des affectations des méthodes "GetId" dans les propriétés "idGenre", "idPublic" et "idRayon" lors de cette procédure. Sans ces affectations, la requête ne récupérerait pas les IDs correspondants aux libellés, entraînant un dysfonctionnement.

PROCÉDURE ÉVÈNEMENTIELLE - BTNRECEPTIONLIVREAJOUTER_CLICK

```
Document document = new Document(id, titre, image, idGenre, genre, idPublic, Public, idRayon, rayon);
Livre livre = new Livre(id, titre, image, isbn, auteur, collection, idGenre, genre, idPublic, Public, idRayon,
rayon);

var idLivreExistant = controller.GetAllDocuments(id);
var idLivreNonExistant = !idLivreExistant.Any();

if (idLivreNonExistant)
{
   if (controller.CreerDocument(document.Id, document.Titre, document.Image, document.IdRayon,
document.IdPublic, document.IdGenre) && controller.CreerLivre(livre.Id, livre.Isbn, livre.Auteur,
livre.Collection))
   {
      lesLivres = controller.GetAllLivres();
      RemplirComboCategorie(controller.GetAllGenres(), bdgGenres, cbxLivresGenres);
      RemplirComboCategorie(controller.GetAllPublics(), bdgPublics, cbxLivresPublics);
      RemplirComboCategorie(controller.GetAllRayons(), bdgRayons, cbxLivresRayons);
      RemplirLivresListeComplete();
      MessageBox.Show("Le livre " + titre + " a correctement été ajouté.");
   }
}
else
{
   MessageBox.Show("Le numéro de document existe déjà, veuillez saisir un autre numéro.", "Erreur");
}
```

Lors de la mise en place de la fonctionnalité d'ajout sur l'onglet "livres", j'ai identifié une limitation : il n'était pas possible d'ajouter un nouveau livre si son ID n'existait pas déjà dans les tables "livres_dvd" et "document". Pour résoudre cette contrainte, j'ai développé un déclencheur (trigger) qui automatise l'ajout de l'ID du livre dans la table "livres_dvd" dès la création d'un nouveau livre.

TRIGGER - AJOUT LIVRE

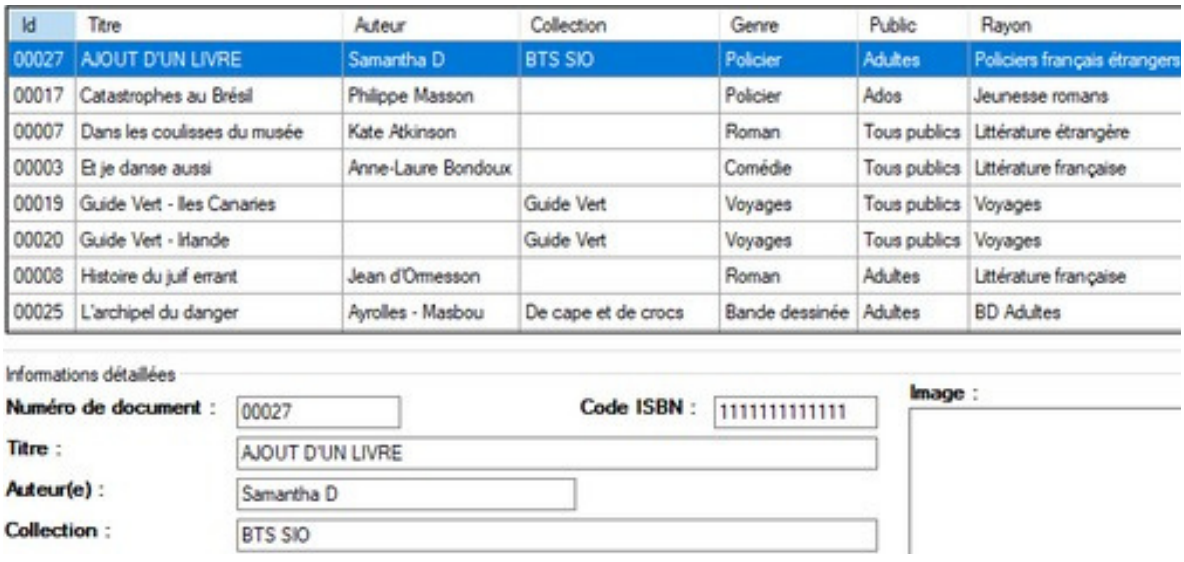
```
DELIMITER $$
CREATE TRIGGER `insert_livre_to_livres_dvd` BEFORE INSERT ON `livre`
FOR EACH ROW BEGIN
  DECLARE cnt INT;
  SELECT COUNT(*) INTO cnt
  FROM livres_dvd
  WHERE id = NEW.id;

  IF cnt = 0 THEN
    INSERT INTO livres_dvd (id) VALUES (NEW.id);
  END IF;
END$$
DELIMITER ;
```

Étant donné que l'ID du nouveau livre doit être présent dans la table "document" avant d'être ajouté dans la table "livres_dvd", j'ai développé un deuxième déclencheur. Ce déclencheur assure l'ajout de l'ID du livre dans la table "document" avant son insertion dans la table "livres_dvd", établissant ainsi le lien entre les IDs de ces deux tables.

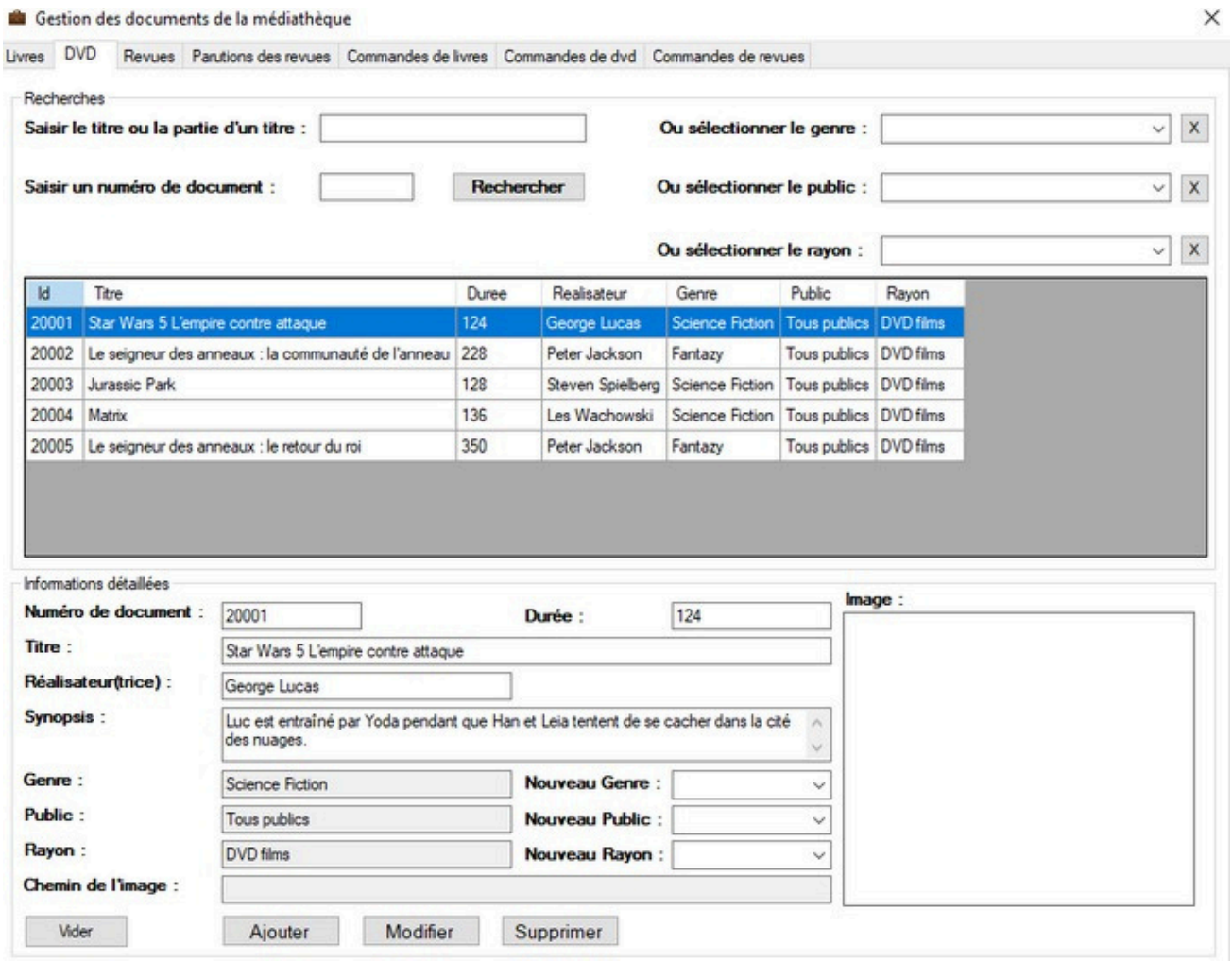
TRIGGER - AJOUT LIVRE_DVD

```
DELIMITER $$
CREATE TRIGGER `insert_livres_dvd_to_document` BEFORE INSERT ON `livres_dvd`
FOR EACH ROW BEGIN
  DECLARE cnt INT;
  SELECT COUNT(*) INTO cnt
  FROM document
  WHERE id = NEW.id;

  IF cnt = 0 THEN
    INSERT INTO document (id) VALUES (NEW.id);
  END IF;
END$$
DELIMITER ;
```

Je teste ensuite le fonctionnement de mon bouton "ajouter"



2-FONCTIONNALITÉ “AJOUTER” -ONGLET DVD



Pour permettre l'ajout d'un nouveau DVD à la table "dvd" en plus d'un document avec la méthode "CreerDocument" déjà implémentée dans la classe "Access", j'ai créé une nouvelle méthode "CreerDvd(string Id, string Synopsis, string Realisateur, int Duree)" dans la même classe. Cette méthode prend en paramètres les différentes valeurs nécessaires pour un DVD, qui seront saisies par l'utilisateur dans les champs des informations détaillées. Les données sont ensuite sérialisées dans une chaîne JSON appelée "jsonCreerDvd" et envoyées via la méthode "POST" vers l'endpoint "dvd/". Cela permet d'étendre la fonctionnalité d'ajout pour inclure à la fois des documents et des DVD dans la base de données.

MÉTHODE CREERDVD

```
public bool CreerDvd(string Id, string Synopsis, string Realisateur, int Duree)
{
   string jsonCreerDvd = "{ \"id\" : \"" + Id + "\", \"synopsis\" : \"" + Synopsis + "\", \"realisateur\" : \"" +
                  Realisateur + "\", \"duree\" : \"" + Duree + "\"}";
   Console.WriteLine("jsonCreerDvd" + jsonCreerDvd);
   try
   {
      List<Dvd> liste = TraitementRecup<Dvd>(POST, "dvd/" + jsonCreerDvd);
      return (liste != null);
   }
   catch (Exception ex)
   {
      Console.WriteLine(ex.Message);
   }
   return false;
}
```

Après avoir implémenté la méthode CreerDvd dans la classe Access, je l'ai également ajoutée au contrôleur FrmMediatekController pour pouvoir l'appeler depuis la vue FrmMediatek.cs. Cette dernière contient une procédure événementielle nommée btnReceptionDvdValider_Click, activée lors du clic sur le bouton "Ajouter". Lorsque l'utilisateur entre le numéro du document dans le champ txbDvdNumero, les données saisies dans chaque champ sont récupérées et insérées à la fois dans un nouveau document et un nouveau DVD, partageant le même ID. Les méthodes CreerDocument et CreerDvd sont alors appelées pour ajouter ces éléments à la base de données. Ensuite, la base de données mise à jour est affichée, incluant les DVD et les informations associées aux documents. Pour que l'ajout de DVD fonctionne, des modifications sont apportées à la fonction "SelectOne" dans l'API rest_mediatekdocuments pour inclure la gestion des DVD. Enfin, afin de remplir les comboBox de l'onglet DVD avec les listes des catégories correspondantes, des méthodes telles que RemplirCbxAjoutModifGenreDvd, RemplirCbxAjoutModifPublicDvd et RemplirCbxAjoutModifRayonDvd sont construites, adaptant les appels des méthodes GetAll en conséquence.

MÉTHODE - REMPLIRCBXAJOUTMODIFGENREDVD()

```
private string RemplirCbxAjoutModifGenreDvd()
{
    List<Categorie> lesGenresDvd = controller.GetAllGenres();
    foreach (Categorie genre in lesGenresDvd)
    {
        cbxAjoutModifGenreDvd.Items.Add(genre.Libelle);
    }
    if (cbxAjoutModifGenreDvd.Items.Count > 0)
    {
        cbxAjoutModifGenreDvd.SelectedIndex = 0;
    }
    return cbxAjoutModifGenreDvd.SelectedItem?.ToString();
}
```

Dans la procédure d'ajout d'un DVD, des mesures de sécurité sont mises en place pour garantir l'intégrité des données. L'utilisateur doit remplir tous les champs obligatoires, et l'ajout est empêché si l'ID du DVD est déjà présent dans la base de données. De plus, les méthodes "GetId" sont utilisées pour obtenir les IDs correspondants aux libellés des catégories, assurant le bon déroulement des requêtes.

PROCÉDURE ÉVÈNEMENTIELLE - BTNRECEPTIONDVDVALIDER_CLICK

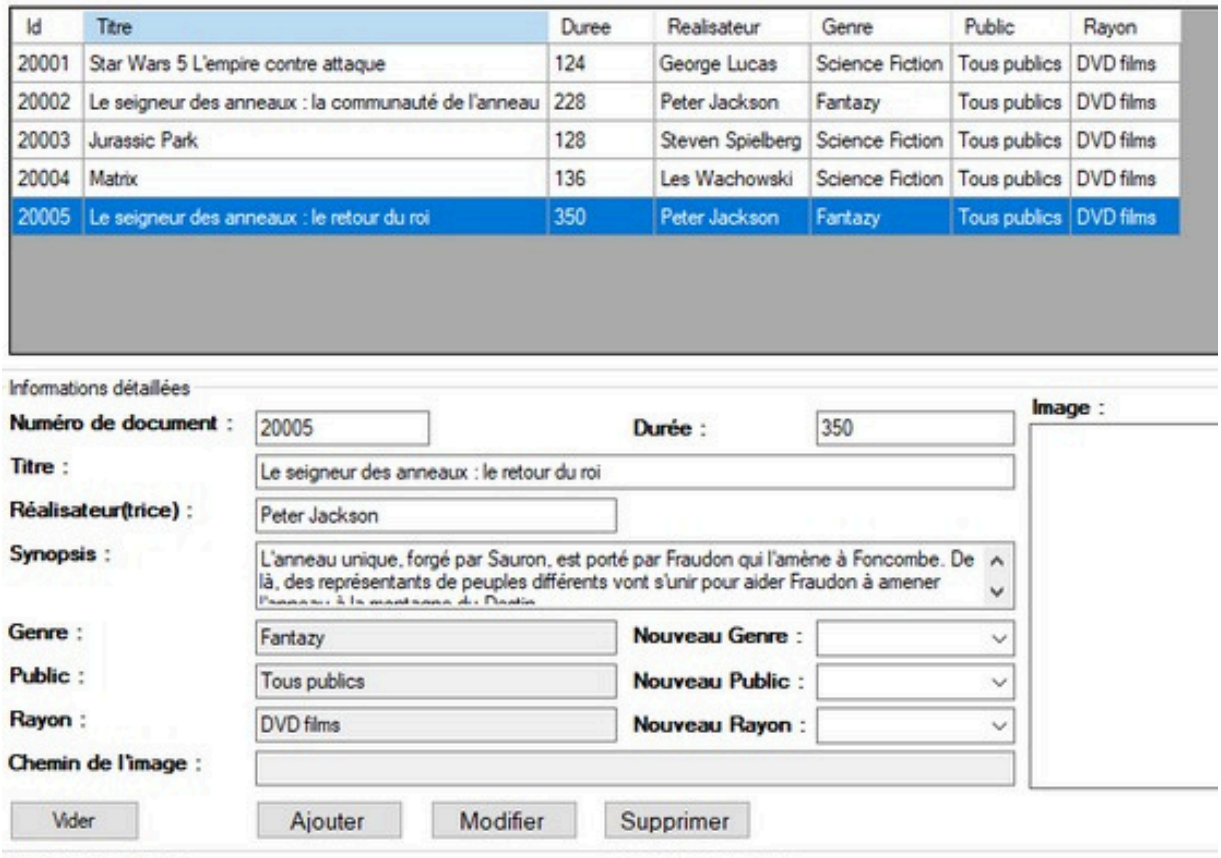
```
string idGenre = GetIdGenreDocument(cbxAjoutModifGenreDvd.Text);
string genre = txbDvdGenre.Text;
string idPublic = GetIdPublicDocument(cbxAjoutModifPublicDvd.Text);
string lePublic = txbDvdPublic.Text;
string idRayon = GetIdRayonDocument(cbxAjoutModifRayonDvd.Text);
string rayon = txbDvdRayon.Text;

Document document = new Document(id, titre, image, idGenre, genre, idPublic, lePublic, idRayon, rayon);
Dvd dvd = new Dvd(id, titre, image, duree, realisateur, synopsis, idGenre, genre, idPublic, lePublic, idRayon,
rayon);

var idDvdExistant = controller.GetAllDocuments(id);
var idDvdNonExistant = !idDvdExistant.Any();

if (idDvdNonExistant)
{
    if (controller.CreerDocument(document.Id, document.Titre, document.Image, document.IdRayon,
document.IdPublic, document.IdGenre) &&
        controller.CreerDvd(dvd.Id, dvd.Synopsis, dvd.Realisateur, dvd.Duree))
    {
        lesDvd = controller.GetAllDvd();
        RemplirComboCategorie(controller.GetAllGenres(), bdgGenres, cbxDvdGenres);
        RemplirComboCategorie(controller.GetAllPublics(), bdgPublics, cbxDvdPublics);
        RemplirComboCategorie(controller.GetAllRayons(), bdgRayons, cbxDvdRayons);
        RemplirDvdListeComplete();
        MessageBox.Show("Le dvd " + titre + " a correctement été ajouté.");
    }
}
else
{
    MessageBox.Show("Le numéro de document existe déjà.", "Erreur");
}
```

Lors de l'ajout de fonctionnalités à l'onglet "DVD", j'ai découvert qu'il était crucial d'empêcher l'ajout d'un nouveau DVD si son ID n'était pas déjà présent dans la table "livres_dvd". Pour résoudre ce problème, j'ai implémenté un trigger qui automatise l'ajout de l'ID du DVD dans cette table dès sa création.

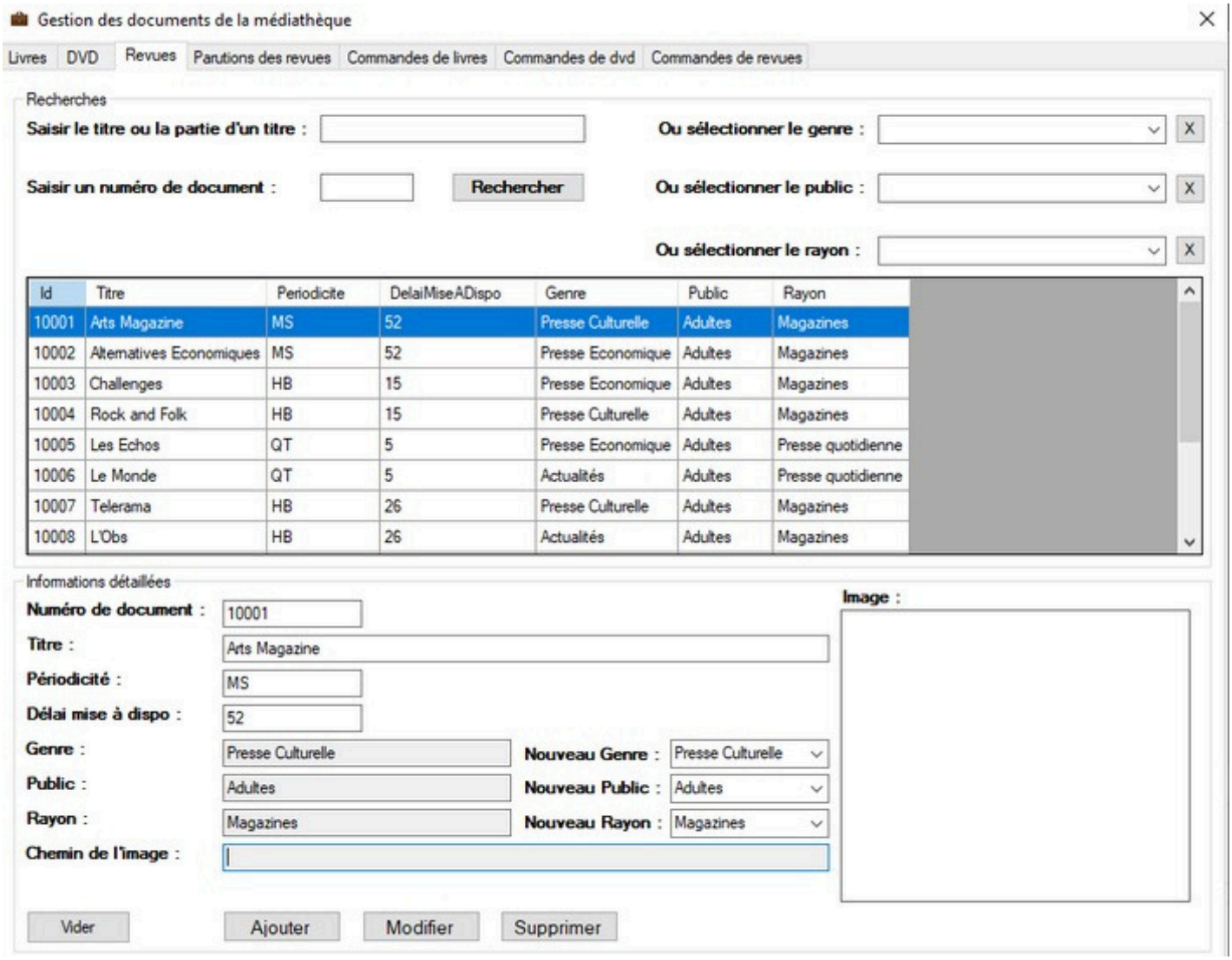
TRIGGER – AJOUT DVD

```
DELIMITER $$
CREATE TRIGGER `insert_dvd_to_livres_dvd` BEFORE INSERT ON `dvd`
FOR EACH ROW BEGIN
  IF NOT EXISTS (SELECT id FROM livres_dvd WHERE id = NEW.id) THEN
    INSERT INTO livres_dvd (id) VALUES (NEW.id);
  END IF;
END$$
DELIMITER ;
}
```

Je teste ensuite la fonctionnalité "ajouter un dvd" dans l'onglet "dvd"



3-FONCTIONNALITÉ “AJOUTER” -ONGLET REVUES



Pour que le bouton "Ajouter" puisse également insérer une nouvelle "revue" dans la table "revue", je développe une nouvelle fonction nommée "CreerRevue(string Id, string Periodicite, int DelaiMiseADispo)" dans la classe "Access.cs". Cette fonction prend en paramètres les différentes informations nécessaires à la création d'une revue, telles que l'identifiant, la périodicité et le délai de mise à disposition. Ces données sont ensuite converties en une chaîne JSON nommée "jsonCreerRevue" et envoyées via une requête POST vers l'endpoint "revue/" pour être ajoutées à la base de données.

MÉTHODE - CREERREVUE

```
public bool CreerRevue(string Id, string Periodicite, int DelaiMiseADispo)
{
    String jsonCreerRevue = "{ \"id\" : \"" + Id + "\", \"periodicite\" : \"" + Periodicite + "\", \"delaiMiseADispo\" : \"" + DelaiMiseADispo + "\"}";
    Console.WriteLine("jsonCreerRevue" + jsonCreerRevue);
    try
    {
        // récupération soit d'une liste vide (requête ok) soit de null (erreur)
        List<Revue> liste = TraitementRecup<Revue>(POST, "revue/" + jsonCreerRevue);
        return (liste != null);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
    return false;
}
```

Une fois réalisée, je dois appeler cette fonction dans le contrôleur

```
public bool CreerRevue(string Id, string Periodicite, int DelaiMiseADispo)
{
    return access.CreerRevue(Id, Periodicite, DelaiMiseADispo);
}
```

Dans la vue "FrmMediatek.cs", j'ai mis en place la procédure d'ajout d'un document et d'une revue lors du clic sur le bouton "Ajouter". Dans la méthode "btnReceptionRevueValider_Click", lorsque l'utilisateur entre le numéro du document dans le champ "txbRevueNumero", les données saisies sont récupérées et utilisées pour créer à la fois un nouveau document et une nouvelle revue, qui auront le même identifiant. L'ajout à la base de données se fait grâce aux méthodes "CreerDocument" et "CreerRevue". Ensuite, la base de données est mise à jour pour afficher les nouvelles revues et les informations du document associées. Pour que l'ajout d'une revue fonctionne, j'ai modifié la fonction "SelectOne" dans l'API REST PHP pour inclure le cas "revue". Les comboBox du genre, du public et du rayon sont remplis de la même manière que dans les onglets précédents, à l'aide des méthodes "RemplirCbxAjoutModifGenreRevue()", "RemplirCbxAjoutModifPublicRevue()" et "RemplirCbxAjoutModifRayonRevue()". Enfin, j'ai également ajouté des mesures de sécurité pour garantir l'intégrité des données.

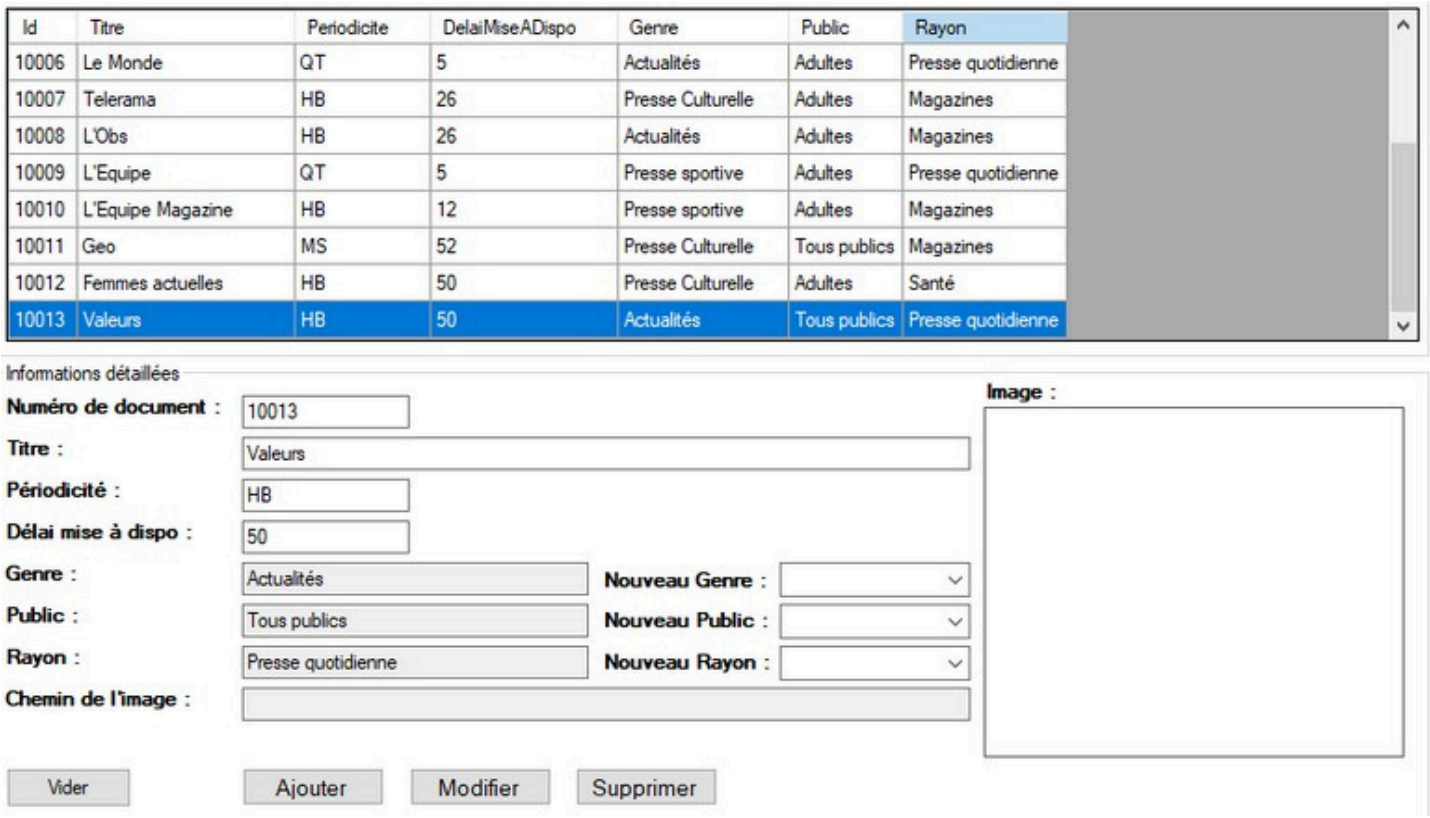
PROCÉDURE ÉVÈNEMENTIELLE - BTNRECEPTIONREVUEVALIDER_CLICK

```
string idRayon = txbIdRayonRevue.Text;
string rayon = txbRevuesRayon.Text;
string periodicite = txbRevuesPeriodicite.Text;
int delaiMiseADispo = int.Parse(txbRevuesDateMiseADispo.Text);

Document document = new Document(id, titre, image, idGenre, genre, idPublic, lePublic, idRayon, rayon);
Revue revue = new Revue(id, titre, image, idGenre, genre, idPublic, lePublic, idRayon, rayon, periodicite, delaiMiseADispo);
```

En développant la fonctionnalité d'ajout sur l'onglet "revue", j'ai remarqué qu'il était impossible d'ajouter une nouvelle revue si son ID n'existait pas déjà dans la table "document". Pour résoudre cette contrainte, j'ai mis en place un déclencheur (trigger) qui ajoute l'ID de la revue dans la table "document" lors de la création d'une nouvelle revue.

```
DELIMITER $$
CREATE TRIGGER `insert_revue_to_document` BEFORE INSERT ON `revue`
FOR EACH ROW BEGIN
  DECLARE cnt INT;
  SELECT COUNT(*) INTO cnt
  FROM document
  WHERE id = NEW.id;
  IF cnt = 0 THEN
    INSERT INTO document (id) VALUES (NEW.id);
  END IF;
END$$
DELIMITER ;
```

Je teste ensuite la fonctionnalité "Ajouter une revue"



B-FONCTIONNALITÉ “MODIFIER”

1-FONCTIONNALITÉ “MODIFIER”- ONGLET LIVRES

1. Pour intégrer le bouton "modifier" dans l'onglet "livres", j'utilise la fonction de l'API REST MediatekDocuments appelée "update One", accessible dans l'application C# via la méthode "PUT". Cette méthode permet de récupérer les données modifiées par l'utilisateur dans l'interface de l'application sur les différents onglets, puis de sauvegarder ces modifications de la ligne sélectionnée dans la base de données, en les envoyant au format JSON à l'API.

Je commence par créer une méthode de modification des informations d'un document dans la classe "Access.cs", nommée "ModifierDocument(string Id, string Titre, string Image, string IdGenre, string IdPublic, string IdRayon)", où les paramètres correspondent aux champs de la table "document". Les données sont ensuite sérialisées en JSON dans une chaîne "jsonModifierDocument", suivie d'un affichage console pour vérifier le résultat. Les données sont envoyées à l'API via la méthode "PUT" pour enregistrer les modifications dans la table "document", avec le endpoint "document/" et l'ID du document concerné.
Ensuite, j'appelle cette méthode dans le contrôleur via une méthode "ModifierDocument" dans la classe "FrmMediatekController.cs".
Pour permettre également la modification des informations d'un livre dans la table "livre" via le bouton "Modifier", je crée une fonction "ModifierLivre(string Id, string Isbn, string Auteur, string Collection)" dans la classe "Access.cs". Les données sont sérialisées en JSON dans une chaîne "jsonModifierLivre" et envoyées à l'API via la méthode "PUT" avec le endpoint "livre/id".
Après avoir créé ces méthodes, je les appelle dans le contrôleur via une méthode "ModifierLivre" dans la classe "FrmMediatekController.cs".
Enfin, je crée une procédure évènementielle "btnReceptionLivreModifier_Click" dans la vue "FrmMediatek.cs" pour gérer la modification d'un document et d'un livre lors du clic sur le bouton "Modifier". Les données saisies par l'utilisateur sont affectées au document sélectionné et au livre associé à ce document. Les méthodes "ModifierDocument" et "ModifierLivre" sont ensuite appelées pour effectuer les modifications dans la base de données. La mise à jour de l'affichage est réalisée ensuite, suivie d'un message de confirmation ou d'erreur.

```
if (controller.ModifierLivre(id, isbn, auteur, collection) &&
controller.ModifierDocument(id, titre, image, idGenre, idPublic, idRayon))
{
    lesLivres = controller.GetAllLivres();
    RemplirComboCategorie(controller.GetAllGenres(), bdgGenres, cbxLivresGenres);
    RemplirComboCategorie(controller.GetAllPublics(), bdgPublics, cbxLivresPublics);
    RemplirComboCategorie(controller.GetAllRayons(), bdgRayons, cbxLivresRayons);
    RemplirLivresListeComplete();
    MessageBox.Show("Le livre " + titre + " a correctement été modifié.");
}
else
{
    MessageBox.Show("Erreur lors de la modification du livre", "Erreur");
}
```

2-FONCTIONNALITÉ “MODIFIER”- ONGLET DVD

Pour que le bouton "Modifier" puisse également modifier un "dvd" sélectionné dans la table "dvd", je crée une nouvelle méthode "ModifierDvd(string Id, string Synopsis, string Realisateur, int Duree)" dans la classe "Access.cs". Cette méthode prend en paramètres les différents champs de la table "dvd". Les données sont sérialisées en JSON dans une chaîne "jsonModifierDvd", puis envoyées via la méthode "PUT" au endpoint "dvd/id".
Ensuite, j'appelle cette méthode dans le contrôleur via une méthode "ModifierDvd" dans la classe "FrmMediatekController.cs".
Dans la vue "FrmMediatek.cs", je crée une procédure évènementielle "btnReceptionDvdModifier_Click" pour gérer la modification d'un document et d'un dvd lors du clic sur le bouton "Modifier". Lorsqu'une ligne est sélectionnée dans la datagrid "dgvDvdListe.SelectedRows.Count > 0", les données saisies par l'utilisateur sont affectées au document sélectionné et au dvd associé à ce document. Les méthodes "ModifierDocument" et "ModifierDvd" sont ensuite appelées pour effectuer les modifications dans la base de données. La mise à jour de l'affichage est réalisée ensuite, suivie d'un message de confirmation.

```
if (controller.ModifierDvd(id, synopsis, realisateur, duree) &&
controller.ModifierDocument(id, titre, image, idGenre, idPublic, idRayon))
{
    lesDvd = controller.GetAllDvd();
    RemplirComboCategorie(controller.GetAllGenres(), bdgGenres, cbxDvdGenres);
    RemplirComboCategorie(controller.GetAllPublics(), bdgPublics, cbxDvdPublics);
    RemplirComboCategorie(controller.GetAllRayons(), bdgRayons, cbxDvdRayons);
    RemplirDvdListeComplete();
    MessageBox.Show("Le dvd " + titre + " a correctement été modifié.");
}
```

3-FONCTIONNALITÉ “MODIFIER”- ONGLET REVUES

Je crée une nouvelle méthode "ModifierRevue" dans la classe "Access.cs", prenant en paramètres les champs de la table "revue". Les données sont sérialisées en JSON dans une chaîne "jsonModifierRevue" et envoyées via la méthode "PUT" au endpoint "revue/id".
Ensuite, j'appelle cette méthode dans le contrôleur via une méthode "ModifierRevue" dans la classe "FrmMediatekController.cs".
Dans la vue "FrmMediatek.cs", je crée une procédure évènementielle "btnReceptionRevueModifier_Click" pour gérer la modification d'un document et d'une revue lors du clic sur le bouton "Modifier". Lorsqu'une ligne est sélectionnée dans la datagrid "dgvRevuesListe.SelectedRows.Count > 0", les données saisies par l'utilisateur sont affectées au document sélectionné et à la revue associée à ce document. Les méthodes "ModifierDocument" et "ModifierRevue" sont ensuite appelées pour effectuer les modifications dans la base de données. La mise à jour de l'affichage est réalisée ensuite, suivie d'un message de confirmation.

```
if (controller.ModifierRevue(id, periodicite, delaiMiseADispo) &&
controller.ModifierDocument(id, titre, image, idGenre, idPublic, idRayon))
{
   lesRevues = controller.GetAllRevues();
   RemplirComboCategorie(controller.GetAllGenres(), bdgGenres, cbxLivreGenres);
   RemplirComboCategorie(controller.GetAllPublics(), bdgPublics, cbxLivrePublics);
   RemplirComboCategorie(controller.GetAllRayons(), bdgRayons, cbxLivreRayons);
   RemplirRevuesListeComplete();
   MessageBox.Show("La revue " + titre + " a correctement été modifiée.");
}
```

C-FONCTIONNALITÉ “SUPPRIMER”

1-FONCTIONNALITÉ “SUPPRIMER”- ONGLET LIVRES

Pour mettre en place la fonctionnalité "supprimer" dans l'onglet "livres", j'utilise la fonction "delete" de l'API REST MediatekDocuments, accessible via la méthode "DELETE" en C#.
La méthode DELETE permet de supprimer les données associées à l'ID sélectionné par l'utilisateur dans l'interface de l'application, en envoyant les données au format JSON à l'API.
Je commence par créer la méthode "SupprimerLivre" dans la classe "Access.cs". Cette méthode prend l'ID du livre à supprimer comme paramètre. Les données sont sérialisées en JSON et envoyées à l'API via la méthode DELETE.
Ensuite, j'appelle cette méthode dans le contrôleur via une méthode "SupprimerLivre" dans la classe "FrmMediatekController.cs".
Dans la vue "FrmMediatek.cs", je crée une procédure événementielle "btnSupprimerLivre_Click" pour gérer la suppression d'un livre lors du clic sur le bouton "Supprimer". L'ID du livre sélectionné par l'utilisateur est récupéré, puis les données de la ligne sélectionnée sont supprimées de la base de données. La mise à jour de l'affichage est réalisée ensuite, suivie d'un message de confirmation.

```
private void btnSupprimerLivre_Click(object sender, EventArgs e)
{
   Livre livre = (Livre)bdgLivresListe.Current;
   if (MessageBox.Show("Souhaitez-vous confirmer la suppression?", "Confirmation de
suppression", MessageBoxButtons.OKCancel) == DialogResult.OK)
   {
      var exemplairesLivre = controller.GetExemplairesDocument(livre.Id);
      var aucunExemplaire = !exemplairesLivre.Any();
      var commandesLivre = controller.GetCommandesDocument(livre.Id);
      var aucuneCommande = !commandesLivre.Any();
      if (aucunExemplaire && aucuneCommande)
      {
         if (controller.SupprimerLivre(livre.Id))
         {
            lesLivres = controller.GetAllLivres();
            RemplirComboCategorie(controller.GetAllGenres(), bdgGenres,
cbxLivresGenres);
            RemplirComboCategorie(controller.GetAllPublics(), bdgPublics,
cbxLivresPublics);
            RemplirComboCategorie(controller.GetAllRayons(), bdgRayons,
cbxLivresRayons);
            RemplirLivresListeComplete();
            MessageBox.Show($"Le livre {livre.Titre} a correctement été supprimé.");
         }
         else
         {
            MessageBox.Show("Une erreur s'est produite.", "Erreur");
         }
      }
      else
      {
         MessageBox.Show($"Impossible de supprimer le livre car il possède
{(aucunExemplaire ? "une ou plusieurs commande(s)" : "un ou plusieurs
exemplaire(s)")}.", "Erreur");
      }
   }
}
```

Une condition empêche la suppression si le livre possède des exemplaires ou des commandes. Des messages d'erreurs s'affichent pour informer l'utilisateur.
Enfin, des déclencheurs (triggers) sont mis en place pour supprimer l'ID du livre dans les tables associées (livres_dvd et document) lorsqu'un livre est supprimé. Cela assure la cohérence des données entre les différentes tables.

2-FONCTIONNALITÉ “SUPPRIMER”- ONGLET DVD

Pour mettre en place la fonctionnalité "supprimer" dans l'onglet "DVD", je crée une nouvelle fonction appelée "SupprimerDvd" dans la classe "Access.cs". Cette fonction prend l'ID du DVD à supprimer en paramètre. Ensuite, l'ID est envoyé sous forme de chaîne JSON appelée "jsonIdDvd" et supprimé avec la méthode DELETE sur le endpoint dvd/id.

```
// Fonction permettant la suppression d'un DVD
public bool SupprimerDvd(string Id)
{
   String jsonIdDvd = "{ \"id\" : \"" + Id + "\"}";
   Console.WriteLine("jsonIdDvd" + jsonIdDvd);
   try
   {
      // récupération soit d'une liste vide (requête ok) soit de null (erreur)
      List<Dvd> liste = TraitementRecup<Dvd>(DELETE, "dvd/" + jsonIdDvd);
      return (liste != null);
   }
   catch (Exception ex)
   {
      Console.WriteLine(ex.Message);
   }
   return false;
}
```

Ensuite, j'appelle cette fonction dans le contrôleur "FrmMediatekController.cs" via une méthode "SupprimerDvd".
Dans la vue "FrmMediatek.cs", je crée une procédure événementielle "btnSupprimerDvd_Click" pour gérer la suppression d'un DVD lors du clic sur le bouton "Supprimer". L'ID du DVD sélectionné par l'utilisateur est récupéré, puis les données de la ligne sélectionnée sont supprimées de la base de données. La mise à jour de l'affichage est ensuite réalisée, suivie d'un message de confirmation.

```
private void btnSupprimerDvd_Click(object sender, EventArgs e)
{
   Dvd dvd = (Dvd)bdgDvdListe.Current;
   if (MessageBox.Show("Souhaitez-vous confirmer la suppression?", "Confirmation de
suppression", MessageBoxButtons.OKCancel) == DialogResult.OK)
   {
      var exemplairesDvd = controller.GetExemplairesDocument(dvd.Id);
      var aucunExemplaireDvd = !exemplairesDvd.Any();
      var commandesDvd = controller.GetCommandesDocument(dvd.Id);
      var aucuneCommandeDvd = !commandesDvd.Any();
      if (aucunExemplaireDvd && aucuneCommandeDvd)
      {
         if (controller.SupprimerDvd(dvd.Id))
         {
            lesDvd = controller.GetAllDvd();
            RemplirComboCategorie(controller.GetAllGenres(), bdgGenres, cbxDvdGenres);
            RemplirComboCategorie(controller.GetAllPublics(), bdgPublics,
cbxDvdPublics);
            RemplirComboCategorie(controller.GetAllRayons(), bdgRayons,
cbxDvdRayons);
            RemplirDvdListeComplete();
            MessageBox.Show("Le DVD " + dvd.Titre + " a correctement été supprimé.");
         }
         else
         {
            MessageBox.Show("Une erreur s'est produite.", "Erreur");
         }
      }
   }
}
```

Une condition empêche la suppression si le DVD possède des exemplaires ou des commandes. Des messages d'erreurs s'affichent pour informer l'utilisateur.
Cette fonctionnalité suit le même principe que la suppression d'un livre, en ajoutant une condition pour empêcher la suppression si le DVD possède des exemplaires ou des commandes.

3-FONCTIONNALITÉ “SUPPRIMER”- ONGLET REVUES

Pour ajouter la fonctionnalité "supprimer" dans l'onglet "Revues", je crée une nouvelle méthode appelée "SupprimerRevue" dans la classe "Access.cs". Cette méthode prend l'ID de la revue à supprimer en paramètre. Ensuite, l'ID est envoyé sous forme de chaîne JSON appelée "jsonIdRevue", puis supprimé avec la méthode DELETE sur le endpoint revue/id.

```
// Méthode permettant la suppression d'une revue
public bool SupprimerRevue(string Id)
{
   String jsonIdRevue = "{ \"id\" : \"" + Id + "\"}";
   Console.WriteLine("jsonIdRevue" + jsonIdRevue);
   try
   {
      // récupération soit d'une liste vide (requête ok) soit de null (erreur)
      List<Revue> liste = TraitementRecup<Revue>(DELETE, "revue/" + jsonIdRevue);
      return (liste != null);
   }
   catch (Exception ex)
   {
      Console.WriteLine(ex.Message);
   }
   return false;
}
```

Ensuite, j'appelle cette fonction dans le contrôleur "FrmMediatekController.cs" via une méthode "SupprimerRevue".
Dans la vue "FrmMediatek.cs", je crée une procédure événementielle "btnSupprimerRevue_Click" pour gérer la suppression d'une revue lors du clic sur le bouton "Supprimer". L'ID de la revue sélectionnée par l'utilisateur est récupéré, puis les données de la ligne sélectionnée sont supprimées de la base de données. Une condition est ajoutée pour éviter la suppression si la revue possède des exemplaires associés.

```
List<Exemplaire> lesExemplairesRevue = controller.GetExemplairesRevue(revue.Id);
bool aucunExemplaire = true;
foreach (Exemplaire exemplaire in lesExemplairesRevue)
{
   if (exemplaire.Id == revue.Id)
   {
      aucunExemplaire = false;
      break;
   }
}
```

MISSION 2 : GERER LES COMMANDES

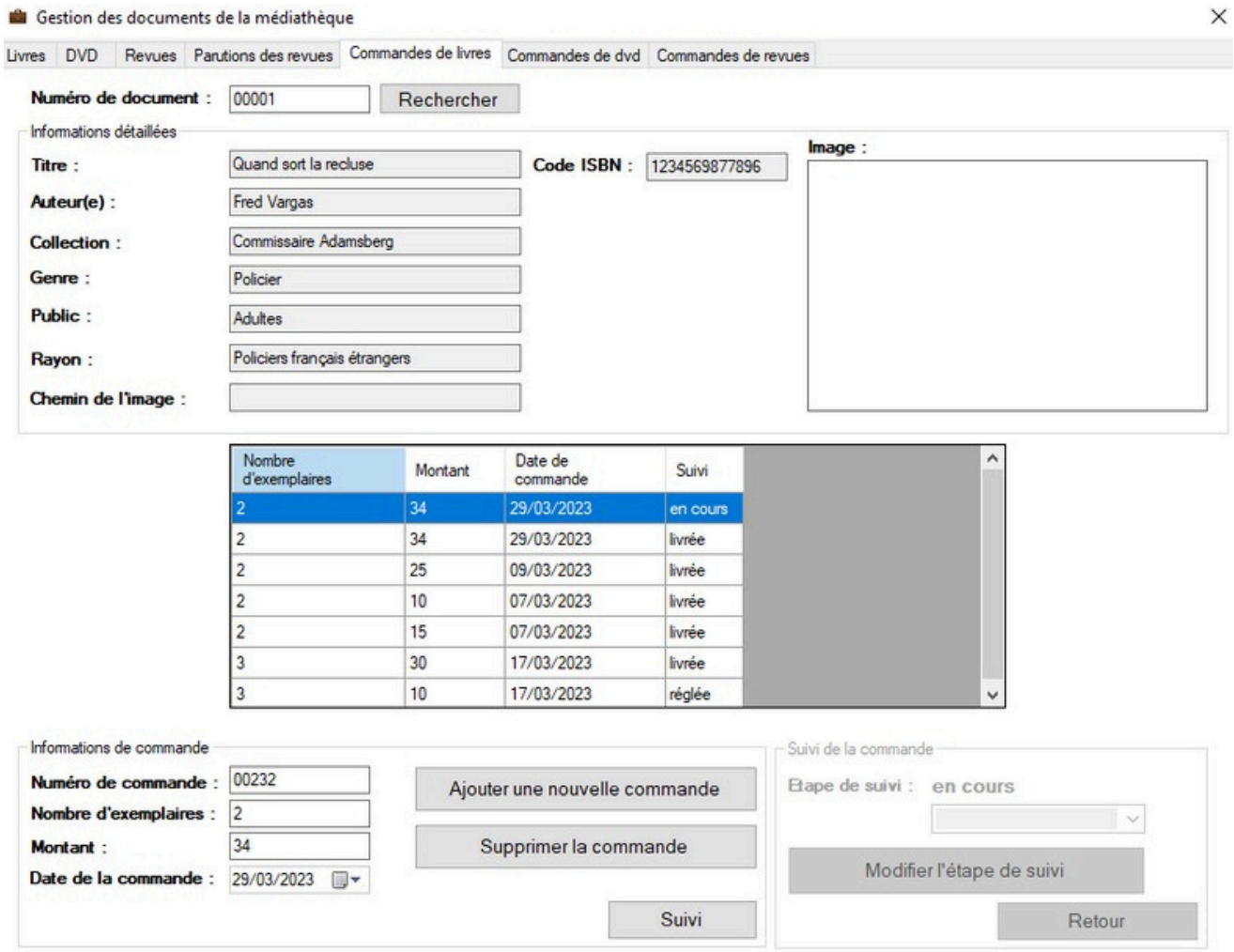
TACHE 1: GÉRER LES COMMANDES DE LIVRES OU DE DVD

1-COMMANDES DE LIVRES

J'ai commencé par créer le nouvel onglet "Commandes de livres", en m'inspirant du design de l'onglet "Parutions des revues". L'interface permet de saisir le numéro d'un livre pour rechercher les commandes associées. Les commandes sont affichées dans une datagrid, tandis que les informations du livre apparaissent dans une groupbox non-modifiable.

Les détails de la commande sélectionnée sont affichés dans une groupbox "Informations de commande", qui est initialement inaccessible. Pour ajouter une nouvelle commande, j'ai intégré des champs pour le numéro de commande, le nombre d'exemplaires, le montant et la date de commande, accompagnés d'un bouton "Ajouter une nouvelle commande". De même, un bouton "Supprimer la commande" permet de supprimer une commande existante.

Pour permettre la modification de l'étape de suivi d'une commande, j'ai ajouté un libellé "Étape" affichant l'étape de suivi actuelle, suivi d'une combobox contenant les étapes de suivi disponibles. Un bouton "Modifier l'étape de suivi" a été placé à cet effet. Enfin, les boutons "Suivi" et "Retour" permettent respectivement de gérer le suivi de la commande et de revenir aux informations de la commande.



Dans la procédure événementielle "tabCommandesLivres_Enter", je mets en place l'affichage initial en déclarant les variables nécessaires pour stocker les listes. Ces listes seront utilisées ultérieurement pour remplir la datagrid et la comboBox des suivis.

Suite à la création de la table "suivi" dans la base de données, j'intègre la fonction "selectAllCommandesDocument($id)" dans l'API rest_mediatekdocuments existante. Cette fonction récupère les commandes de documents en fonction de leur ID. Elle récupère les données des champs relatifs au livre, au suivi, à la table commande, et à la table commandedocument, en les triant par ordre descendant.

```
public function selectAllCommandesDocument($id) {
    $param = array (
        "id" => $id
    );
    $req = "SELECT l.nbExemplaire, l.idLivreDvd, l.idSuivi, s.libelle, l.id,
MAX(c.dateCommande) as dateCommande, SUM(c.montant) as montant ";
    $req .= "FROM commandedocument l ";
    $req .= "JOIN suivi s ON s.id = l.idSuivi ";
    $req .= "LEFT JOIN commande c ON l.id = c.id ";
    $req .= "WHERE l.idLivreDvd = :id ";
    $req .= "GROUP BY l.id ";
    $req .= "ORDER BY dateCommande DESC";
    return $this->conn->query($req, $param);
}
```

Dans la mise en place de la fonctionnalité des commandes de livres, j'ai commencé par créer un nouvel onglet dans l'interface utilisateur, intitulé "Commandes de livres", en m'inspirant du design de l'onglet existant pour les parutions des revues. L'objectif était de permettre à l'utilisateur de saisir un numéro de livre et de rechercher les commandes associées à ce livre. Les commandes sont affichées dans une grille de données (datagrid), tandis que les détails du livre sont présentés dans une groupbox non modifiable.

Pour récupérer les commandes de document associées à un livre depuis l'API REST PHP, j'ai ajouté une fonction "selectAllCommandesDocument($id)" qui récupère ces données, triées par date de commande décroissante. Cette fonction est ensuite utilisée dans la requête "selectOne($table, $id)" pour le cas spécifique "commandedocument", ce qui permet à la classe "Access" de l'application C# d'exploiter ces données.

Dans la classe "Access", j'ai également étendu la fonction "selectAll($table)" pour inclure le cas "commandedocument", afin de récupérer toutes les commandes de document disponibles.

Pour l'affichage des commandes dans la datagrid de l'onglet "Commandes de livres", j'ai mis en place plusieurs méthodes. Tout d'abord, j'ai créé les entités "Commande" et "CommandeDocument" dans le dossier "Model", en leur attribuant les propriétés appropriées. Ensuite, dans la vue "FrmMediatek", j'ai développé la méthode "RemplirCommandesLivresListe(List<CommandeDocument> lesCommandesDocument)" qui se charge de remplir la grille avec les données des commandes de document fournies en paramètre. Cette méthode assure également un ajustement automatique des colonnes de la grille pour une meilleure lisibilité.

Enfin, j'ai implémenté la méthode "AfficheReceptionCommandesLivre()" pour mettre à jour l'affichage des commandes lorsque l'utilisateur saisit un numéro de livre dans le champ de recherche. Cette méthode récupère les commandes correspondant au livre saisi et met à jour la datagrid en appelant la méthode "RemplirCommandesLivresListe(lesCommandesDocument)" avec la liste des commandes récupérées.

MÉTHODE - REMPLIRCOMMANDESLIVRESLISTE

```
private void RemplirCommandesLivresListe(List<CommandeDocument>
lesCommandesDocument)
{
   if (lesCommandesDocument != null)
   {
      bdgCommandesLivre.DataSource = lesCommandesDocument;
      dgvCommandesLivre.DataSource = bdgCommandesLivre;
      dgvCommandesLivre.Columns["id"].Visible = false;
      dgvCommandesLivre.Columns["idLivreDvd"].Visible = false;
      dgvCommandesLivre.Columns["idSuivi"].Visible = false;
      dgvCommandesLivre.AutoSizeColumnsMode =
DataGridViewAutoSizeColumnsMode.AllCells;
      dgvCommandesLivre.Columns["dateCommande"].DisplayIndex = 4;
      dgvCommandesLivre.Columns["montant"].DisplayIndex = 1;
      dgvCommandesLivre.Columns[5].HeaderCell.Value = "Date de commande";
      dgvCommandesLivre.Columns[0].HeaderCell.Value = "Nombre d'exemplaires";
      dgvCommandesLivre.Columns[3].HeaderCell.Value = "Suivi";
   }
   else
   {
      bdgCommandesLivre.DataSource = null;
   }
}
```

MÉTHODE - AFFICHERECEPTIONCOMMANDESLIVRE

```
private void AfficheReceptionCommandesLivre()
{
   string idDocument = txbCommandesLivresNumRecherche.Text;
   lesCommandesDocument = controller.GetCommandesDocument(idDocument);
   RemplirCommandesLivresListe(lesCommandesDocument);
}
```

Lorsque l'utilisateur clique sur le bouton "rechercher", la procédure événementielle "btnCommandesLivresNumRecherche_Click" est déclenchée pour afficher les commandes liées au livre recherché dans le champ "txbCommandesLivresNumRecherche.Text".

Cette méthode commence par vérifier si un numéro de document a été saisi dans le champ de recherche en utilisant la condition "if (!txbCommandesLivresNumRecherche.Text.Equals(""))". La saisie d'un numéro de document est rendue obligatoire pour effectuer la recherche.

Ensuite, la méthode "Find()" est utilisée pour retrouver le livre correspondant à l'ID saisi dans le champ de recherche dans la liste "lesLivres". Si le livre existe, la méthode "AfficheReceptionCommandesLivre()" est appelée pour afficher la liste des commandes associées à ce livre. Enfin, la zone d'affichage des informations de commande concernant ce livre est activée, permettant à l'utilisateur d'effectuer des ajouts, des modifications ou des suppressions de commande.

MÉTHODE - BTNCOMMANDESLIVRESNUMRECHERCHE_CLICK

```
private void btnCommandesLivresNumRecherche_Click(object sender, EventArgs e)
{
   if (!txbCommandesLivresNumRecherche.Text.Equals(""))
   {
      Livre livre = lesLivres.Find(x =>
x.Id.Equals(txbCommandesLivresNumRecherche.Text));
      if (livre != null)
      {
         AfficheReceptionCommandesLivre();
         gbxInfosCommandeLivre.Enabled = true;
      }
   }
}
```

Une fois les commandes du livre affichées avec succès, j'ai ajouté une fonction pour afficher les détails du livre recherché, remplissant les champs appropriés avec les informations telles que le titre et les auteurs. Cette fonction est déclenchée lors du clic sur le bouton de recherche, garantissant ainsi une expérience utilisateur plus fluide et complète.

MÉTHODE - AFFICHERECEPTIONCOMMANDESLIVREINFOS

```
cbxCommandeLivresLibelleSuivi.Items.Clear();
if (etapeSuivi == "livrée")
{
   cbxCommandeLivresLibelleSuivi.Text = "";
   cbxCommandeLivresLibelleSuivi.Items.Add("réglée");
}
else if (etapeSuivi == "en cours")
{
   cbxCommandeLivresLibelleSuivi.Text = "";
   cbxCommandeLivresLibelleSuivi.Items.Add("relancée");
   cbxCommandeLivresLibelleSuivi.Items.Add("livrée");
}
else if (etapeSuivi == "relancée")
{
   cbxCommandeLivresLibelleSuivi.Text = "";
   cbxCommandeLivresLibelleSuivi.Items.Add("en cours");
   cbxCommandeLivresLibelleSuivi.Items.Add("livrée");
}
```

Pour récupérer l'id de l'étape de suivi d'une commande, j'ajoute une fonction "GetIdSuivi(string libelle)" qui récupère la liste des objets "Suivi" et trouve l'id correspondant au libellé donné.

MÉTHODE - GETIDSUIVI

```
private string GetIdSuivi(string libelle)
{
   List<Suivi> lesSuivisCommande = controller.GetAllSuivis();
   foreach (Suivi unSuivi in lesSuivisCommande)
   {
      if (unSuivi.Libelle == libelle)
      {
         return unSuivi.Id;
      }
   }
   return null;
}
```

Je crée la procédure "dgvCommandesLivre_RowEnter" pour afficher les informations de la commande sélectionnée dans la datagrid dans la groupBox dédiée. En utilisant des variables pour stocker les données de la ligne sélectionnée, je les affecte ensuite aux champs correspondants. Pour garantir une meilleure expérience utilisateur, j'ajoute une condition pour empêcher la modification de l'étape de suivi si la commande est déjà réglée, en appelant la méthode "GetIdSuivi" pour obtenir l'ID correspondant au libellé "réglée".

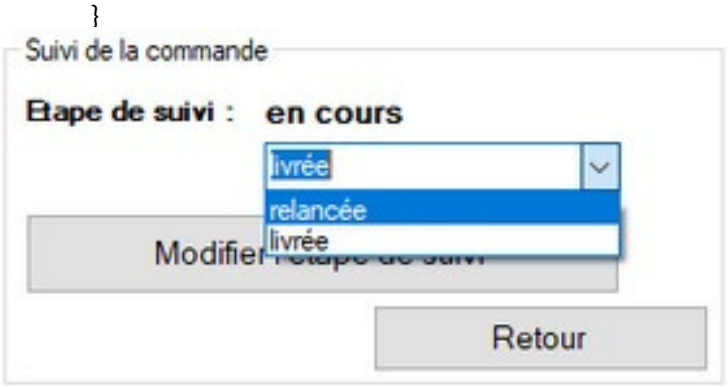
PROCÉDURE ÉVÈNEMENTIELLE - DGVCOMMANDESLIVRE_ROWENTER

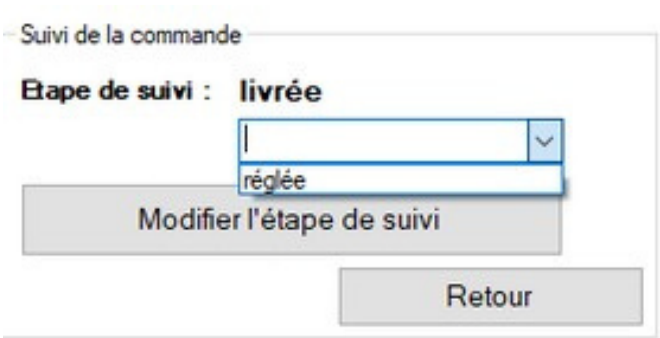
```
private void dgvCommandesLivres_RowEnter(object sender,
DataGridViewCellEventArgs e)
{
   DataGridViewRow row = dgvCommandesLivres.Rows[e.RowIndex];
   string id = row.Cells["Id"].Value.ToString();
   DateTime dateCommande = (DateTime)row.Cells["dateCommande"].Value;
   double montant = double.Parse(row.Cells["Montant"].Value.ToString());
   int nbExemplaire = int.Parse(row.Cells["NbExemplaire"].Value.ToString());
   string libelle = row.Cells["Libelle"].Value.ToString();

   txbCommandeLivreNumero.Text = id;
   txbCommandeLivreNbExemplaires.Text = nbExemplaire.ToString();
   txbCommandeLivreMontant.Text = montant.ToString();
   dtpCommandeLivre.Value = dateCommande;
   lblEtapeSuivi.Text = libelle;

   if (GetIdSuivi(libelle) == "00003")
   {
      cbxCommandeLivresLibelleSuivi.Enabled = false;
      btnReceptionCommandeLivresModifierSuivi.Enabled = false;
   }
   else
   {
      cbxCommandeLivresLibelleSuivi.Enabled = true;
      btnReceptionCommandeLivresModifierSuivi.Enabled = true;
   }
}
```

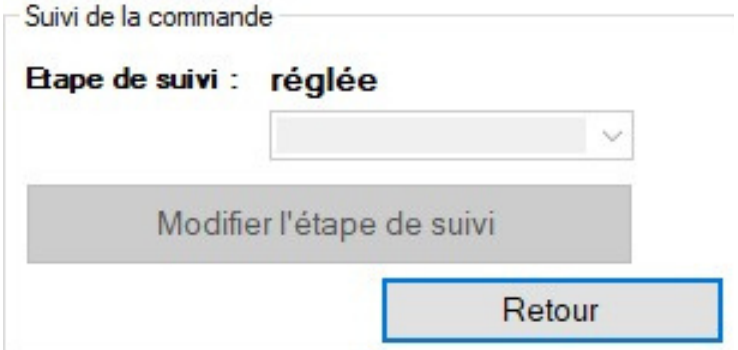
Je construis la procédure événementielle "dgvCommandesLivre_ColumnHeaderMouseClick_1" pour permettre le tri des colonnes dans l'ordre inverse de la chronologie lors du clic sur le header des colonnes.

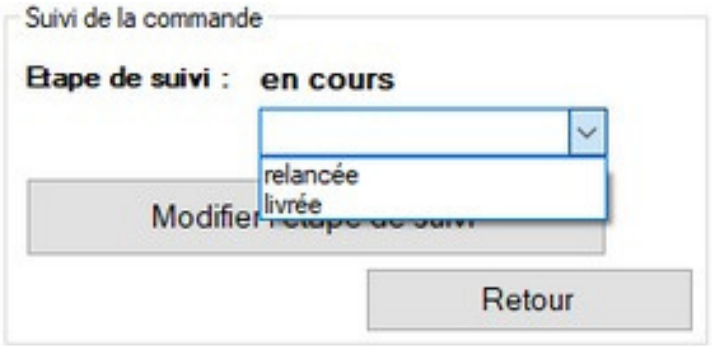
PROCÉDURE ÉVÈNEMENTIELLE - DGVCOMMANDESLIVRE_COLUMNHEADERMOUSECLICK_1

```
private void dgvCommandesLivre_ColumnHeaderMouseClick_1(object sender,
DataGridViewCellMouseEventArgs e)
{
   string titreColonne = dgvCommandesLivre.Columns[e.ColumnIndex].HeaderText;
   List<CommandeDocument> sortedList = new List<CommandeDocument>();
   switch (titreColonne)
   {
      case "Date de commande":
         sortedList = lesCommandesDocument.OrderBy(o =>
o.DateCommande).Reverse().ToList();
         break;
      case "Montant":
         sortedList = lesCommandesDocument.OrderBy(o => o.Montant).ToList();
         break;
      case "Nombre d'exemplaires":
         sortedList = lesCommandesDocument.OrderBy(o => o.NbExemplaire).ToList();
         break;
      case "Suivi":
         sortedList = lesCommandesDocument.OrderBy(o => o.Libelle).ToList();
         break;
   }
   RemplirCommandesLivresListe(sortedList);
}
```









Pour enrichir l'onglet des commandes de livres, je commence par développer la fonctionnalité "Ajouter", qui implique la création d'une nouvelle commande dans la base de données. Cela nécessite la mise en place de la méthode "CreerCommande(Commande commande)" dans la classe "Access", utilisant une requête HTTP POST pour envoyer les données à l'API Rest PHP via le endpoint "commande/", après avoir converti l'objet "Commande" en format JSON avec l'aide de "Newtonsoft.Json", incluant une conversion de format de date spécifique.

MÉTHODE - CREERCOMMANDE(COMMANDE COMMANDE)

```
public bool CreerCommande(Commande commande)
{
   string jsonCreerCommande = JsonConvert.SerializeObject(commande, new
CustomDateTimeConverter());
   Console.WriteLine("jsonCreerCommande " + jsonCreerCommande);
   try
   {
      // récupération soit d'une liste vide (requête ok) soit de null (erreur)
      List<Commande> liste = TraitementRecup<Commande>(POST, "commande/" +
jsonCreerCommande);
      return (liste != null);
   }
   catch (Exception ex)
   {
      Console.WriteLine(ex.Message);
   }
   return false;
}
```

Pour compléter l'ajout d'une commande, je dois créer une autre méthode nommée "CreerCommandeDocument(string id, int nbExemplaire, string idLivreDvd, string idSuivi)". Cette méthode va écrire les détails de la commande de document en envoyant les données correspondantes à la table "commandedocument" via une requête HTTP POST sur le endpoint "commandedocument/".

MÉTHODE - CREERCOMMANDEDOCUMENT

```
public bool CreerCommandeDocument(string id, int nbExemplaire, string idLivreDvd,
string idSuivi)
{
   string jsonCreerCommandeDocument = "{ \"id\" : \"" + id + "\", \"nbExemplaire\" : \"" +
      nbExemplaire + "\", \"idLivreDvd\" : \"" + idLivreDvd + "\", \"idSuivi\" : \"" + idSuivi +
"\"}";
   try
   {
      List<CommandeDocument> liste = TraitementRecup<CommandeDocument>(POST,
         "commandedocument/" + jsonCreerCommandeDocument);
      return (liste != null);
   }
```

Pour ajouter une nouvelle commande de livre, j'ai mis en place la fonctionnalité dans le "FrmMediatekController" et créé la procédure évènementielle "btnReceptionCommandeLivreValider_Click". Cette procédure vérifie que les champs obligatoires sont remplis avant d'insérer une nouvelle commande de livre dans la base de données. Ensuite, elle appelle les méthodes "CreerCommande" et "CreerCommandeDocument" pour enregistrer les données et met à jour l'affichage de la grille des commandes de livres. Une vérification supplémentaire a été ajoutée pour éviter les duplications d'identifiants de commande.

PROCÉDURE ÉVÈNEMENTIELLE - BTNRECEPTIONCOMMANDELIVREVALIDER_CLICK

```
string id = txbCommandeLivreNumero.Text;
int nbExemplaire = int.Parse(txbCommandeLivreNbExemplaires.Text);
double montant = double.Parse(txbCommandeLivreMontant.Text);
DateTime dateCommande = dtpCommandeLivre.Value;
string idLivreDvd = txbCommandesLivresNumRecherche.Text;
string idSuivi = lesSuivis[0].Id;
Commande commande = new Commande(id, dateCommande, montant);
var idCommandeLivreExistante = controller.GetCommandes(id);
var idCommandeLivreNonExistante = !idCommandeLivreExistante.Any();
if (idCommandeLivreNonExistante)
{
   if (controller.CreerCommande(commande))
   {
      controller.CreerCommandeDocument(id, nbExemplaire, idLivreDvd, idSuivi);
      MessageBox.Show("La commande " + id + " a bien été enregistrée.", "Information");
      AfficheReceptionCommandesLivre();
   }
}
else
{
   MessageBox.Show("Le numéro de la commande existe déjà, veuillez saisir un
nouveau numéro.", "Erreur");
}
```

Pour faciliter la fonctionnalité de "Modifier", j'ai déjà mis en place des méthodes pour gérer les étapes de suivi. Les règles sont directement appliquées dans la liste déroulante des étapes de suivi en fonction du suivi actuel de la commande sélectionnée. Maintenant, pour permettre la modification de l'étape lors du clic sur le bouton "Modifier", je retourne dans la classe "Access" et je crée la fonction "ModifierSuiviCommandeDocument(string id, string idSuivi)". Cette fonction prend en paramètres "id" et "idSuivi" et construit la chaîne JSON correspondante. Ensuite, une requête HTTP PUT est exécutée pour envoyer les données modifiées de l'objet "CommandeDocument" à l'endpoint "commandedocument/", en ajoutant l'ID du document à modifier, suivi de la chaîne JSON.

MÉTHODE - MODIFIERSUIVICOMMANDEDOCUMENT

```
public bool ModifierSuiviCommandeDocument(string id, int nbExemplaire, string
idLivreDvd, string idSuivi)
{
   String jsonModifierSuiviCommandeDocument = "{ \"id\" : \"" + id + "\",
\"nbExemplaire\" : \"" + nbExemplaire + "\", \"idLivreDvd\" : \"" + idLivreDvd + "\",
\"idSuivi\" : \"" + idSuivi + "\"}";
   try
   {
      List<CommandeDocument> liste = TraitementRecup<CommandeDocument>(PUT,
"commandedocument/" + id + "/" + jsonModifierSuiviCommandeDocument);
      return (liste != null);
   }
```

Dans la fonctionnalité "Modifier", j'ai mis en place une procédure événementielle sur le clic du bouton "Modifier", "btnModifierSuiviCommandeLivre_Click", qui permet de modifier l'étape de suivi de la commande. Les règles de modification sont appliquées en fonction de l'étape actuelle de la commande. Les données sont récupérées depuis les champs de saisie et la sélection de la comboBox des étapes de suivi. Un nouvel objet "CommandeDocument" est créé avec ces données, et une vérification est effectuée pour s'assurer que la modification est autorisée. Ensuite, la méthode "ModifierSuiviCommandeDocument" du controller est appelée pour effectuer la modification, et l'affichage de la grille des données est mis à jour.

PROCÉDURE ÉVÈNEMENTIELLE - BTNMODIFIERSUIVICOMMANDELIVRE_CLICK

```
string id = txbCommandeLivreNumero.Text;
int nbExemplaire = int.Parse(txbCommandeLivreNbExemplaires.Text);
double montant = double.Parse(txbCommandeLivreMontant.Text);
DateTime dateCommande = dtpCommandeLivre.Value;
string idLivreDvd = txbCommandesLivresNumRecherche.Text;
string idSuivi = GetIdSuivi(cbxCommandeLivresLibelleSuivi.Text);
try
{
   string libelle = cbxCommandeLivresLibelleSuivi.SelectedItem.ToString();
   CommandeDocument commandeDocument = new CommandeDocument(id,
dateCommande, montant, nbExemplaire, idLivreDvd, idSuivi, libelle);
   if (MessageBox.Show("Voulez-vous modifier le suivi de la commande " +
commandeDocument.Id + " en " + libelle + " ?", "Confirmation",
MessageBoxButtons.YesNo) == DialogResult.Yes)
   {
      controller.ModifierSuiviCommandeDocument(commandeDocument.Id,
commandeDocument.IdSuivi);
      MessageBox.Show("L'étape de suivi de la commande " + id + " a bien été modifiée.",
"Information");
      AfficheReceptionCommandesLivre();
   }
}
```

Je réalise la fonction "SupprimerCommandeDocument(CommandeDocument commandesDocument)" dans la classe "Access", qui supprime l'objet "CommandeDocument" via une requête http DELETE sur "commandedocument/" avec la chaîne json. J'appelle cette méthode dans le controller et je crée la procédure évènementielle "btnCommandeLivreSupprimer_Click", qui supprime la commande de livre sélectionnée dans la datagrid si elle n'est pas encore livrée. Une condition "if" vérifie si une ligne est sélectionnée avant de supprimer, et une autre condition permet la suppression seulement si le libellé est "en cours" ou "relancée".

PROCÉDURE ÉVÈNEMENTIELLE - BTNCOMMANDELIVRESUPPRIMER_CLICK

```
private void btnCommandeLivreSupprimer_Click(object sender, EventArgs e)
{
  if (dgvCommandesLivre.SelectedRows.Count > 0)
  {
    CommandeDocument commandeDocument =
(CommandeDocument)bdgCommandesLivre.List[bdgCommandesLivre.Position];
    if (commandeDocument.Libelle == "en cours" || commandeDocument.Libelle ==
"relancée")
    {
      if (MessageBox.Show("Voulez-vous vraiment supprimer la commande " +
commandeDocument.Id + " ?", "Confirmation de suppression",
MessageBoxButtons.YesNo) == DialogResult.Yes)
      {
        controller.SupprimerCommandeDocument(commandeDocument);
        AfficheReceptionCommandesLivre();
      }
    }
    else
    {
      MessageBox.Show("La commande sélectionnée a été livrée, elle ne peut pas être
supprimée.", "Information");
    }
  }
  else
  {
    MessageBox.Show("Une ligne doit être sélectionnée.", "Information");
  }
}
```

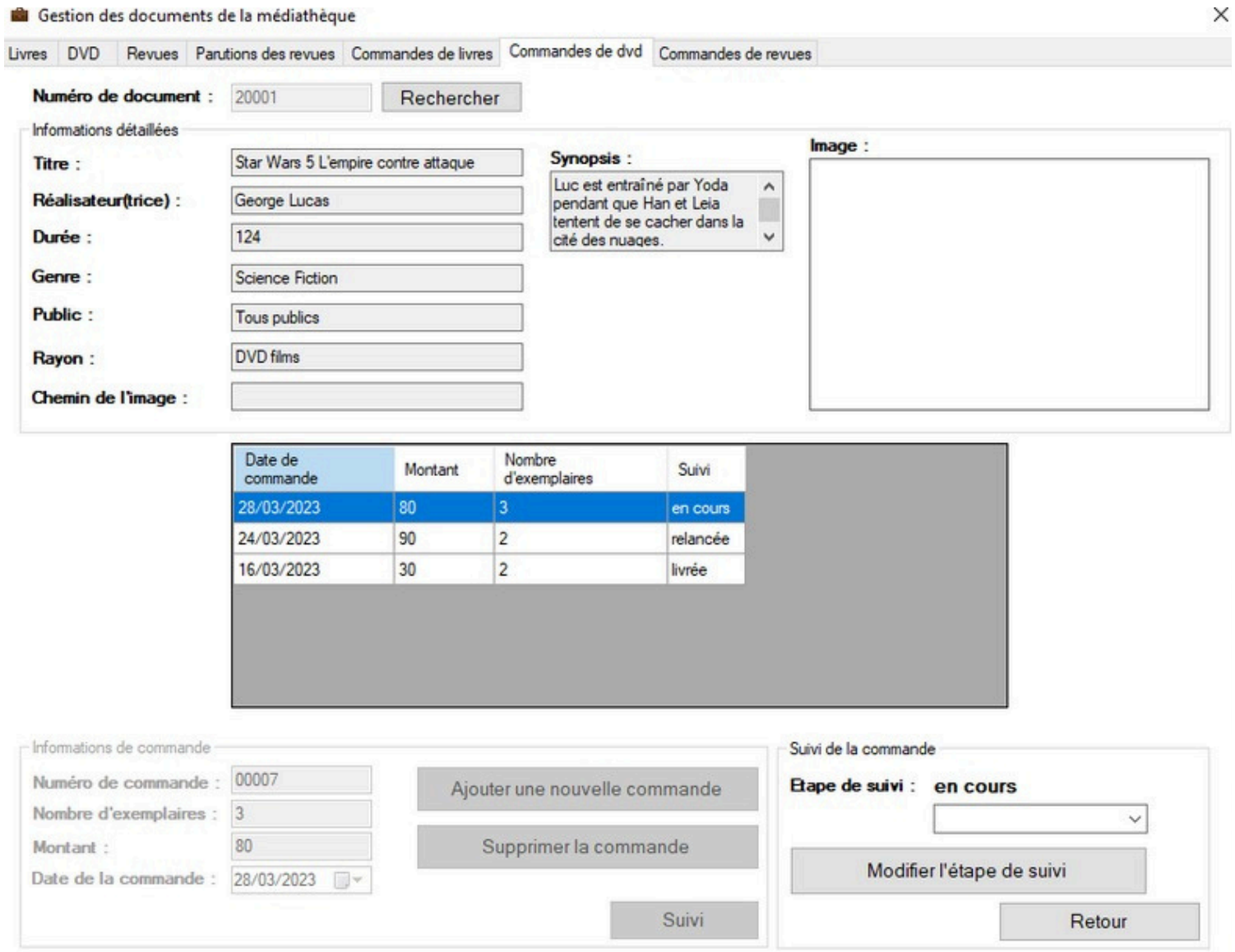
Pour créer un trigger permettant l'ajout des exemplaires d'un document dans la table "exemplaire" lorsque la commande passe à l'étape "livrée", j'ai déclaré et initialisé plusieurs variables telles que "compteur", "dateAchatExemplaire", "nbExemplaire", "maxNumeroExemplaire", et "newNumeroExemplaire". Ensuite, j'ai utilisé des instructions "SET" pour leur affecter les valeurs appropriées. Le déclencheur comporte une condition "IF" vérifiant si l'étape de suivi de la commande est "livrée". Dans ce cas, il exécute une boucle "while" pour insérer les nouveaux exemplaires dans la table "exemplaire" en utilisant les informations de la commande, tels que l'id du document, le numéro d'exemplaire, la date d'achat/de commande, et l'état de l'exemplaire, que je mets à "neuf" avec l'identifiant correspondant.

TRIGGER - CREATION_EXEMPLAIRE

```
CREATE TRIGGER `Creation_Exemplaire` AFTER UPDATE ON `commandedocument`
FOR EACH ROW BEGIN
DECLARE compteur INTEGER;
DECLARE dateAchatExemplaire DATE;
DECLARE nbExemplaire INTEGER;
DECLARE maxNumeroExemplaire INTEGER;
DECLARE newNumeroExemplaire INTEGER;
SET compteur = 0;
SET dateAchatExemplaire = (SELECT dateCommande FROM commande JOIN
commandedocument ON commande.id = commandedocument.id WHERE
commandedocument.id = NEW.id);
SET nbExemplaire = NEW.nbExemplaire;
SET maxNumeroExemplaire = (SELECT MAX(numero) FROM exemplaire WHERE id =
NEW.idlivreDvd);
IF (NEW.idSuivi != OLD.idSuivi AND NEW.idSuivi = "00002") THEN
  IF (maxNumeroExemplaire IS NOT NULL) THEN
    SET newNumeroExemplaire = maxNumeroExemplaire;
  ELSE
    SET newNumeroExemplaire = 0;
  END IF;
  WHILE compteur < nbExemplaire DO
    SET newNumeroExemplaire = newNumeroExemplaire + 1;
    INSERT INTO exemplaire (id, numero, dateAchat, photo, idEtat) VALUES
(NEW.idLivreDvd, newNumeroExemplaire, dateAchatExemplaire, "", "00001");
    SET compteur = compteur + 1;
  END WHILE;
END IF;
END;
```

2-COMMANDES DE DVD

- Création de l'onglet "Commandes de DVD" avec un design similaire à celui des "Commandes de livres".
- Possibilité de saisir un numéro de DVD pour rechercher les commandes le concernant, affichées dans une datagrid.
- Affichage des informations détaillées du DVD dans une groupbox non modifiable.
- Les détails de la commande sélectionnée sont récupérés dans une autre groupbox "Informations de commande", uniquement accessible après saisie d'un numéro de document valide.
- Ajout d'une nouvelle commande avec numéro, nombre d'exemplaires, montant et date de commande.
- Fonctionnalité de suppression de commande avec un bouton dédié.
- Modification de l'étape de suivi de la commande selon des règles spécifiques, gérée par une comboBox et un bouton "Modifier l'étape de suivi".



Dans la mise en place de l'onglet "Commandes de DVD", l'objectif est de permettre à l'utilisateur de saisir le numéro d'un DVD pour rechercher les commandes associées. Lorsque cet onglet est activé, les listes nécessaires pour remplir la datagrid et la comboBox des suivis sont initialisées dans la procédure évènementielle "tabCommandesDvd_Enter". Cette initialisation est cruciale pour assurer un affichage correct des données.

Pour afficher les commandes relatives à un DVD recherché, trois nouvelles méthodes sont créées. La première méthode, nommée "RemplirCommandesDvdListe", est chargée d'afficher les commandes dans la datagrid. Elle prend en paramètre une liste d'objets "CommandeDocument" et les affiche dans la datagrid "dgvCommandesDvd". Les colonnes inutiles sont masquées pour éviter toute confusion, et l'ajustement automatique des colonnes est activé pour garantir une meilleure visibilité des données.

Cette structuration permet à l'utilisateur de visualiser clairement les commandes associées à un DVD donné, facilitant ainsi la gestion des commandes de DVD dans l'application.

MÉTHODE – REMPLIRCOMMANDESDVDLISTE

```
private void RemplirCommandesDvdListe(List<CommandeDocument>
lesCommandesDocument)
{
   if (lesCommandesDocument != null)
   {
      bdgCommandesDvd.DataSource = lesCommandesDocument;
      dgvCommandesDvd.DataSource = bdgCommandesDvd;
      dgvCommandesDvd.Columns["id"].Visible = false;
      dgvCommandesDvd.Columns["idLivreDvd"].Visible = false;
      dgvCommandesDvd.Columns["idSuivi"].Visible = false;
      dgvCommandesDvd.AutoSizeColumnsMode =
DataGridViewAutoSizeColumnsMode.AllCells;
      dgvCommandesDvd.Columns["dateCommande"].DisplayIndex = 0;
      dgvCommandesDvd.Columns["montant"].DisplayIndex = 1;
      dgvCommandesDvd.Columns[5].HeaderCell.Value = "Date de commande";
      dgvCommandesDvd.Columns[0].HeaderCell.Value = "Nombre d'exemplaires";
      dgvCommandesDvd.Columns[3].HeaderCell.Value = "Suivi";
   }
   else
   {
      bdgCommandesDvd.DataSource = null;
   }
}
```

Pour mettre à jour l'affichage des commandes de DVD saisies dans le champ de recherche "txbCommandesDvdNumRecherche", j'ai créé la méthode "AfficheReceptionCommandesDvd()". Elle récupère l'ID du document saisi dans le champ et utilise cette valeur pour obtenir la liste des commandes associées grâce à un getter. Ensuite, la méthode "RemplirCommandesDvdListe(lesCommandesDocument)" est appelée pour mettre à jour l'affichage de la grille avec les nouvelles commandes associées au document recherché.

MÉTHODE - AFFICHERECEPTIONCOMMANDESDVD

```
private void AfficheReceptionCommandesDvd()
{
   string idDocument = txbCommandesDvdNumRecherche.Text;
   lesCommandesDocument = controller.GetCommandesDocument(idDocument);
   RemplirCommandesDvdListe(lesCommandesDocument);
}
```

Pour afficher les commandes relatives au DVD recherché dans la zone de données lors du clic sur le bouton "rechercher", j'ai implémenté la méthode "btnCommandesDvdNumRecherche_Click". Cette méthode comporte une condition vérifiant la saisie d'un numéro de document dans le champ de recherche, puis elle utilise la méthode "Find()" pour retrouver le DVD correspondant dans la liste "lesDvd". Si le DVD est trouvé, la fonction "AfficheReceptionCommandesDvd()" est appelée pour afficher la liste des commandes associées à ce DVD, et la zone d'affichage des informations de commande est activée pour permettre à l'utilisateur d'accéder aux fonctionnalités d'ajout, de modification ou de suppression de commande.

PROCÉDURE ÉVÈNEMENTIELLE - BTNCOMMANDESDVDNUMRECHERCHE_CLICK

```
private void btnCommandesDvdNumRecherche_Click(object sender, EventArgs e)
{
   if (!txbCommandesDvdNumRecherche.Text.Equals(""))
   {
      Dvd dvd = lesDvd.Find(x => x.Id.Equals(txbCommandesDvdNumRecherche.Text));
      if (dvd != null)
      {
         AfficheReceptionCommandesDvd();
         gbxInfosCommandeDvd.Enabled = true;
      }
      else
      {
         MessageBox.Show("Numéro introuvable", "Erreur");
      }
   }
   else
   {
      MessageBox.Show("Le numéro de document est obligatoire", "Information");
   }
}
```

Après avoir mis en place l'affichage des commandes de DVD et les informations détaillées concernant le DVD recherché, j'ai développé les fonctionnalités d'ajout, de modification et de suppression des commandes.

Pour l'ajout d'une nouvelle commande de DVD, j'ai implémenté la procédure évènementielle "btnReceptionCommandeDvdValider_Click". Cette procédure vérifie les données saisies telles que le numéro de commande, le montant, le nombre d'exemplaires et la date. Ensuite, elle insère la nouvelle commande dans la base de données en appelant les méthodes appropriées dans la classe "Access". L'affichage de la grille est ensuite mis à jour pour inclure le nouvel enregistrement.

La fonctionnalité de modification de l'étape de suivi d'une commande a été réalisée en implémentant la méthode "ModifierSuiviCommandeDocument" dans le contrôleur. Les règles de modification sont gérées directement dans la comboBox des étapes de suivi, en fonction de l'étape actuelle de la commande. L'utilisateur peut sélectionner une nouvelle étape de suivi, puis la modification est enregistrée dans la base de données.

En ce qui concerne la suppression des commandes de DVD, j'ai créé la procédure évènementielle "btnCommandeDvdSupprimer_Click". Cette procédure vérifie d'abord si une ligne de la grille est sélectionnée, puis elle vérifie si la commande peut être supprimée en fonction de son état. Si la commande n'est pas encore livrée, elle est supprimée de la base de données en appelant la méthode appropriée dans la classe "Access".

PROCÉDURE ÉVÈNEMENTIELLE - BTNCOMMANDEDVDSUPPRIMER_CLICK

```
private void btnCommandeDvdSupprimer_Click(object sender, EventArgs e)
{
  if (dgvCommandesDvd.SelectedRows.Count > 0)
  {
    CommandeDocument commandeDocument =
(CommandeDocument)bdgCommandesDvd.List[bdgCommandesDvd.Position];
    if (commandeDocument.Libelle == "en cours" || commandeDocument.Libelle ==
"relancée")
    {
      if (MessageBox.Show("Voulez-vous vraiment supprimer la commande " +
commandeDocument.Id + " ?", "Confirmation de suppression",
MessageBoxButtons.YesNo) == DialogResult.Yes)
      {
        controller.SupprimerCommandeDocument(commandeDocument);
        AfficheReceptionCommandesDvd();
      }
    }
    else
    {
      MessageBox.Show("La commande sélectionnée a été livrée elle ne peut pas être
supprimée.", "Information");
    }
  }
  else
  {
    MessageBox.Show("Une ligne doit être sélectionnée.", "Information");
  }
}
```

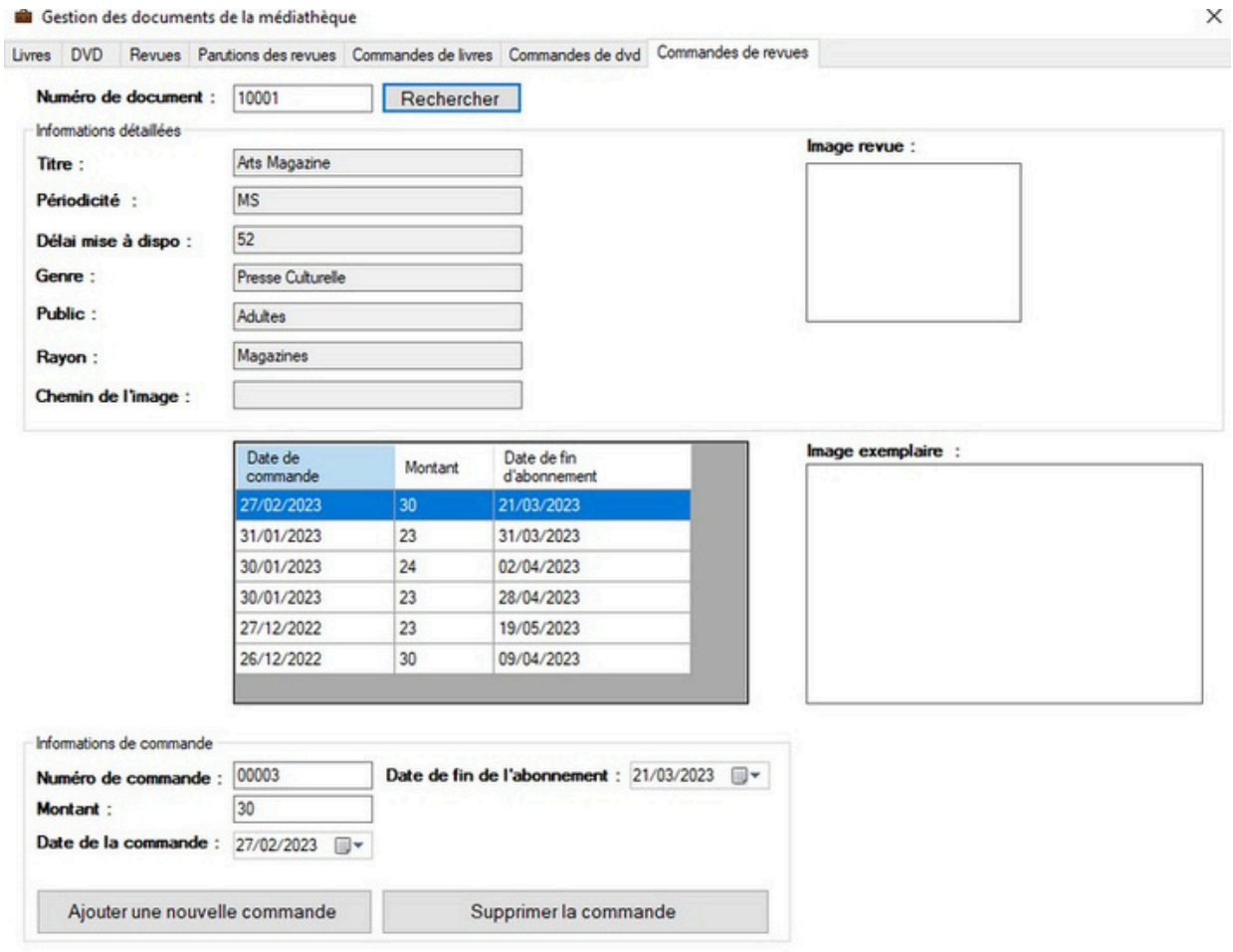
MISSION 2 : GERER LES COMMANDES

TACHE 2: GÉRER LES COMMANDES DE REVUES

Dans la création de l'onglet "Commandes de revues", je me suis inspiré du design de l'onglet "Parutions des revues". L'objectif est de permettre à l'utilisateur de saisir le numéro d'une revue pour rechercher les commandes associées. Ces commandes doivent être affichées dans une datagrid, tandis que les informations de la revue doivent figurer dans une groupbox non modifiable, puisqu'il s'agit seulement d'affichage.

Les informations de la commande sélectionnée dans la grille doivent être récupérées dans une groupbox "Informations de commande". Cette groupbox sera disponible uniquement après la saisie d'un numéro de document valide.

Pour ajouter une nouvelle commande, il est nécessaire de fournir un numéro de commande, un montant, ainsi que la date de commande et la date de fin de l'abonnement, car les commandes de revues fonctionnent par abonnement. Un bouton "Ajouter une nouvelle commande" est prévu à cet effet. De plus, afin de permettre la suppression d'une commande ou d'un abonnement, un bouton "Supprimer la commande" est également présent dans l'interface.



Dans la phase d'initialisation de l'affichage des listes, je déclare les variables nécessaires pour contenir les données qui seront utilisées pour remplir la datagrid. Cela se fait dans la procédure évènementielle "tabCommandesRevues_Enter".

Pour que l'application C# récupère les données de la base de données, elle doit exploiter les fonctions de l'API rest_mediatekdocuments déjà pré-conçue et fournie.

Une fonction essentielle à ajouter est celle qui permettra de récupérer les abonnements d'une revue en fonction de leur identifiant. Pour cela, j'insère la fonction "selectAllAbonnementsRevues($id)". Cette fonction prend en paramètre "$id" et sélectionne les données correspondantes dans les tables "abonnement" et "commande", triées par ordre descendant.

FONCTION - SELECTALLABONNEMENTSREVUES($ID)

```
public function selectAllAbonnementsRevues($id){
   $param = array(
      "id" => $id
   );
   $req = "SELECT c.id, c.dateCommande, c.montant, a.dateFinAbonnement, a.idRevue ";
   $req .= "FROM commande c JOIN abonnement a ON c.id = a.id ";
   $req .= "WHERE a.idRevue = :id ";
   $req .= "ORDER BY c.dateCommande DESC ";
   return $this->conn->queryAll($req, $param);
}
```

Pour afficher la liste des abonnements d'une revue recherchée dans une DataGridView, j'ai d'abord créé la classe "Abonnement" dans le dossier "Model", héritant de la classe "Commande" pour initialiser ses propriétés de base. Ensuite, j'ai réalisé la méthode "RemplirAbonnementsRevueListe" qui remplit la grille avec les abonnements récupérés. Cette méthode prend en paramètre une liste d'objets "Abonnement" et les insère dans la DataGridView après avoir masqué les colonnes inutiles. J'ai également ajusté automatiquement la taille des colonnes pour une meilleure visibilité.

MÉTHODE - REMPLIRABONNEMENTSREVUELISTE

```
private void RemplirAbonnementsRevueListe(List<Abonnement>
lesAbonnementsRevue)
{
   if (lesAbonnementsRevue != null)
   {
      bdgAbonnementsRevue.DataSource = lesAbonnementsRevue;
      dgvAbonnementsRevue.DataSource = bdgAbonnementsRevue;
      dgvAbonnementsRevue.Columns["id"].Visible = false;
      dgvAbonnementsRevue.Columns["idRevue"].Visible = false;
      dgvAbonnementsRevue.Columns["titre"].Visible = false;
      dgvAbonnementsRevue.AutoSizeColumnsMode =
DataGridViewAutoSizeColumnsMode.AllCells;
      dgvAbonnementsRevue.Columns["dateCommande"].DisplayIndex = 0;
      dgvAbonnementsRevue.Columns["montant"].DisplayIndex = 1;
      dgvAbonnementsRevue.Columns[4].HeaderCell.Value = "Date de commande";
      dgvAbonnementsRevue.Columns[0].HeaderCell.Value = "Date de fin
d'abonnement";
   }
}
```

La méthode "AfficheReceptionAbonnementsRevue" est appelée lorsqu'un utilisateur saisit le numéro de revue dans le champ de recherche "txbCommandesRevueNumRecherche". Cette méthode récupère la liste des abonnements de la revue correspondante et met à jour l'affichage dans la grille en appelant la méthode "RemplirAbonnementsRevueListe". Cela permet à l'utilisateur de voir les abonnements associés à la revue recherchée.

MÉTHODE - AFFICHERECEPTIONABONNEMENTSREVUE

```
private void AfficheReceptionAbonnementsRevue()
{
   string idDocument = txbCommandesRevueNumRecherche.Text;
   lesAbonnementsRevue = controller.GetAbonnementRevue(idDocument);
   RemplirAbonnementsRevueListe(lesAbonnementsRevue);
}
```

La troisième méthode consiste en une procédure événementielle déclenchée par le clic sur le bouton "rechercher". Son nom est "btnCommandesRevueNumRecherche_Click". Elle vérifie d'abord si le champ de recherche de numéro de revue n'est pas vide. Ensuite, elle utilise la méthode "Find()" pour rechercher la revue correspondant à l'ID saisi dans le champ de recherche, dans la liste des revues disponibles. Si la revue est trouvée, la méthode "AfficheReceptionAbonnementsRevue()" est appelée pour afficher la liste des abonnements associés à cette revue. Enfin, la zone d'affichage des informations de commande concernant cette revue est activée pour permettre à l'utilisateur d'accéder aux ajouts, modifications ou suppressions d'abonnements.

PROCÉDURE ÉVÈNEMENTIELLE - BTNCOMMANDESREVUENUMRECHERCHE_CLICK

```
private void btnCommandesRevueNumRecherche_Click(object sender, EventArgs e)
{
  if (!txbCommandesRevueNumRecherche.Text.Equals(""))
  {
    Revue revue = lesRevues.Find(x =>
x.Id.Equals(txbCommandesRevueNumRecherche.Text));
    if (revue != null)
    {
      AfficheReceptionAbonnementsRevue();
      gbxInfosCommandeRevue.Enabled = true;
    }
    else
    {
      MessageBox.Show("Ce numéro de revue n'existe pas.");
    }
  }
  else
  {
    MessageBox.Show("Le numéro de revue est obligatoire.");
  }
}
```

Une fois que l'affichage des abonnements de la revue concernée fonctionne correctement, je crée une méthode "AfficheReceptionAbonnementsRevueInfos(Revue revue)" qui affiche les informations détaillées de la revue recherchée. Cette méthode prend un objet "Revue" en paramètre, ce qui permet de remplir les champs de texte avec les informations de la revue grâce aux propriétés de cet objet. Ensuite, j'appelle cette fonction dans la procédure "btnCommandesRevueNumRecherche_Click" afin que lors du clic sur le bouton "Recherche", les informations détaillées de la revue s'affichent dans les champs respectifs.

MÉTHODE - AFFICHERECEPTIONABONNEMENTSREVUEINFOS

```
private void AfficheReceptionAbonnementsRevueInfos(Revue revue)
{
  txbCommandeRevueTitre.Text = revue.Titre;
  txbCommandeRevuePeriodicite.Text = revue.Periodicite;
  txbCommandeRevueDelai.Text = revue.DelaiMiseADispo.ToString();
  txbCommandeRevueGenre.Text = revue.Genre;
  txbCommandeRevuePublic.Text = revue.Public;
  txbCommandeRevueRayon.Text = revue.Rayon;
  txbCommandeRevueCheminImage.Text = revue.Image;
  string image = revue.Image;
  try
  {
    pictureBox4.Image = Image.FromFile(image);
  }
  catch
  {
    pictureBox4.Image = null;
  }
}
```

Dans la méthode événementielle "dgvAbonnementsRevue_RowEnter", je configure l'affichage des informations de l'abonnement sélectionné dans la DataGridView, en les plaçant dans la groupBox des informations de commande. En parcourant les lignes et les colonnes de la DataGridView, je stocke les valeurs dans des variables pour les affecter aux champs correspondants des informations de commande.

PROCÉDURE ÉVÈNEMENTIELLE - DGVABONNEMENTSREVUE_ROWENTER

```
private void dgvAbonnementsRevue_RowEnter(object sender,
DataGridViewCellEventArgs e)
{
   DataGridViewRow row = dgvAbonnementsRevue.Rows[e.RowIndex];
   string id = row.Cells["Id"].Value.ToString();
   DateTime dateCommande = (DateTime)row.Cells["dateCommande"].Value;
   double montant = double.Parse(row.Cells["Montant"].Value.ToString());
   DateTime dateFinAbonnement =
(DateTime)row.Cells["DateFinAbonnement"].Value;
   txbCommandeRevueNumero.Text = id;
   txbCommandeRevueMontant.Text = montant.ToString();
   dtpCommandeRevue.Value = dateCommande;
   dtpCommandeRevueAbonnementFin.Value = dateFinAbonnement;
}
```

Pour ajouter la fonctionnalité "Ajouter" dans l'onglet des commandes de revues, j'ai d'abord mis en place la méthode "CreerAbonnementRevue" dans la classe "Access". Cette méthode utilise une requête HTTP POST pour écrire un nouvel abonnement dans la base de données. Elle prend en paramètres l'identifiant de la revue, la date de fin d'abonnement et l'identifiant de la revue. J'ai converti la date de fin d'abonnement au format JSON à l'aide de la bibliothèque Newtonsoft.Json. Ensuite, j'ai réalisé la requête HTTP POST pour envoyer les données du nouvel objet "Abonnement" sur le endpoint "abonnement/".

MÉTHODE - CREERABONNEMENTREVUE

```
public bool CreerAbonnementRevue(string id, DateTime dateFinAbonnement, string
idRevue)
{
   string jsonDateCommande = JsonConvert.SerializeObject(dateFinAbonnement, new
CustomDateTimeConverter());
   string jsonCreerAbonnementRevue = "{\"id\":\"" + id + "\", \"dateFinAbonnement\" : "
+ jsonDateCommande + ", \"idRevue\" : \"" + idRevue + "\"}";
   Console.WriteLine("jsonCreerAbonnementRevue" + jsonCreerAbonnementRevue);
   try
   {
      // récupération soit d'une liste vide (requête ok) soit de null (erreur)
      List<Abonnement> liste = TraitementRecup<Abonnement>(POST, "abonnement/" +
jsonCreerAbonnementRevue);
      return (liste != null);
   }
   catch (Exception ex)
   {
      Console.WriteLine(ex.Message);
   }
   return false;
}
```

Dans le "FrmMediatekController", j'invoque cette méthode, puis je mets en place la procédure événementielle "btnReceptionAbonnementRevueValider_Click". Cette dernière est déclenchée lors du clic sur le bouton "Ajouter" et enregistre un nouvel abonnement de revue dans la base de données. Pour que cette opération réussisse, les champs obligatoires tels que le numéro de commande, le montant, la date de commande et la date de fin d'abonnement doivent être renseignés.

Je commence par établir une condition "if" qui autorise ou non l'ajout du nouvel abonnement de revue et de la commande associée dans la base de données. Ensuite, je déclare des variables pour stocker les valeurs des champs de saisie, et j'attribue la nouvelle commande à la liste "Commande" dans la variable "commande", et le nouvel abonnement à la liste "Abonnement" dans la variable "abonnement".

Enfin, j'ajoute une autre condition "if" pour appeler la méthode "CreerCommande" qui intègre les données de la nouvelle commande. Le contrôleur est ensuite invoqué pour récupérer la méthode "CreerAbonnementRevue", où les données du nouvel objet "Abonnement" sont affectées. Enfin, j'actualise l'affichage de la grille contenant le nouvel enregistrement en appelant la méthode "AfficheReceptionCommandesLivre();". Une mesure de sécurité est également ajoutée pour empêcher l'ajout d'une commande si son identifiant existe déjà.

PROCÉDURE ÉVÈNEMENTIELLE – BTNRECEPTIONABONNEMENTREVUEVALIDER_CLICK

```
private void btnReceptionAbonnementRevueValider_Click(object sender, EventArgs
e)
{
   if (!txbCommandeRevueNumero.Text.Equals("") &&
      !txbCommandeRevueMontant.Text.Equals(""))
   {
      string idRevue = txbCommandesRevueNumRecherche.Text;
      string id = txbCommandeRevueNumero.Text;
      double montant = double.Parse(txbCommandeRevueMontant.Text);
      DateTime dateCommande = dtpCommandeRevue.Value;
      DateTime dateFinAbonnement = dtpCommandeRevueAbonnementFin.Value;
      Commande commande = new Commande(id, dateCommande, montant);
      var idCommandeRevueExistante = controller.GetCommandes(id);
      var idCommandeRevueNonExistante = !idCommandeRevueExistante.Any();
      if (idCommandeRevueNonExistante)
      {
         if (controller.CreerCommande(commande))
         {
            controller.CreerAbonnementRevue(id, dateFinAbonnement, idRevue);
            MessageBox.Show("La commande " + id + " a bien été enregistrée.",
"Information");
            AfficheReceptionAbonnementsRevue();
         }
      }
      else
      {
         MessageBox.Show("Le numéro de la commande existe déjà, veuillez saisir un
nouveau numéro.", "Erreur");
      }
   }
}
```

Je crée la fonction "SupprimerAbonnementRevue(Abonnement abonnement)" dans la classe "Access". Cette fonction utilise une requête HTTP DELETE avec le endpoint "abonnement/" et une chaîne JSON pour supprimer l'abonnement en utilisant son ID.

MÉTHODE - SUPPRIMERABONNEMENTREVUE(ABONNEMENT ABONNEMENT)

```
public bool SupprimerAbonnementRevue(Abonnement abonnement)
{
   string jsonSupprimerAbonnementRevue = "{\"id\":\"" + abonnement.Id + "\"}";
   try
   {
      List<Abonnement> liste = TraitementRecup<Abonnement>(DELETE,
"abonnement/" + jsonSupprimerAbonnementRevue);
      return (liste != null);
   }
   catch (Exception ex)
   {
      Console.WriteLine(ex.Message);
   }
   return false;
}
```

Pour supprimer un abonnement de revue, je crée d'abord une méthode appelée "ParutionDansAbonnement(DateTime dateCommande, DateTime dateFinAbonnement, DateTime dateParution)". Cette méthode vérifie si la date de parution d'un exemplaire est comprise entre la date de commande et la date de fin d'abonnement. Si c'est le cas, elle retourne "true".

Ensuite, j'appelle cette méthode dans la procédure événementielle "btnAbonnementRevueSupprimer_Click". Cette procédure permet de supprimer l'abonnement de la revue sélectionnée dans la DataGridView, à condition qu'elle ne comporte aucun exemplaire. Si la méthode "ParutionDansAbonnement" retourne "false" pour tous les exemplaires de l'abonnement sélectionné, alors l'abonnement peut être supprimé. Sinon, un message d'erreur est affiché indiquant qu'il existe des exemplaires liés à cet abonnement et qu'il ne peut pas être supprimé.

MÉTHODE - PARUTIONDANSABONNEMENT

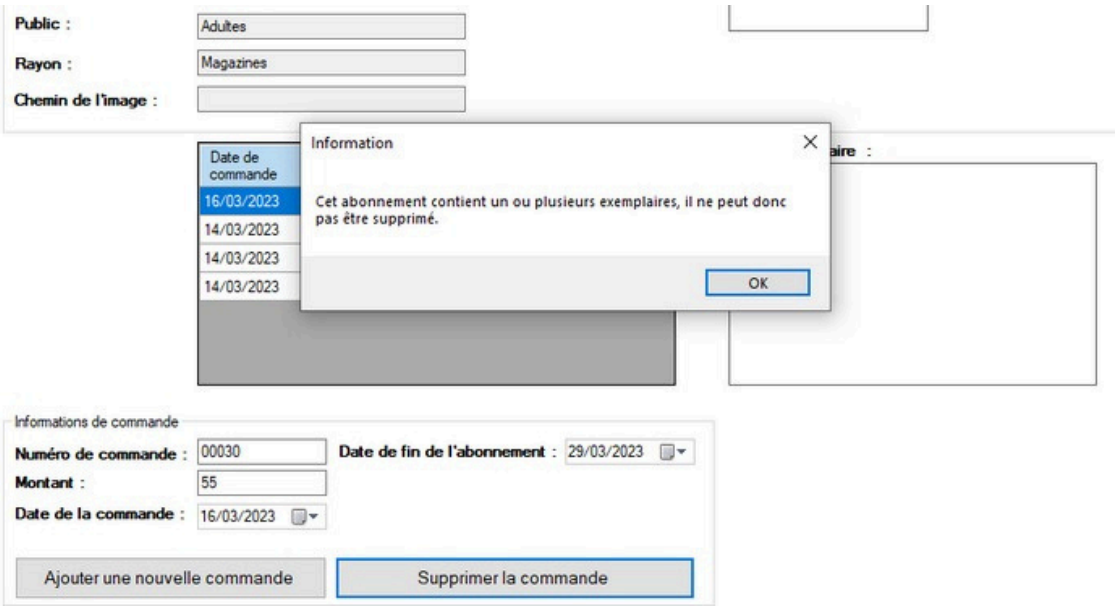
```
public bool ParutionDansAbonnement(DateTime dateCommande, DateTime
dateFinAbonnement, DateTime dateParution)
{
   return (DateTime.Compare(dateCommande, dateParution) < 0 &&
      DateTime.Compare(dateParution, dateFinAbonnement) < 0);
}
```

Dans la méthode `VerificationExemplaire(Abonnement abonnement)`, je vérifie si aucun exemplaire n'est rattaché à un abonnement de revue. Pour ce faire, j'appelle la méthode `ParutionDansAbonnement`.

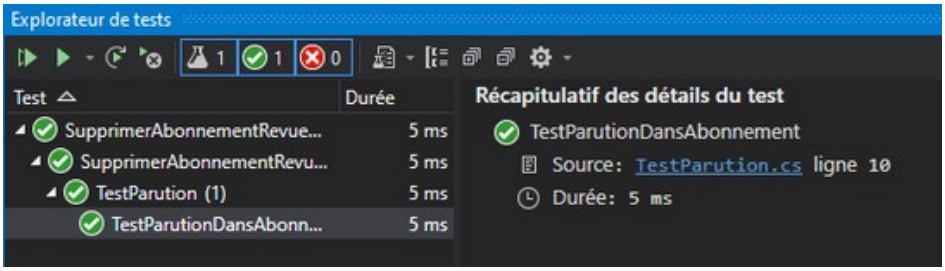
MÉTHODE - VERIFICATIONEXEMPLAIRE(ABONNEMENT ABONNEMENT)

```
public bool VerificationExemplaire(Abonnement abonnement)
{
  List<Exemplaire> lesExemplaires =
controller.GetExemplairesRevue(abonnement.IdRevue);
  bool datedeparution = false;
  foreach (Exemplaire exemplaire in lesExemplaires.Where(exemplaires =>
    ParutionDansAbonnement(abonnement.DateCommande,
abonnement.DateFinAbonnement,
    exemplaires.DateAchat)))
  {
    datedeparution = true;
  }
  return !datedeparution;
}
```

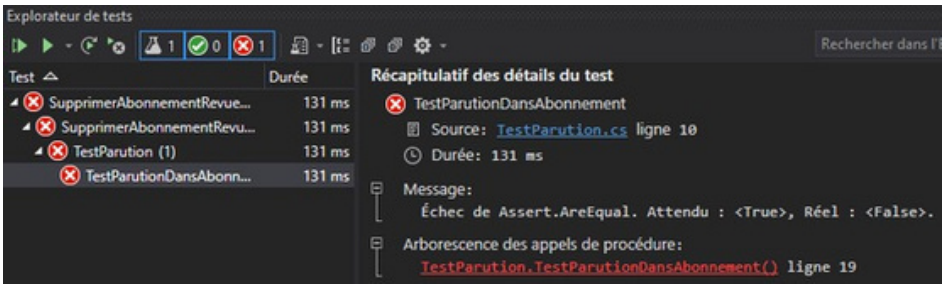
Dans la procédure évènementielle "btnAbonnementRevueSupprimer_Click", je vérifie d'abord si aucun exemplaire n'est associé à l'abonnement de la revue en appelant la méthode "VerificationExemplaire". Si aucun exemplaire n'est trouvé, je supprime alors l'abonnement de la revue. Sinon, j'affiche un message à l'utilisateur indiquant que l'abonnement ne peut pas être supprimé car il contient des exemplaires.



Dans le projet "Controller-ParutionTest", j'ai créé la méthode de test "TestParutionDansAbonnement" dans la classe "ParutionTest". J'ai défini des valeurs de type "DateTime" pour les variables "dateCommande", "dateFinAbonnement" et "dateParution". Ensuite, j'ai défini la valeur attendue comme "true" et j'ai obtenu le résultat réel en exécutant la méthode "ParutionDansAbonnement". Le test réussit lorsque la date de parution se situe entre la date de commande et la date de fin d'abonnement.



Lorsqu'elle n'est pas comprise entre les deux dates, le test échoue



TEST UNITAIRE - TESTPARUTIONDANSABONNEMENT()

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System;

[TestClass]
public class TestParution
{
  [TestMethod]
  public void TestParutionDansAbonnement()
  {
    DateTime dateCommande = new DateTime(2022, 10, 3);
    DateTime dateFinAbonnement = new DateTime(2022, 11, 3);
    DateTime dateParution = new DateTime(2022, 10, 4);
    bool resultatAttendu = true;
    bool resultatActuel = ParutionDansAbonnement(dateCommande,
dateFinAbonnement, dateParution);
    Assert.AreEqual(resultatAttendu, resultatActuel);
  }

  public bool ParutionDansAbonnement(DateTime dateCommande, DateTime
dateFinAbonnement, DateTime dateParution)
  {
    return (DateTime.Compare(dateCommande, dateParution) < 0 &&
DateTime.Compare(dateParution, dateFinAbonnement) < 0);
  }
}
```

Pour afficher une alerte lorsqu'un abonnement arrive à expiration dans moins de 30 jours, je crée une nouvelle fenêtre appelée "FrmAlerteFinAbonnement" dans le dossier "view". Cette fenêtre contient un titre indiquant les abonnements arrivant à échéance, une datagrid affichant la liste de ces abonnements, et un bouton "OK" pour fermer l'alerte.
Pour obtenir la liste des abonnements expirant dans les 30 prochains jours, je crée une nouvelle fonction dans l'API Rest PHP appelée "selectAllAbonnementsEcheance()". Cette fonction effectue une requête SQL pour sélectionner la date de fin de l'abonnement, l'identifiant de la revue et le titre du document. Elle réalise ensuite une jointure entre les tables "revue" et "abonnement" ainsi qu'une seconde jointure entre les tables "document" et "revue". Enfin, elle calcule la différence entre la date de fin de l'abonnement et la date actuelle avec la requête "WHERE DATEDIFF(CURRENT_DATE(), a.dateFinAbonnement) < 30" et trie les résultats par ordre croissant pour afficher les abonnements arrivant à expiration le plus rapidement en premier.

MÉTHODE - SELECTALLABONNEMENTSECHEANCE()

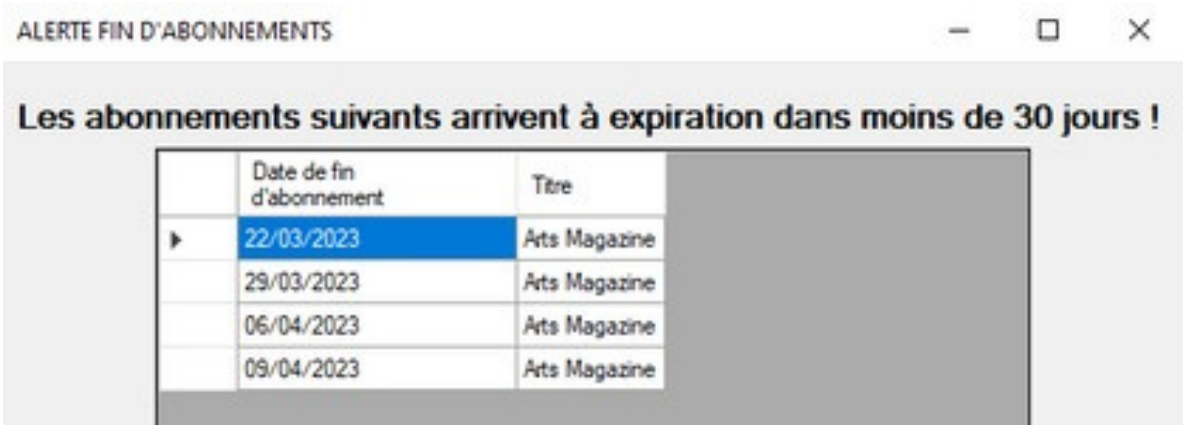
```
public function selectAllAbonnementsEcheance() {
   $req = "SELECT a.dateFinAbonnement, a.idRevue, d.titre ";
   $req .= "FROM abonnement a ";
   $req .= "JOIN revue r ON a.idRevue = r.id ";
   $req .= "JOIN document d ON r.id = d.id ";
   $req .= "WHERE DATEDIFF(CURRENT_DATE(), a.dateFinAbonnement) < 30 ";
   $req .= "ORDER BY a.dateFinAbonnement ASC; ";
   return $this->conn->queryAll($req);
}
```

Pour intégrer les abonnements arrivant à échéance dans l'application, j'ai ajouté une fonction `selectAllAbonnementsEcheance()` dans l'API PHP. Cette fonction effectue une requête SQL pour récupérer les abonnements avec une date de fin d'abonnement dans les 30 prochains jours, triés par date croissante. Ensuite, j'ai mis à jour la classe `Access` dans l'application C# avec un getter `GetAbonnementsEcheance()` qui récupère cette liste depuis l'API.

Dans la vue principale de l'application, j'ai inclus la fenêtre `FrmAlerteFinAbonnement` pour afficher ces abonnements. J'ai utilisé `.ShowDialog()` pour afficher cette fenêtre de manière modale.

Dans `FrmAlerteFinAbonnement`, j'ai créé une méthode `RemplirAbonnementsAEcheance(List<Abonnement> lesAbonnementsAEcheance)` pour remplir la grille avec les abonnements à échéance. Seules les colonnes pertinentes (titre et date de fin d'abonnement) sont affichées. J'ai également ajouté la fonctionnalité de tri sur les colonnes de la grille.

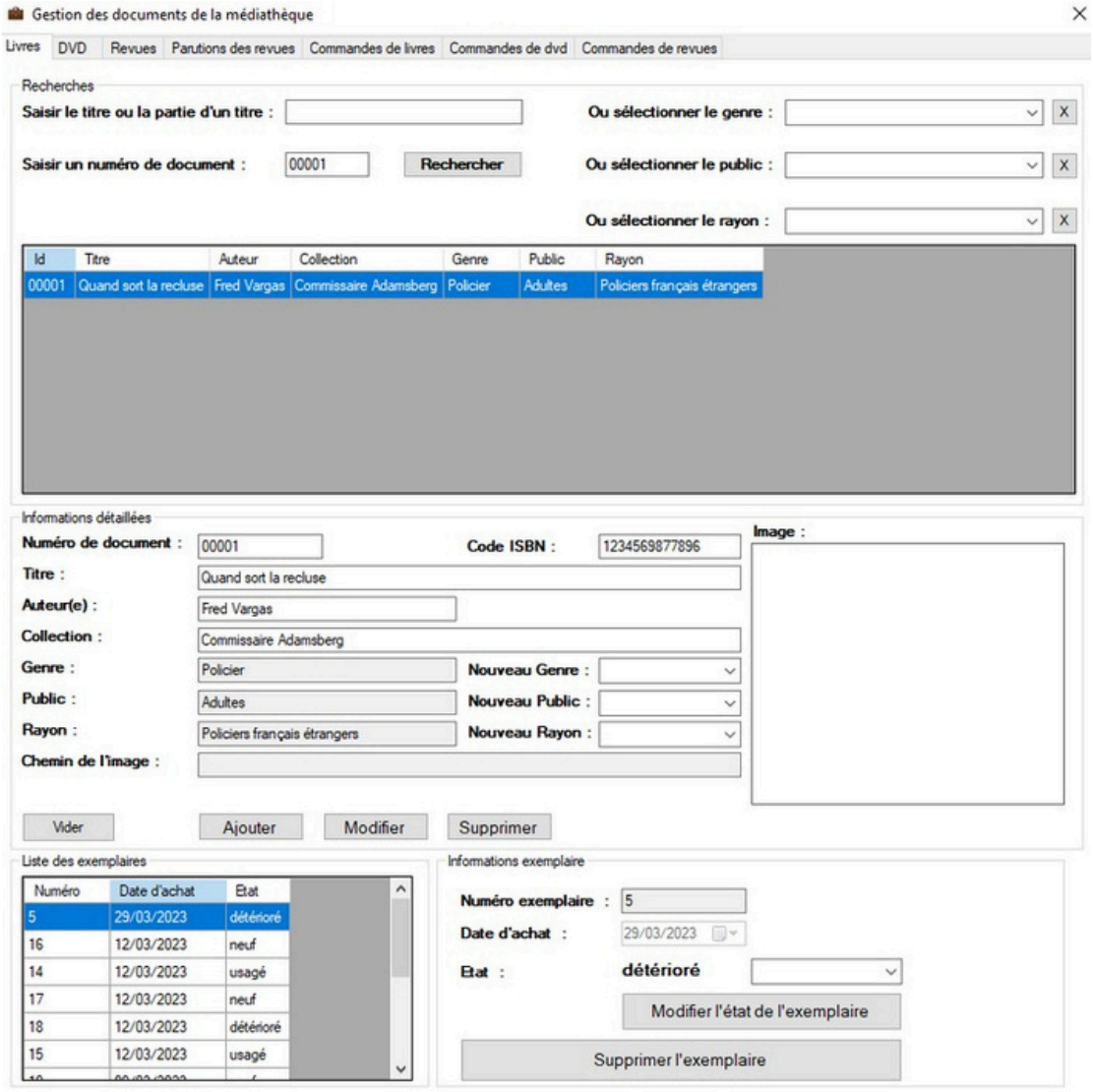
Enfin, j'ai ajouté un évènement sur le bouton "OK" pour fermer la fenêtre d'alerte lorsque l'utilisateur clique dessus.



MISSION 3 : GÉRER L'ÉTAT DES EXEMPLAIRES

A. ETAT EXEMPLAIRES - ONGLET LIVRES

Pour améliorer l'interface de l'onglet "Livres", j'ai ajouté deux groupbox. La première contient une grille affichant la liste des exemplaires du livre sélectionné. Dans la seconde, j'ai inclus des champs pour afficher les informations détaillées de l'exemplaire sélectionné, notamment le numéro d'exemplaire, la date d'achat et l'état actuel. J'ai également ajouté une combobox permettant à l'utilisateur de sélectionner le nouvel état de l'exemplaire. Deux boutons ont été ajoutés : "Modifier l'état de l'exemplaire" pour enregistrer les modifications une fois l'état sélectionné, et "Supprimer l'exemplaire" pour supprimer un exemplaire.



Dans l'onglet "Livres", j'ai amélioré l'interface en ajoutant deux groupbox. La première contient une grille affichant les exemplaires du livre sélectionné, tandis que la seconde présente en détail l'exemplaire choisi, incluant son numéro, sa date d'achat et son état actuel. J'ai intégré une combobox pour choisir le nouvel état de l'exemplaire, ainsi que deux boutons fonctionnels : "Modifier l'état de l'exemplaire" pour enregistrer les modifications et "Supprimer l'exemplaire" pour retirer un exemplaire.

MÉTHODE - REMPLIREXEMPLAIRESLIVRE

```
private void RemplirExemplairesLivre(List<Exemplaire> lesExemplaires)
{
   if (lesExemplaires != null)
   {
      bdgExemplairesLivre.DataSource = lesExemplaires;
      dgvExemplairesLivre.DataSource = bdgExemplairesLivre;
      dgvExemplairesLivre.Columns["photo"].Visible = false;
      dgvExemplairesLivre.Columns["idEtat"].Visible = false;
      dgvExemplairesLivre.Columns["id"].Visible = false;
      dgvExemplairesLivre.AutoSizeColumnsMode =
DataGridViewAutoSizeColumnsMode.AllCells;
      dgvExemplairesLivre.Columns[0].HeaderCell.Value = "Numéro";
      dgvExemplairesLivre.Columns[2].HeaderCell.Value = "Date d'achat";
      dgvExemplairesLivre.Columns[5].HeaderCell.Value = "Etat";
   }
   else
   {
      dgvExemplairesLivre.DataSource = null;
   }
}
```

Pour récupérer les données sur les exemplaires d'un livre, j'ajoute une nouvelle fonction à l'API Rest PHP. Cette fonction, nommée "selectAllExemplairesDocument($id)", prend en paramètre l'identifiant du livre. Elle effectue une requête pour sélectionner les données des exemplaires, ainsi que le libellé de l'état de chaque exemplaire. Une jointure entre les tables "exemplaire" et "etat" permet de récupérer l'identifiant de l'état de chaque exemplaire. Les résultats sont ensuite triés par date d'achat, dans l'ordre décroissant. Cette fonction est appelée dans un nouveau cas de la méthode "select One", où les données sont récupérées depuis le point de terminaison "exemplairesdocument".

MÉTHODE - SELECTALLEXEMPLAIRESDOCUMENT($ID)

```
public function selectAllExemplairesDocument($id){
$param = array(
            "id" => $id
    );
    $req = "select ex.id, ex.numero, ex.dateAchat, ex.photo, ex.idEtat, et.libelle ";
    $req .= "from exemplaire ex JOIN etat et ON ex.idEtat = et.id ";
    $req .= "where ex.id = :id ";
    $req .= "order by ex.dateAchat DESC";
    return $this->conn->query($req, $param);
}
```

Dans la classe "Access" de l'application C#, j'ai ajouté le getter "GetExemplairesDocument(string idDocument)". Ce getter utilise une requête HTTP GET sur le point de terminaison "exemplairesdocument/idDocument", où "idDocument" est l'identifiant du document sélectionné. Ensuite, dans le contrôleur, j'ai appelé cette méthode, puis j'ai créé la méthode "AfficheExemplairesLivres()" dans la vue de l'onglet "Livre". Cette méthode récupère l'identifiant du document à partir du champ "txbLivresNumRecherche.Text" et affecte la liste des exemplaires du document sélectionné à la variable "lesExemplairesDocument". Ensuite, j'ai appelé la méthode "RemplirExemplairesLivre" pour afficher la liste des exemplaires dans la grille.

La méthode "AfficheExemplairesLivres()" est liée à l'événement de clic sur le bouton "Rechercher", de sorte que la liste des exemplaires s'affiche automatiquement lorsque l'utilisateur recherche un document. Enfin, j'ai ajouté la procédure événementielle "dgvExemplairesLivre_ColumnHeaderMouseClick" pour permettre à l'utilisateur de trier les colonnes, notamment la date d'achat, dans l'ordre inverse de la chronologie.

PROCÉDURE ÉVÈNEMENTIELLE - DGVEXEMPLAIRESLIVRES_COLUMNHEADERMOUSECLICK

```
private void dgvExemplairesLivres_ColumnHeaderMouseClick(object sender,
DataGridViewCellMouseEventArgs e)
{
   string titreColonne = dgvExemplairesLivre.Columns[e.ColumnIndex].HeaderText;
   List<Exemplaire> sortedList = new List<Exemplaire>();
   switch (titreColonne)
   {
      case "Date d'achat":
         sortedList = lesExemplairesDocument.OrderBy(o =>
o.DateAchat).Reverse().ToList();
         break;
      case "Numéro":
         sortedList = lesExemplairesDocument.OrderBy(o => o.Numero).ToList();
         break;
      case "Etat":
         sortedList = lesExemplairesDocument.OrderBy(o => o.Libelle).ToList();
         break;
   }
   RemplirExemplairesLivre(sortedList);
}
```

Pour remplir la combobox de suivi de l'état d'un exemplaire, vous pouvez utiliser des méthodes similaires à celles utilisées pour suivre l'étape d'une commande. Commencez par créer une méthode "RemplirCbxEtatLibelleExemplaireLivre(string etatExemplaireLivre)" qui va remplir la combobox en fonction des états possibles de l'exemplaire et leur libellé correspondant. Dans cette méthode, vous pouvez utiliser une structure conditionnelle "if" pour affecter le libellé correspondant à l'état de l'exemplaire et ajouter les libellés des états possibles.

MÉTHODE - REMPLIRCBXETATLIBELLEEXEMPLAIRELIVRE(STRING ETATEXEMPLAIRELIVRE)

```
private void RemplirCbxEtatLibelleExemplaireLivre(string etatExemplaireLivre)
{
   cbxEtatLibelleExemplaireLivre.Items.Clear();
   if (etatExemplaireLivre == "neuf")
   {
      cbxEtatLibelleExemplaireLivre.Items.Add("usagé");
      cbxEtatLibelleExemplaireLivre.Items.Add("détérioré");
      cbxEtatLibelleExemplaireLivre.Items.Add("inutilisable");
   }
   else if (etatExemplaireLivre == "usagé")
   {
      cbxEtatLibelleExemplaireLivre.Text = "";
      cbxEtatLibelleExemplaireLivre.Items.Add("neuf");
      cbxEtatLibelleExemplaireLivre.Items.Add("détérioré");
      cbxEtatLibelleExemplaireLivre.Items.Add("inutilisable");
   }
   else if (etatExemplaireLivre == "détérioré")
   {
      cbxEtatLibelleExemplaireLivre.Text = "";
      cbxEtatLibelleExemplaireLivre.Items.Add("neuf");
      cbxEtatLibelleExemplaireLivre.Items.Add("usagé");
      cbxEtatLibelleExemplaireLivre.Items.Add("inutilisable");
   }
   else if (etatExemplaireLivre == "inutilisable")
   {
      cbxEtatLibelleExemplaireLivre.Text = "";
      cbxEtatLibelleExemplaireLivre.Items.Add("neuf");
      cbxEtatLibelleExemplaireLivre.Items.Add("usagé");
      cbxEtatLibelleExemplaireLivre.Items.Add("détérioré");
   }
}
```

Pour obtenir l'ID de l'état d'un exemplaire selon son libellé, je crée la méthode "GetIdEtat(string libelle)" dans "Access". Elle récupère tous les états du document, puis parcourt la liste pour trouver l'ID correspondant au libellé donné. Ensuite, dans "lblEtatExemplaireLivre_TextChanged", je récupère le libellé de l'état à partir du label et j'appelle "GetIdEtat". Cette approche permet une gestion dynamique des ID d'état en fonction du libellé.

MÉTHODE - GETIDETAT(STRING LIBELLE)

```
private void dgvExemplairesLivre_RowEnter(object sender,
DataGridViewCellEventArgs e)
{
   DataGridViewRow row = dgvExemplairesLivre.Rows[e.RowIndex];
   string id = row.Cells["Id"].Value.ToString();
   string numero = row.Cells["Numero"].Value.ToString();
   DateTime dateAchat = (DateTime)row.Cells["dateAchat"].Value;
   string libelle = row.Cells["Libelle"].Value.ToString();
   txbExemplaireLivresNumero.Text = numero;
   dtpDateAchatExemplaireLivre.Value = dateAchat;
   lblEtatExemplaireLivre.Text = libelle;
   gbxEtatExemplaireLivre.Enabled = true;
}
```

Pour améliorer la fonctionnalité de modification de l'état d'un exemplaire sélectionné, j'ajuste la méthode "updateOne" dans l'API Rest PHP. Plutôt que de modifier l'état de tous les exemplaires d'un document, je souhaite modifier spécifiquement l'état d'un exemplaire. J'ajoute donc un nouveau cas pour gérer cette logique, en spécifiant la table et les champs pour cibler l'exemplaire concerné par son identifiant et son numéro. Ce changement me permettra de mettre à jour l'état de manière précise pour un exemplaire donné, sans affecter les autres exemplaires du document.

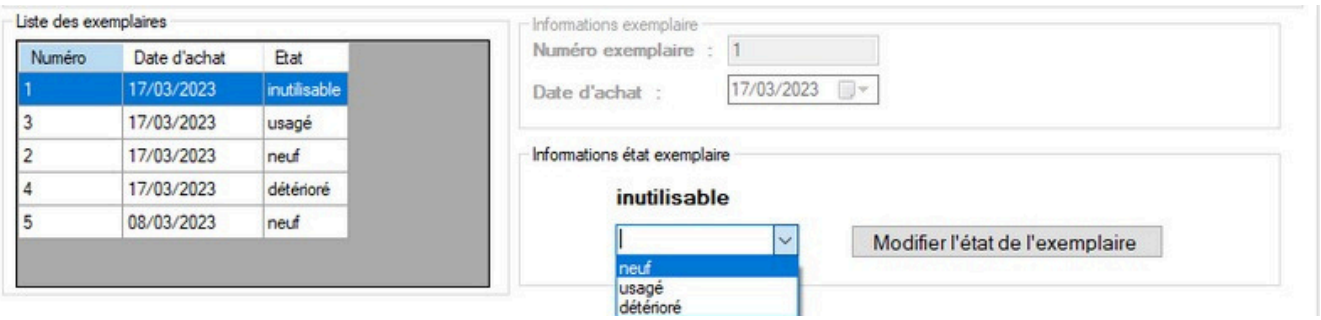
FONCTION - UPDATEONE

```
public function updateOne($table, $id, $champs){
    if($this->conn != null && $champs != null){
        switch($table){
            case "exemplairesdocument":
                $champsExemplaire = [
                    'id' => $champs['Id'],
                    'numero' => $champs['Numero'],
                    'dateAchat' => $champs['DateAchat'],
                    'photo' => $champs['Photo'],
                    'idEtat' => $champs['IdEtat']
                ];
                $requete = "UPDATE exemplaire SET ";
                foreach ($champsExemplaire as $key => $value) {
                    $requete .= "$key=:$key,";
                }
                $requete = substr($requete, 0, strlen($requete)-1);
                $requete .= " WHERE id=:id AND numero=:numero;";
                $champsExemplaire['numero'] = $id;
                $updateExemplaire = $this->conn->execute($requete, $champsExemplaire);
                if(!$updateExemplaire){
                    return null;
                }
                break;
            default:
                $champs['id'] = $id;
                $requete = "UPDATE $table SET ";
                foreach ($champs as $key => $value) {
                    $requete .= "$key=:$key,";
                }
                $requete = substr($requete, 0, strlen($requete)-1);
                $requete .= " WHERE id=:id;";
                return $this->conn->execute($requete, $champs);
        }
    } else {
        return null;
    }
}
```

Dans la classe "Access" de l'application C#, j'ai créé la méthode "ModifierEtatExemplaireDocument(Exemplaire exemplaire)" pour modifier l'état d'un exemplaire en base de données en utilisant la méthode HTTP PUT. J'ai ajusté le endpoint "exemplairesdocument/" pour inclure le numéro de l'exemplaire plutôt que l'ID du document. Ensuite, dans le "FrmMediatekController", j'ai appelé cette méthode et créé la procédure événementielle "btnEtatExemplaireLivreModifier_Click" pour déclencher la modification lors du clic sur le bouton correspondant. J'ai récupéré les valeurs des champs, créé un nouvel objet "Exemplaire" et appelé la méthode de contrôleur pour effectuer la modification.

PROCÉDURE ÉVÈNEMENTIELLE - BTNETATEXEMPLAIRELIVREMODIFIER_CLICK

```
private void btnEtatExemplaireLivreModifier_Click(object sender, EventArgs e)
{
   string idDocument = txbLivresNumRecherche.Text;
   int numero = int.Parse(txbExemplaireLivresNumero.Text);
   DateTime dateAchat = dtpDateAchatExemplaireLivre.Value;
   string photo = "";
   string idEtat = GetIdEtat(cbxEtatLibelleExemplaireLivre.Text);
   string libelle = cbxEtatLibelleExemplaireLivre.SelectedItem.ToString();
   Exemplaire exemplaire = new Exemplaire(numero, dateAchat, photo, idEtat,
idDocument, libelle);

   if (MessageBox.Show("Voulez-vous modifier l'état de l'exemplaire " +
exemplaire.Numero + " en " + libelle + " ?", "Confirmation", MessageBoxButtons.YesNo)
== DialogResult.Yes)
   {
      controller.ModifierEtatExemplaireDocument(exemplaire);
      MessageBox.Show("L'état de l'exemplaire " + exemplaire.Numero + " a bien été
modifié.", "Information");
      AfficheExemplairesLivres();
   }
}
```



Pour supprimer un exemplaire de document de la base de données, j'ai créé la méthode "SupprimerExemplaireDocument(Exemplaire exemplaire)" dans la classe "Access". Cette méthode utilise une chaîne au format JSON pour spécifier l'exemplaire à supprimer, en utilisant à la fois l'ID de l'exemplaire (qui correspond à l'ID du document) et le numéro de l'exemplaire. Ensuite, j'ai effectué une requête HTTP DELETE en utilisant le point de terminaison "exemplaire/" auquel j'ai ajouté la chaîne JSON correspondante.

MÉTHODE - SUPPRIMEREXEMPLAIREDOCUMENT(EXEMPLAIRE EXEMPLAIRE)

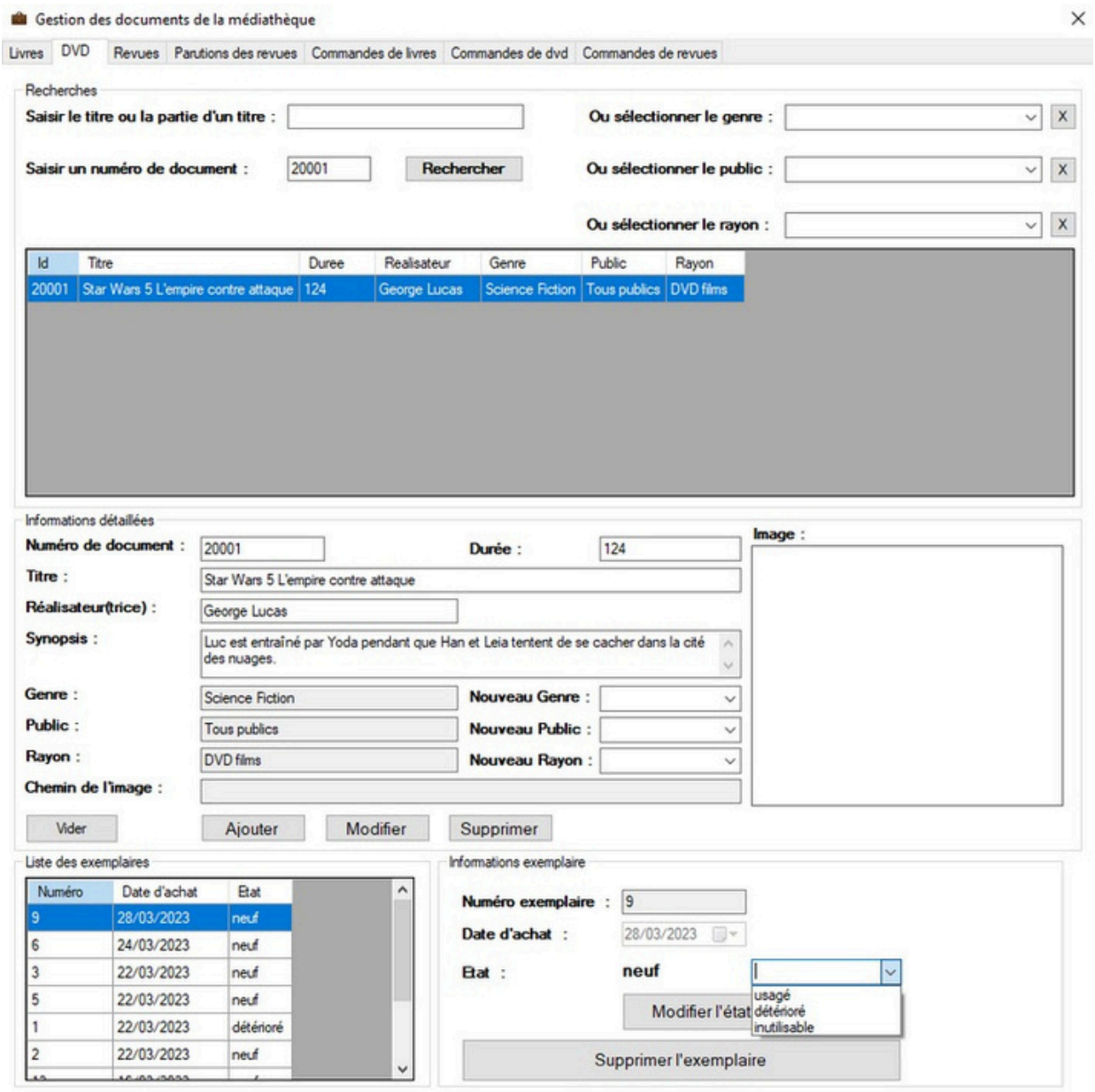
```
public bool SupprimerExemplaireDocument(Exemplaire exemplaire)
{
   string jsonSupprimerExemplaireDocument = "{\"id\":\"" + exemplaire.Id +
"\",\"numero\":\"" + exemplaire.Numero + "\"}";
   Console.WriteLine("jsonSupprimerExemplaireDocument" +
jsonSupprimerExemplaireDocument);
   try
   {
      // récupération soit d'une liste vide (requête ok) soit de null (erreur)
      List<Exemplaire> liste = TraitementRecup<Exemplaire>(DELETE, "exemplaire/" +
jsonSupprimerExemplaireDocument);
      return (liste != null);
   }
   catch (Exception ex)
   {
      Console.WriteLine(ex.Message);
   }
   return false;
}
```

Cette méthode "SupprimerExemplaireDocument" prend un objet "Exemplaire" en paramètre et essaie de supprimer cet exemplaire de la base de données en envoyant une requête HTTP DELETE avec les informations de l'exemplaire. Si la suppression réussit, elle retourne true, sinon elle retourne false. En cas d'erreur, elle affiche un message dans la console.

PROCÉDURE ÉVÈNEMENTIELLE - BTNEXEMPLAIRELIVRESUPPRIMER_CLICK

```
private void btnExemplaireLivreSupprimer_Click(object sender, EventArgs e)
{
   if (dgvExemplairesLivre.SelectedRows.Count > 0)
   {
      Exemplaire exemplaire =
(Exemplaire)bdgExemplairesLivre.List[bdgExemplairesLivre.Position];
      if (MessageBox.Show("Voulez-vous supprimer l'exemplaire " + exemplaire.Numero
+ " du livre " + exemplaire.Id + " ?", "Confirmation de suppression",
MessageBoxButtons.YesNo) == DialogResult.Yes)
      {
         controller.SupprimerExemplaireDocument(exemplaire);
         MessageBox.Show("L'exemplaire " + exemplaire.Numero + " a bien été supprimé.",
"Information");
         AfficheExemplairesLivres();
      }
   }
   else
   {
      MessageBox.Show("Une ligne doit être sélectionnée.", "Information");
   }
}
```

B. ETAT EXEMPLAIRES - ONGLET DVD



- Ajout d'une GroupBox pour afficher la liste des exemplaires du DVD sélectionné.
- Insertion d'une deuxième GroupBox pour afficher les détails de l'exemplaire sélectionné, y compris le numéro d'exemplaire, la date d'achat et l'état actuel.
- Intégration d'une ComboBox pour sélectionner le nouvel état de l'exemplaire.
- Ajout d'un bouton "Modifier l'état de l'exemplaire" pour appliquer les modifications d'état.
- Inclusion d'un bouton "Supprimer l'exemplaire" pour effacer un exemplaire.
- Implémentation de la méthode "RemplirExemplairesDvd(List<Exemplaire> lesExemplaires)" pour remplir la grille des exemplaires avec les détails pertinents de la classe "Dvd" du dossier "Model", en masquant les colonnes non nécessaires afin de ne montrer que le numéro, la date d'achat et l'état des exemplaires.

MÉTHODE - REMPLIREXEMPLAIRESDVD(LIST LESEXEMPLAIRES)

```
private void RemplirExemplairesDvd(List<Exemplaire> lesExemplaires)
{
   if (lesExemplaires != null)
   {
      bdgExemplairesDvd.DataSource = lesExemplaires;
      dgvExemplairesDvd.DataSource = bdgExemplairesDvd;
      dgvExemplairesDvd.Columns["photo"].Visible = false;
      dgvExemplairesDvd.Columns["idEtat"].Visible = false;
      dgvExemplairesDvd.Columns["id"].Visible = false;
      dgvExemplairesDvd.AutoSizeColumnsMode =
DataGridViewAutoSizeColumnsMode.AllCells;
      dgvExemplairesDvd.Columns[0].HeaderCell.Value = "Numéro";
      dgvExemplairesDvd.Columns[2].HeaderCell.Value = "Date d'achat";
      dgvExemplairesDvd.Columns[5].HeaderCell.Value = "Etat";
   }
   else
   {
      dgvExemplairesDvd.DataSource = null;
   }
}
```

Pour afficher les exemplaires d'un DVD, j'ai créé la méthode "AfficheExemplairesDvd()" dans la vue de l'onglet "DVD". Cette méthode récupère d'abord l'identifiant du DVD à partir du champ de recherche, puis elle récupère la liste des exemplaires associés à ce DVD. Ensuite, elle appelle la méthode "RemplirExemplairesDvd()" pour afficher ces exemplaires dans la grille. J'ai également lié cette méthode à l'événement de clic sur le bouton de recherche pour afficher la liste des exemplaires dès que l'utilisateur recherche un DVD. De plus, j'ai ajouté une procédure événementielle pour permettre à l'utilisateur de trier les colonnes de la grille, en particulier pour trier la date d'achat dans l'ordre inverse de la chronologie.

PROCÉDURE ÉVÈNEMENTIELLE - DGVEXEMPLAIRESLIVRES_COLUMNHEADERMOUSECLICK

```
private void dgvExemplairesDvd_ColumnHeaderMouseClick(object sender,
DataGridViewCellMouseEventArgs e)
{
   string titreColonne = dgvExemplairesDvd.Columns[e.ColumnIndex].HeaderText;
   List<Exemplaire> sortedList = new List<Exemplaire>();
   switch (titreColonne)
   {
      case "Date d'achat":
         sortedList = lesExemplairesDocument.OrderBy(o =>
o.DateAchat).Reverse().ToList();
         break;
      case "Numéro":
         sortedList = lesExemplairesDocument.OrderBy(o => o.Numero).ToList();
         break;
      case "Etat":
         sortedList = lesExemplairesDocument.OrderBy(o => o.Libelle).ToList();
         break;
   }
   RemplirExemplairesDvd(sortedList);
}
```

Pour uniformiser la gestion des états des exemplaires, j'ai reproduit les mêmes démarches déjà utilisées pour les livres. J'ai ainsi créé une méthode nommée "RemplirCbxEtatLibelleExemplaireDvd(string etatExemplaireDvd)" chargée de remplir une combobox avec les différents états possibles pour un exemplaire de DVD, en fonction de l'état actuel de cet exemplaire. Cette méthode utilise des conditions "if" pour déterminer l'état en cours et ajoute les options d'état correspondantes à la combobox.

MÉTHODE - REMPLIRCBXETATLIBELLEEXEMPLAIREDVD(STRING ETATEXEMPLAIREDVD)

```
private void RemplirCbxEtatLibelleExemplaireDvd(string etatExemplaireDvd)
{
   cbxEtatLibelleExemplaireDvd.Items.Clear();
   if (etatExemplaireDvd == "neuf")
   {
      cbxEtatLibelleExemplaireDvd.Items.Add("usagé");
      cbxEtatLibelleExemplaireDvd.Items.Add("détérioré");
      cbxEtatLibelleExemplaireDvd.Items.Add("inutilisable");
   }
   else if (etatExemplaireDvd == "usagé")
   {
      cbxEtatLibelleExemplaireDvd.Text = "";
      cbxEtatLibelleExemplaireDvd.Items.Add("neuf");
      cbxEtatLibelleExemplaireDvd.Items.Add("détérioré");
      cbxEtatLibelleExemplaireDvd.Items.Add("inutilisable");
   }
   else if (etatExemplaireDvd == "détérioré")
   {
      cbxEtatLibelleExemplaireDvd.Text = "";
      cbxEtatLibelleExemplaireDvd.Items.Add("neuf");
      cbxEtatLibelleExemplaireDvd.Items.Add("usagé");
      cbxEtatLibelleExemplaireDvd.Items.Add("inutilisable");
   }
   else if (etatExemplaireDvd == "inutilisable")
   {
      cbxEtatLibelleExemplaireDvd.Text = "";
      cbxEtatLibelleExemplaireDvd.Items.Add("neuf");
      cbxEtatLibelleExemplaireDvd.Items.Add("usagé");
      cbxEtatLibelleExemplaireDvd.Items.Add("détérioré");
   }
}
```

Lorsque l'utilisateur sélectionne un exemplaire dans la grille, les informations sur son état et les modifications d'état possibles s'affichent dans la groupbox des informations d'exemplaire. Pour ce faire, j'appelle la méthode dans la procédure évènementielle "lblEtatExemplaireDvd_TextChanged", où la variable "etatExemplaireDvd" récupère la valeur du label "EtatExemplaireDvd.Text". En fonction de cette variable, le remplissage de la combobox se fait selon les conditions établies précédemment. Ensuite, pour afficher les informations d'état et permettre les modifications, je crée la méthode "dgvExemplairesDvd_RowEnter". Cette méthode affecte la valeur de chaque cellule de la colonne sélectionnée aux champs correspondants, et active la groupbox permettant de sélectionner un nouvel état et de le modifier.

MÉTHODE - DGVEXEMPLAIRESDVD_ROWENTER

```
private void dgvExemplairesDvd_RowEnter(object sender,
DataGridViewCellEventArgs e)
{
  DataGridViewRow row = dgvExemplairesDvd.Rows[e.RowIndex];
  string id = row.Cells["Id"].Value.ToString();
  string numero = row.Cells["Numero"].Value.ToString();
  DateTime dateAchat = (DateTime)row.Cells["dateAchat"].Value;
  string libelle = row.Cells["Libelle"].Value.ToString();
  txbExemplaireDvdNumero.Text = numero;
  dtpDateAchatExemplaireDvd.Value = dateAchat;
  lblEtatExemplaireDvd.Text = libelle;
  gbxEtatExemplaireDvd.Enabled = true;
}
```

Pour ajouter la fonctionnalité de modification de l'état d'un exemplaire, je crée la procédure événementielle "btnEtatExemplaireDvdModifier_Click", activée par le clic sur le bouton "Modifier l'état de l'exemplaire". En récupérant les valeurs des champs, je crée un nouvel objet "Exemplaire". Ensuite, j'appelle la méthode du contrôleur "ModifierEtatExemplaireDocument(exemplaire)" pour appliquer les modifications.

PROCÉDURE ÉVÈNEMENTIELLE - BTNETATEXEMPLAIREDVDMODIFIER_CLICK

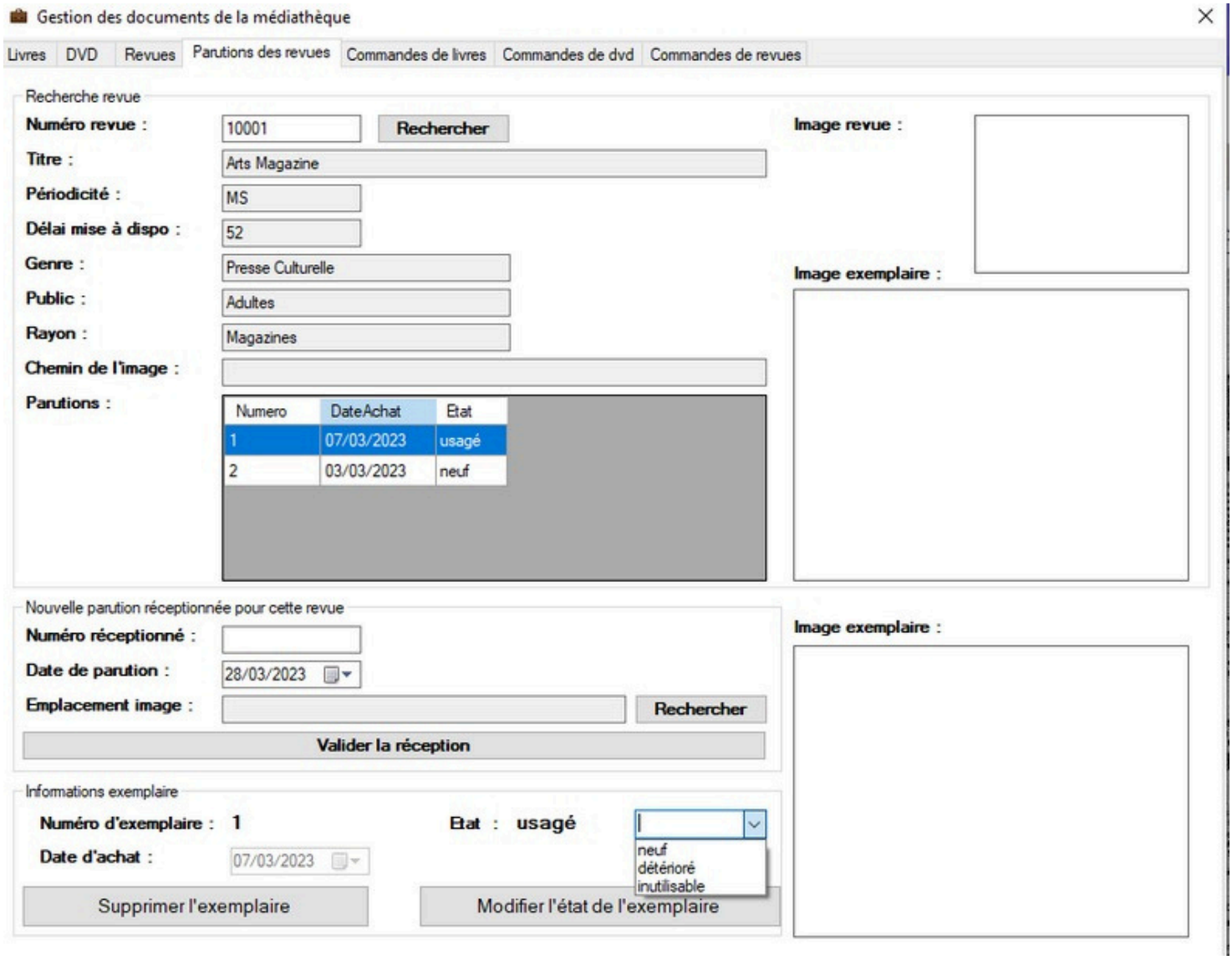
```
private void btnEtatExemplaireDvdModifier_Click(object sender, EventArgs e)
{
  string idDocument = txbDvdNumRecherche.Text;
  int numero = int.Parse(txbExemplaireDvdNumero.Text);
  DateTime dateAchat = dtpDateAchatExemplaireDvd.Value;
  string photo = "";
  string idEtat = GetIdEtat(cbxEtatLibelleExemplaireDvd.Text);
  try
  {
    string libelle = cbxEtatLibelleExemplaireDvd.SelectedItem.ToString();
    Exemplaire exemplaire = new Exemplaire(numero, dateAchat, photo, idEtat,
idDocument, libelle);
    if (MessageBox.Show("Voulez-vous modifier l'état de l'exemplaire " +
exemplaire.Numero + " en " + libelle + " ?", "Confirmation", MessageBoxButtons.YesNo)
== DialogResult.Yes)
    {
      controller.ModifierEtatExemplaireDocument(exemplaire);
      MessageBox.Show("L'état de l'exemplaire " + exemplaire.Numero + " a bien été
modifié.", "Information");
      AfficheExemplairesDvd();
    }
  }
  catch (NullReferenceException)
  {
    MessageBox.Show("Le nouvel état de l'exemplaire doit être sélectionné.",
"Information");
  }
}
```

Pour supprimer un exemplaire de DVD, j'ai créé la procédure évènementielle "btnExemplaireDvdSupprimer_Click". Cette procédure permet à l'utilisateur de supprimer un exemplaire du DVD sélectionné. Elle comprend une condition "if" qui vérifie si une ligne est sélectionnée dans la grille des exemplaires. Ensuite, elle affecte l'objet "Exemplaire" sélectionné dans la liste à la variable "exemplaire". Enfin, elle appelle la méthode "SupprimerExemplaireDocument(exemplaire)" du contrôleur pour enregistrer la suppression de l'exemplaire dans la base de données, puis elle affiche la liste mise à jour des exemplaires du DVD.

PROCÉDURE ÉVÈNEMENTIELLE - BTNEXEMPLAIREDVDSUPPRIMER_CLICK

```
private void btnExemplaireDvdSupprimer_Click(object sender, EventArgs e)
{
  if (dgvExemplairesDvd.SelectedRows.Count > 0)
  {
    Exemplaire exemplaire =
(Exemplaire)bdgExemplairesDvd.List[bdgExemplairesDvd.Position];
    if (MessageBox.Show("Voulez-vous supprimer l'exemplaire " + exemplaire.Numero
+ " du DVD " + exemplaire.Id + " ?", "Confirmation de suppression",
MessageBoxButtons.YesNo) == DialogResult.Yes)
    {
      controller.SupprimerExemplaireDocument(exemplaire);
      MessageBox.Show("L'exemplaire " + exemplaire.Numero + " a bien été supprimé.",
"Information");
      AfficheExemplairesDvd();
    }
  }
  else
  {
    MessageBox.Show("Une ligne doit être sélectionnée.", "Information");
  }
}
```

C. ETAT EXEMPLAIRES - ONGLET REVUES



Pour commencer, j'ai ajouté une groupbox dans l'interface de l'onglet "parution" pour afficher les informations de l'exemplaire et permettre la gestion de son état. Cela inclut le numéro d'exemplaire, la date d'achat et un libellé affichant l'état actuel, ainsi qu'une combobox pour sélectionner un nouvel état. J'ai également ajouté des boutons "Modifier l'état de l'exemplaire" et "Supprimer l'exemplaire" pour effectuer ces actions.

Ensuite, dans la classe "Access", j'ai créé une nouvelle méthode "CreerExemplaireRevue(Exemplaire exemplaire)" car la méthode existante "CreerExemplaire" ne fonctionnait pas lorsque j'ajoutais un exemplaire à une revue. Malgré mes efforts pour résoudre ce problème, j'ai dû créer cette nouvelle méthode pour le gérer. J'ai ensuite adapté le code de la vue dans l'onglet "parution" en conséquence.

MÉTHODE - CREEREXEMPLAIREREVUE(EXEMPLAIRE EXEMPLAIRE)

```
public bool CreerExemplaireRevue(string id, int numero, DateTime dateAchat, string
photo, string idEtat)
{
   string jsonDateAchat = JsonConvert.SerializeObject(dateAchat, new
CustomDateTimeConverter());
   string jsonCreerExemplaireRevue = "{\"id\":\"" + id + "\", \"numero\":\"" + numero + "\",
\"dateAchat\" : " + jsonDateAchat + ", \"photo\" : \"" + photo + "\" , \"idEtat\" : \"" + idEtat+
"\"}";
   try
   {
      List<Abonnement> liste = TraitementRecup<Abonnement>(POST, "exemplaire/" +
jsonCreerExemplaireRevue);
      return (liste != null);
   }
```

La modification des colonnes dans l'interface de l'onglet "parution" comprend le changement de la colonne "Photo" en "Etat", qui affiche désormais l'état de l'exemplaire sélectionné. Dans la méthode "RemplirReceptionExemplairesListe(List<Exemplaire> exemplaires)", la colonne "photo" est masquée en la définissant sur "false". Dans la procédure événementielle "dgvExemplairesListe_ColumnHeaderMouseClick", le case "Photo" est modifié en "Etat", et "Libelle" lui est affecté pour obtenir le libellé de l'état des exemplaires.

En ce qui concerne le remplissage de la combobox pour le suivi de l'état d'un exemplaire de revue, les mêmes méthodes que celles utilisées pour le suivi de l'état d'un livre ou d'un DVD sont employées. La méthode "RemplirCbxEtatLibelleExemplaireRevue(string etatExemplaireRevue)" remplit la combobox selon les états possibles de l'exemplaire et le libellé correspondant. Une condition "if" est créée pour affecter le libellé correspondant à un des états d'exemplaire, et les libellés des états possibles sont ajoutés en conséquence.

MÉTHODE - REMPLIRCBXETATLIBELLEEXEMPLAIREDVD(STRING ETATEXEMPLAIREREVUE)

```
public bool CreerExemplaireRevue(string id, int numero, DateTime dateAchat, string
photo, string idEtat)
{
   string jsonDateAchat = JsonConvert.SerializeObject(dateAchat, new
CustomDateTimeConverter());
   string jsonCreerExemplaireRevue = "{\"id\":\"" + id + "\", \"numero\":\"" + numero + "\",
\"dateAchat\" : " + jsonDateAchat + ", \"photo\" : \"" + photo + "\" , \"idEtat\" : \"" + idEtat+
"\"}";
   try
   {
      List<Abonnement> liste = TraitementRecup<Abonnement>(POST, "exemplaire/" +
jsonCreerExemplaireRevue);
      return (liste != null);
   }
```

```
private void RemplirCbxEtatLibelleExemplaireRevue(string etatExemplaireRevue)
{
   cbxEtatLibelleExemplaireRevue.Items.Clear();
   if (etatExemplaireRevue == "neuf")
   {
      cbxEtatLibelleExemplaireRevue.Items.Add("usagé");
      cbxEtatLibelleExemplaireRevue.Items.Add("détérioré");
      cbxEtatLibelleExemplaireRevue.Items.Add("inutilisable");
   }
   else if (etatExemplaireRevue == "usagé")
   {
      cbxEtatLibelleExemplaireRevue.Text = "";
      cbxEtatLibelleExemplaireRevue.Items.Add("neuf");
      cbxEtatLibelleExemplaireRevue.Items.Add("détérioré");
      cbxEtatLibelleExemplaireRevue.Items.Add("inutilisable");
   }
   else if (etatExemplaireRevue == "détérioré")
   {
      cbxEtatLibelleExemplaireRevue.Text = "";
      cbxEtatLibelleExemplaireRevue.Items.Add("neuf");
      cbxEtatLibelleExemplaireRevue.Items.Add("usagé");
      cbxEtatLibelleExemplaireRevue.Items.Add("inutilisable");
   }
   else if (etatExemplaireRevue == "inutilisable")
   {
      cbxEtatLibelleExemplaireRevue.Text = "";
      cbxEtatLibelleExemplaireRevue.Items.Add("neuf");
      cbxEtatLibelleExemplaireRevue.Items.Add("usagé");
      cbxEtatLibelleExemplaireRevue.Items.Add("détérioré");
   }
}
```

Dans la procédure événementielle "lblEtatExemplaireRevue_TextChanged", j'appelle une méthode pour mettre à jour les options de la combobox en fonction de l'état actuel de l'exemplaire. Lorsque l'utilisateur sélectionne un exemplaire dans la grille, les informations sur son état et les options de modification doivent s'afficher dans la groupbox des informations d'exemplaire. Pour cela, j'ai créé la méthode "dgvReceptionExemplairesListe_RowEnter", où j'affecte les valeurs des cellules de la ligne sélectionnée aux champs correspondants, et j'active la groupbox pour permettre la sélection et la modification de l'état.

MÉTHODE - DGVRECEPTIONEXEMPLAIRESLISTE_ROWENTER

```
private void dgvReceptionExemplairesListe_RowEnter(object sender,
DataGridViewCellEventArgs e)
{
   DataGridViewRow row = dgvReceptionExemplairesListe.Rows[e.RowIndex];
   string id = row.Cells["Id"].Value.ToString();
   string numero = row.Cells["Numero"].Value.ToString();
   DateTime dateAchat = (DateTime)row.Cells["dateAchat"].Value;
   string libelle = row.Cells["Libelle"].Value.ToString();
   gbxEtatExemplaireRevue.Enabled = true;
   lblNumeroExemplaireRevue.Text = numero;
   dtpDateAchatExemplaireRevue.Value = dateAchat;
   lblEtatExemplaireRevue.Text = libelle;
}
```

Pour permettre la modification de l'état d'un exemplaire, j'ai créé la procédure événementielle "btnEtatExemplaireRevueModifier_Click". Lorsque l'utilisateur clique sur le bouton "Modifier l'état de l'exemplaire", les valeurs des champs sont récupérées dans des variables. Ensuite, un nouvel objet Exemplaire est créé avec ces valeurs, puis la méthode du controller "ModifierEtatExemplaireDocument(exemplaire)" est appelée pour effectuer la modification en fonction des nouvelles valeurs.

PROCÉDURE ÉVÈNEMENTIELLE – BTNETATEXEMPLAIREREVUEMODIFIER_CLICK

```
private void btnEtatExemplaireRevueModifier_Click(object sender, EventArgs e)
{
   string idDocument = txbReceptionRevueNumero.Text;
   int numero = int.Parse(lblNumeroExemplaireRevue.Text);
   DateTime dateAchat = dtpDateAchatExemplaireRevue.Value;
   string photo = "";
   string idEtat = GetIdEtat(cbxEtatLibelleExemplaireRevue.Text);
   try
   {
      string libelle = cbxEtatLibelleExemplaireRevue.SelectedItem.ToString();
      Exemplaire exemplaire = new Exemplaire(numero, dateAchat, photo, idEtat,
      idDocument, libelle);
      if (MessageBox.Show("Voulez-vous modifier l'état de l'exemplaire " +
      exemplaire.Numero + " en " + libelle + " ?", "Confirmation",
MessageBoxButtons.YesNo) ==
      DialogResult.Yes)
      {
         controller.ModifierEtatExemplaireDocument(exemplaire);
         MessageBox.Show("L'état de l'exemplaire " + exemplaire.Numero + " a bien été
modifié.", "Information");
         AfficheReceptionExemplairesRevue();
      }
   }
   catch (NullReferenceException)
   {
      MessageBox.Show("Le nouvel état de l'exemplaire doit être sélectionné.",
"Information");
   }
}
```

Pour permettre la suppression d'un exemplaire de revue, j'ai mis en place la procédure évènementielle "btnExemplaireRevueSupprimer_Click". Cette fonctionnalité exige que l'utilisateur sélectionne une ligne dans la grille des exemplaires. Ensuite, je récupère l'objet "Exemplaire" correspondant à cette sélection et j'appelle la méthode "SupprimerExemplaireDocument(exemplaire)" du contrôleur pour effectuer la suppression dans la base de données. Enfin, je m'assure d'afficher la liste des exemplaires de la revue mise à jour après la suppression.

PROCÉDURE ÉVÈNEMENTIELLE - BTNETATEXEMPLAIREREVUEMODIFIER_CLICK

```
private void btnExemplaireRevueSupprimer_Click(object sender, EventArgs e)
{
   if (dgvReceptionExemplairesListe.SelectedRows.Count > 0)
   {
      Exemplaire exemplaire =
(Exemplaire)bdgExemplairesListe.List[bdgExemplairesListe.Position];
      if (MessageBox.Show("Voulez-vous supprimer l'exemplaire " + exemplaire.Numero
+ " de la revue " + exemplaire.Id + " ?", "Confirmation de suppression",
MessageBoxButtons.YesNo) == DialogResult.Yes)
      {
         controller.SupprimerExemplaireDocument(exemplaire);
         MessageBox.Show("L'exemplaire " + exemplaire.Numero + " a bien été supprimé.",
"Information");
         AfficheReceptionExemplairesRevue();
      }
   }
   else
   {
      MessageBox.Show("Une ligne doit être sélectionnée.", "Information");
   }
}
```

MISSION 4 : METTRE EN PLACE DES AUTHENTIFICATIONS

A. AJOUTER UNE TABLE “USER” -BDD

Pour gérer l'authentification des utilisateurs, je commence par créer une nouvelle table "utilisateur" dans la base de données, avec des champs pour les identifiants d'authentification.

```
CREATE TABLE `utilisateur` (
  `id` varchar(5) NOT NULL,
  `login` varchar(50) DEFAULT NULL,
  `password` varchar(50) DEFAULT NULL,
  `idService` varchar(5) DEFAULT NULL
) ENGINE=MyISAM DEFAULT CHARSET=utf8mb3 COLLATE=utf8mb3_unicode_ci;
```

En parallèle, je crée une table "service" pour répertorier les différents services de la médiathèque. Ensuite, j'insère des données fictives dans ces tables, notamment des comptes utilisateurs avec des identifiants et des mots de passe.

```
INSERT INTO service (id, libelle) VALUES
(1, 'administratif'),
(2, 'prêts'),
(3, 'culture'),
(0, 'administrateur');

INSERT INTO utilisateur (id, login, password, idService) VALUES ('00001',
'CharlotteOFraise', 'CharlotteOFraise230', '0'), ('00002', 'ZakariZotto', 'ZakariZotto430',
'1'), ('00003', 'BobLeponge', 'BobLeponge530', '2'), ('00004', 'RenaultMegane',
'RenaultMegane630', '3');
```

Enfin, je définis les clés primaires et étrangères correspondantes pour assurer l'intégrité des données.

B. FONCTIONNALITÉ D’AUTHENTIFICATION

Je vais dans l'application C# et inclut une nouvelle vue appelée "FrmAuthentification" dans le dossier "view", configurée comme une fenêtre ("Form"). Cette vue comprend un panneau contenant le titre "Authentification", suivi d'une étiquette "Utilisateur" avec une zone de texte en dessous pour l'identifiant de l'utilisateur, puis une autre étiquette "Mot de passe" avec une zone de texte en dessous pour le mot de passe de l'utilisateur. Enfin, j'ajoute un bouton "Connexion" pour permettre à l'utilisateur de se connecter à l'application.

Pour afficher la fenêtre d'authentification avant celle de l'application, je modifie la classe "Program.cs" en remplaçant "Application.Run" par "Application.Run(new FrmAuthentification());" dans la fonction "Main()". Cela assure que la première fenêtre affichée est celle d'authentification.

Ensuite, je crée un nouveau contrôleur "FrmAuthentificationController" pour récupérer les informations d'identification de l'utilisateur, ainsi que deux nouvelles classes, "Service" et "Utilisateur", pour obtenir les propriétés des tables correspondantes dans la base de données.

Pour permettre l'authentification, je retourne à l'API "rest_mediatekdocuments" et je crée la méthode "selectUtilisateur($id)" qui récupère la liste des utilisateurs avec leur service associé. Cette méthode prend en paramètre l'identifiant de l'utilisateur et effectue une requête "Select" pour obtenir le login, le mot de passe et l'identifiant du service de l'utilisateur, ainsi que le libellé du service en utilisant une jointure entre les tables "utilisateur" et "service".

Enfin, j'appelle cette méthode dans un nouveau cas de la méthode "selectOne($table, $id)" en lui assignant l'endpoint "utilisateur".

MÉTHODE - SELECTUTILISATEUR($ID)

```
public function selectUtilisateur($id)
{
    $param = array(
            "id" => $id

    );
    $req = "select u.login, u.password , u.idService, s.libelle  ";
    $req .= "from utilisateur u  join service s on s.id=u.idService ";
    $req .= "where u.login =:id  ";
    return $this->conn->queryAll($req, $param);
}
```

Dans la classe "Access" de l'application C#, j'implémente la méthode "GetUtilisateur(string login, string password)" pour récupérer l'utilisateur. Cette méthode envoie une requête HTTP GET vers la fonction "selectUtilisateur" de l'API REST, en utilisant l'endpoint "utilisateur/" suivi du login comme paramètre.

Si le login existe dans la base de données, la fonction retourne les informations de l'utilisateur correspondant. Sinon, elle renvoie une réponse vide.

MÉTHODE - GETUTILISATEUR(STRING LOGIN, STRING PASSWORD)

```
public bool GetUtilisateur(string login, string password)
{
    Utilisateur utilisateur = access.GetUtilisateur(login);
    if (utilisateur == null)
    {
        return false;
    }
    if (utilisateur.Password.Equals(password))
    {
        Service.Id = utilisateur.IdService;
        Service.Libelle = utilisateur.Libelle;
        return true;
    }
    return false;
}
```

Dans le "FrmAuthentificationController", je construis la méthode d'authentification "GetUtilisateur(string login, string password)". À l'intérieur de cette méthode, j'appelle la fonction "GetUtilisateur(login, password)" de la classe "Access" pour récupérer les informations de l'utilisateur.

Si les informations de l'utilisateur existent et qu'il a un mot de passe associé, je récupère l'identifiant du service auquel il est affecté ainsi que le libellé correspondant.

MÉTHODE - GETUTILISATEUR(STRING LOGIN, STRING PASSWORD)

```
public Service GetUtilisateur(string login, string password)
{
  Utilisateur utilisateur = access.GetUtilisateur(login, password);
  if (utilisateur != null && utilisateur.Password.Equals(password))
  {
    return new Service(utilisateur.IdService, utilisateur.Libelle);
  }
  return null;
}
```

La procédure événementielle "btnConnexion_Click" du formulaire "FrmAuthentification" est créée pour gérer le clic sur le bouton de connexion. Elle récupère les informations saisies par l'utilisateur dans les champs "utilisateur" et "mot de passe". Une condition vérifie si les champs ne sont pas vides. Si c'est le cas, elle appelle la méthode "GetUtilisateur(utilisateur, password)" du contrôleur, où les valeurs saisies sont vérifiées. Si les identifiants ne sont pas valides, un message d'erreur est affiché. En cas de succès, la fenêtre de l'application est affichée.

PROCÉDURE ÉVÈNEMENTIELLE - BTNCONNEXION_CLICK

```
private void btnConnexion_Click(object sender, EventArgs e)
{
    string login = txbLogin.Text;
    string password = txbPassword.Text;
    if (!string.IsNullOrEmpty(login) && !string.IsNullOrEmpty(password))
    {
        Service service = controller.GetUtilisateur(login, password);
        if (service == null)
        {
            MessageBox.Show("Erreur d'authentification", "Alerte");
            txbPassword.Text = "";
        }
        else if (service.Libelle == "culture")
        {
            MessageBox.Show("Vous n'avez pas les droits d'accès à l'application.", "Alerte");
            Application.Exit();
        }
        else
        {
            MessageBox.Show("Vous êtes connecté", "Information");
            FrmMediatek frmMediatek = new FrmMediatek(service);
            frmMediatek.ShowDialog();
        }
    }
}
```

Pour mettre en place les habilitations d'accès, j'ai ajouté des conditions dans le code du "FrmMediatek" pour contrôler l'affichage des fonctionnalités en fonction des différents services :

- Les employés du service administratif et l'administrateur ont un accès complet à toutes les fonctionnalités. Pour gérer les commandes, j'ai inclus une condition dans laquelle j'affiche une fenêtre d'alerte si le libellé du service est "administratif" ou "administrateur". Cette condition est placée dans la procédure événementielle "FrmAlerteFinAbonnement_Shown".
- Les employés du service de prêts peuvent uniquement consulter le catalogue (livres, DVD, revues). J'ai créé une condition pour masquer les onglets de commande lorsque le libellé du service est "prêts" et désactiver tous les composants liés à la gestion du catalogue. Pour cela, j'ai mis en place une méthode "AutorisationsAcces(Service service)" dans laquelle je vérifie si le service est associé au libellé "prêts", auquel cas les champs de gestion et les onglets de commande sont masqués ou supprimés.

MÉTHODE - AUTORISATIONSACCES

```
private void AutorisationsAcces(Service service)
{
   if (service.Libelle == "prêts")
   {
      tabOngletsApplication.TabPages.Remove(tabCommandesLivres);
      tabOngletsApplication.TabPages.Remove(tabCommandesDvd);
      tabOngletsApplication.TabPages.Remove(tabCommandesRevues);
      grpLivresInfos.Enabled = false;
      txbExemplaireLivresNumero.Enabled = false;
      dtpDateAchatExemplaireLivre.Enabled = false;
      cbxEtatLibelleExemplaireLivre.Enabled = false;
      btnEtatExemplaireLivreModifier.Enabled = false;
      btnExemplaireLivreSupprimer.Enabled = false;
      grpDvdInfos.Enabled = false;
      txbExemplaireDvdNumero.Enabled = false;
      dtpDateAchatExemplaireDvd.Enabled = false;
      cbxEtatLibelleExemplaireDvd.Enabled = false;
      btnEtatExemplaireDvdModifier.Enabled = false;
      btnExemplaireDvdSupprimer.Enabled = false;
      grpRevuesInfos.Enabled = false;
      txbReceptionExemplaireNumero.Enabled = false;
      dtpReceptionExemplaireDate.Enabled = false;
      txbReceptionExemplaireImage.Enabled = false;
      btnReceptionExemplaireImage.Enabled = false;
      btnReceptionExemplaireValider.Enabled = false;
      dtpDateAchatExemplaireRevue.Enabled = false;
      btnExemplaireRevueSupprimer.Enabled = false;
      cbxEtatLibelleExemplaireRevue.Enabled = false;
      btnEtatExemplaireRevueModifier.Enabled = false;
   }
}
```

Les employés du service "culture" n'ont accès à aucune fonctionnalité. Pour gérer cela, j'ajoute une condition "else if (Service.Libelle == "culture")" dans la méthode "btnConnexion_Click" du formulaire d'authentification (FrmAuthentification). À l'intérieur de cette condition, j'affiche un message informant l'utilisateur qu'il n'a pas accès à l'application, puis j'utilise "Application.Exit()" pour fermer l'application.

MISSION 5 : ASSURER LA SÉCURITÉ, LA QUALITÉ ET INTÉGRER LES LOGS

TACHE 1: CORRIGER DES PROBLÈMES DE SÉCURITÉ

Pour résoudre le premier problème, je commence par modifier le fichier App.config afin d'y ajouter le nom de la chaîne de connexion. Cela permettra à la connexion vers la base de données de s'établir sans difficulté. Je remplace également le couple "admin:adminpwd" qui était initialement intégré en dur dans la classe Access.

Ensuite, je retourne dans la classe "Access" où j'ajoute la propriété "connectionName" à laquelle j'assigne le nom de la connexion provenant du fichier App.config. En créant un getter nommé "GetConnectionStringByName(string name)", je peux récupérer la chaîne de connexion. Je m'assure également que l'objet "ConnectionStringSettings" n'est pas null, garantissant ainsi que la chaîne existe bel et bien dans le fichier de configuration.

Enfin, dans le constructeur de la classe "Access()", j'intègre la chaîne d'authentification obtenue via le getter dans la variable permettant l'authentification.

MÉTHODE - ACCESS()

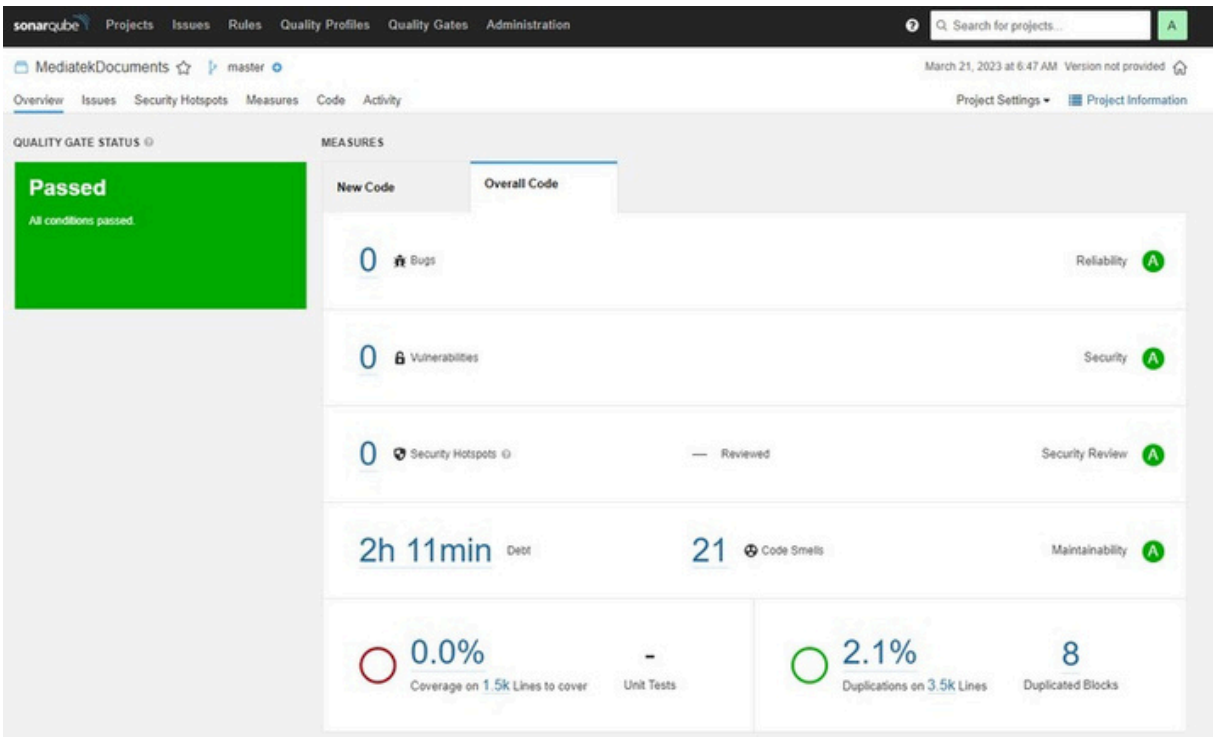
```
private Access()
{
  string authenticationString;
  try
  {
    authenticationString = GetConnectionStringByName(connectionName);
    api = ApiRest.GetInstance(uriApi, authenticationString);
  }
  catch (Exception e)
  {
    Console.WriteLine(e.Message);
    Environment.Exit(0);
  }
}
```

Pour résoudre le problème de l'API qui retourne la liste des fichiers du dossier lorsque l'adresse est accédée sans paramètres, je modifie le fichier htaccess pour prendre en compte une route vide et la gérer dans le fichier mediatekdocuments.php. Cela permettra d'avoir un retour d'erreur plutôt que la liste des fichiers du dossier.

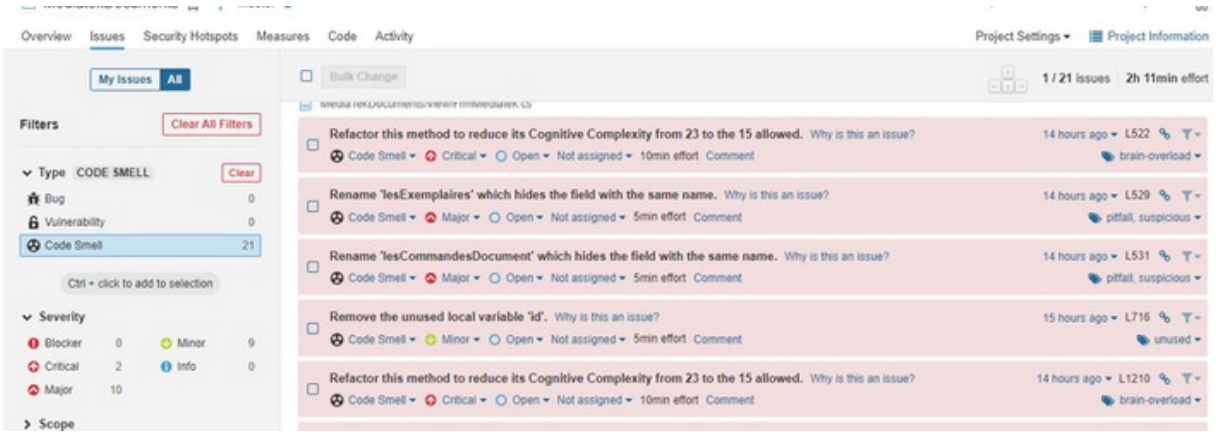
Pour résoudre le problème, j'ai ajouté une règle dans le fichier htaccess pour rediriger vers une page d'erreur lorsque la route est vide. Ensuite, j'ai modifié le fichier mediatekdocuments.php pour afficher un message d'erreur 404 lorsque cette situation se produit.

TACHE 2: CONTRPLER LA QUALITÉ

J'ai configuré SonarQube dans le projet "MediatekDocuments" en accédant à localhost:9000 après avoir lancé le serveur avec StartServer.bat. J'ai ajouté un nouveau projet et créé un token que j'ai intégré dans le fichier SonarQube.Analysis.xml. Ensuite, j'ai utilisé les commandes affichées sur localhost:9000 pour exécuter l'analyse SonarCube de mon application, détectant ainsi les erreurs et les problèmes de sécurité. Après leur exécution, le dashboard correspondant est apparu sur la page localhost:9000.



Le tableau de bord révèle principalement des problèmes de duplication de code, des variables inutilisées et des violations des conventions de nommage.

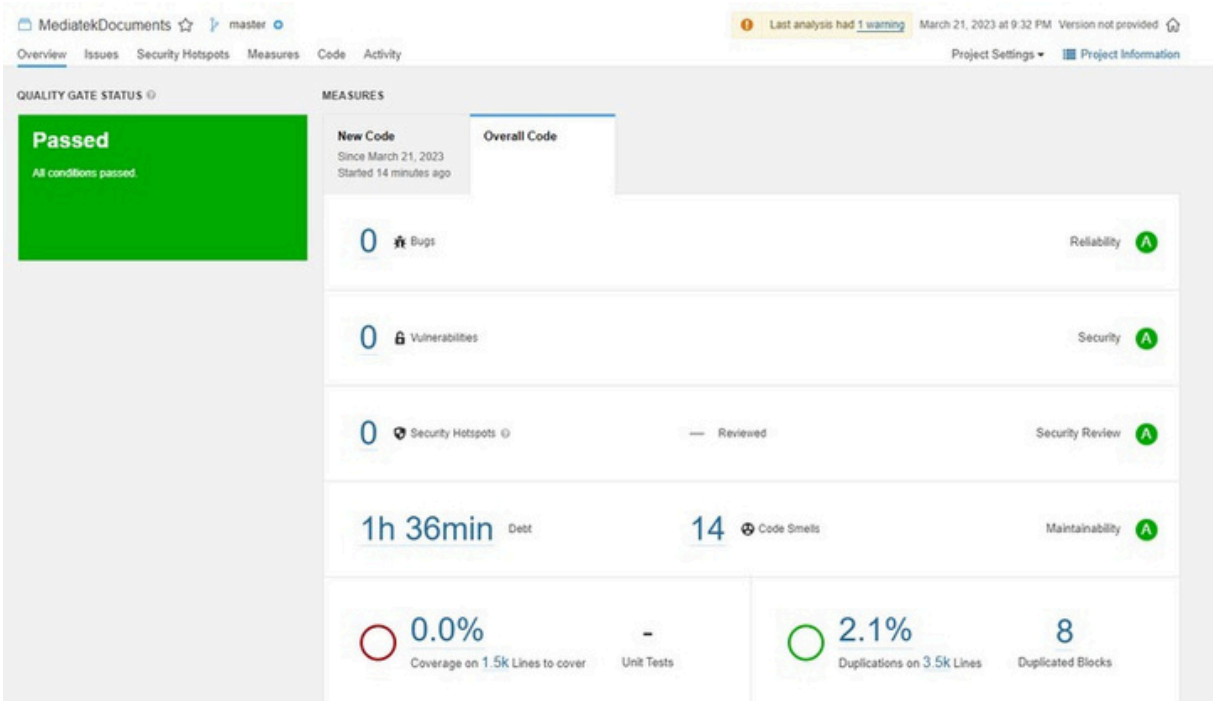


Ensuite, j'établis la connexion entre le projet C# et SonarCube en fournissant le token d'authentification et l'URL "http://localhost:9000" dans l'onglet "SonarLint". En procédant ensuite à une analyse de code en cliquant avec le bouton droit sur le projet puis en sélectionnant "exécuter l'analyse du code", je peux visualiser les messages et les avertissements relatifs aux règles de codage à suivre et aux erreurs détectées.

Après avoir effectué quelques configurations sur mon serveur Jenkins, notamment dans l'onglet "Administrer Jenkins" puis "Configurer le système", où j'ai ajouté un token d'authentification au serveur SonarQube, nommé "Admin Token", ainsi que dans "Configuration globale des outils" où j'ai ajouté une installation de "SonarScanner for MSBuild", j'ai pu passer à la création d'un nouvel élément "MediatekDocuments". J'ai précisé le dépôt correspondant à l'adresse GitHub du projet, puis ajouté les étapes de construction.

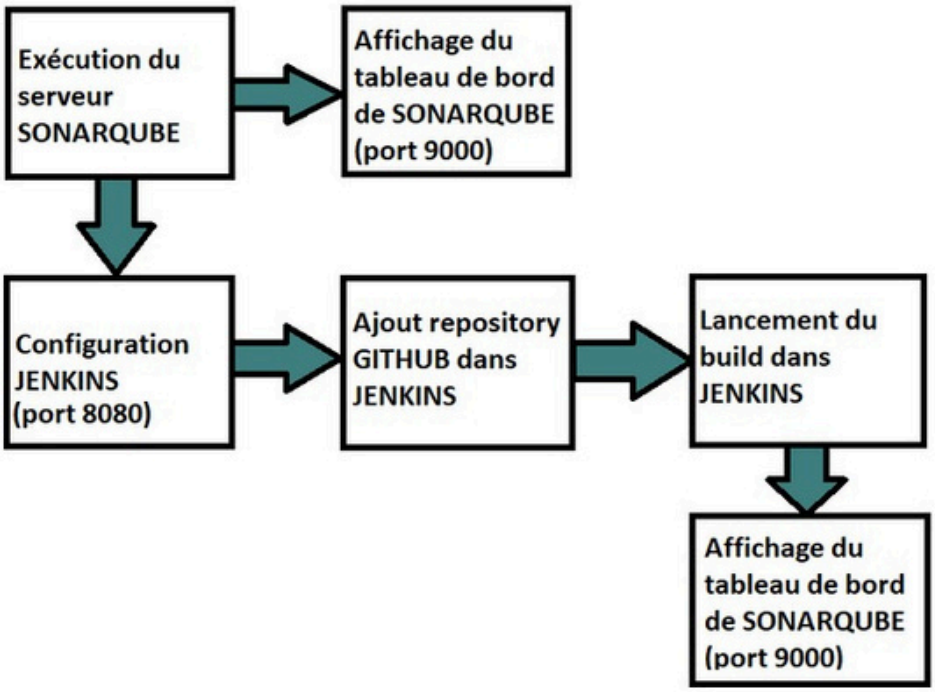
La première étape consistait à préparer une analyse SonarQube Scanner pour MSBuild, où j'ai attribué la clé du projet générée précédemment dans SonarQube et nommée "MediatekDocuments". Ensuite, j'ai créé une deuxième étape pour construire le projet Visual Studio avec MSBuild, en sélectionnant la version de MSBuild précédemment ajoutée. J'ai spécifié le chemin d'accès au fichier du projet "MediaTekDocuments.sln". Enfin, j'ai ajouté une étape pour terminer l'analyse SonarQube Scanner pour MSBuild, puis j'ai sauvegardé la configuration.

Pour lancer le build, j'ai cliqué sur "Lancer un build", et j'ai pu voir les erreurs et les avertissements concernant le code en consultant l'onglet "Sortie de la console" du build correspondant. Pour vérifier que la liaison avec le serveur SonarQube a été effectuée correctement, je me suis rendu sur l'URL http://localhost:9000/projects, où j'ai pu voir apparaître le projet avec les informations relatives aux erreurs et aux avertissements. En cliquant sur le projet, j'ai pu afficher les détails des erreurs dans l'onglet "Issues".



Après avoir ajouté la deuxième étape pour construire le projet avec MSBuild et terminé l'analyse SonarQube Scanner, j'ai sauvegardé la configuration dans Jenkins. Ensuite, j'ai lancé le build pour vérifier que les erreurs et les avertissements étaient correctement détectés, ce que j'ai pu confirmer en consultant la sortie de la console.

Pour m'assurer que la liaison avec SonarQube avait été établie avec succès, je suis allé sur le serveur SonarQube via l'URL http://localhost:9000/projects. Là, j'ai pu voir mon projet répertorié avec les informations relatives aux erreurs et aux avertissements. En cliquant sur le projet, j'ai pu explorer plus en détail les problèmes dans l'onglet "Issues". Cette vérification m'a permis de confirmer que l'analyse du code fonctionnait comme prévu et que les erreurs étaient correctement signalées.



Après avoir configuré SonarQube avec succès, je me suis attaqué à la correction des erreurs signalées, en commençant par celles concernant les "assignments" inutiles à des variables locales. Ces erreurs se sont révélées être le résultat de l'affectation de variables dans la classe "Access" qui n'étaient pas utilisées par la suite. J'ai donc supprimé ces variables non utilisées pour résoudre ces erreurs.

MÉTHODE "MODIFIERSUIVICOMMANDEDOCUMENT" (ACCESS) AVANT CORRECTION

```
public bool ModifierSuiviCommandeDocument(string id, int nbExemplaire, string
idLivreDvd, string idSuivi)
{
   string jsonModifierSuiviCommandeDocument = "{\"id\":\"" + id + "\", \"idSuivi\":\"" +
idSuivi + "\"}";
   Console.WriteLine("jsonModifierSuiviCommandeDocument" +
jsonModifierSuiviCommandeDocument);
}
```

MÉTHODE "MODIFIERSUIVICOMMANDEDOCUMENT" (ACCESS) APRÈS CORRECTION

```
public bool ModifierSuiviCommandeDocument(string id, string idSuivi)
{
   string jsonModifierSuiviCommandeDocument = "{\"id\":\"" + id + "\", \"idSuivi\":\"" +
idSuivi + "\"}";
   Console.WriteLine("jsonModifierSuiviCommandeDocument" +
jsonModifierSuiviCommandeDocument);
}
```

MÉTHODE "MODIFIERSUIVICOMMANDEDOCUMENT" (FRMMEDIATEKCONTROLLER) AVANT CORRECTION

```
internal bool ModifierSuiviCommandeDocument(string id, int nbExemplaire, string
idLivreDvd, string idSuivi)
{
 return access.ModifierSuiviCommandeDocument(id, nbExemplaire, idLivreDvd,
idSuivi);
}
```

APRÈS CORRECTION :

```
internal bool ModifierSuiviCommandeDocument(string id, string idSuivi)
{
   return access.ModifierSuiviCommandeDocument(id, idSuivi);
}
```

Pour corriger l'erreur "Refactor this method to reduce its Cognitive Complexity from 23 to the 15 allowed", je simplifie les blocs de code dans les procédures évènementielles permettant la suppression d'un document si aucun exemplaire ou aucune commande n'est associé à ce document. Je remplace les boucles "foreach" par des variables vérifiant si l'identifiant du document est présent dans la liste des exemplaires ou dans la liste des commandes.

PROCÉDURE ÉVÈNEMENTIELLE "BTNSUPPRIMERLIVRE_CLICK" AVANT CORRECTION :

```
List<Exemplaire> lesExemplaires = controller.GetExemplairesDocument(livre.Id);
bool aucunExemplaire = true;
List<CommandeDocument> lesCommandesDocument =
controller.GetCommandesDocument(livre.Id);
bool aucuneCommande = true;

foreach (Exemplaire exemplaire in lesExemplaires)
{
   if (exemplaire.Id == livre.Id)
   {
      aucunExemplaire = false;
      break;
   }
}

foreach (CommandeDocument commandeDocument in lesCommandesDocument)
{
   if (commandeDocument.IdLivreDvd == livre.Id)
   {
```

```
List<Exemplaire> lesExemplaires = controller.GetExemplairesDocument(livre.Id);
bool aucunExemplaire = true;
List<CommandeDocument> lesCommandesDocument =
controller.GetCommandesDocument(livre.Id);
bool aucuneCommande = true;

foreach (Exemplaire exemplaire in lesExemplaires)
{
    if (exemplaire.Id == livre.Id)
    {
        aucunExemplaire = false;
        break;
    }
}

foreach (CommandeDocument commandeDocument in lesCommandesDocument)
{
    if (commandeDocument.IdLivreDvd == livre.Id)
    {
        aucuneCommande = false;
        break;
    }
}

if (aucunExemplaire)
{
    if (aucuneCommande)
    {
        if (controller.SupprimerLivre(livre.Id))
        {
            // Code à exécuter en cas de suppression réussie du livre
        }
    }
}
```

PROCÉDURE ÉVÈNEMENTIELLE "BTNSUPPRIMERLIVRE_CLICK" APRÈS CORRECTION :

```
var exemplairesDvd = controller.GetExemplairesDocument(dvd.Id);
var aucunExemplaireDvd = !exemplairesDvd.Any();
var commandesDvd = controller.GetCommandesDocument(dvd.Id);
var aucuneCommandeDvd = !commandesDvd.Any();

if (aucunExemplaireDvd && aucuneCommandeDvd)
{
    if (controller.SupprimerDvd(dvd.Id))
    {
        lesDvd = controller.GetAllDvd();
        RemplirComboCategorie(controller.GetAllGenres(), bdgGenres, cbxDvdGenres);
        RemplirComboCategorie(controller.GetAllPublics(), bdgPublics, cbxDvdPublics);
        RemplirComboCategorie(controller.GetAllRayons(), bdgRayons, cbxDvdRayons);
        RemplirDvdListeComplete();
        MessageBox.Show("Le dvd " + dvd.Titre + " a correctement été supprimé.");
    }
}
```

TACHE 3: INTÉGRER DES LOGS

Après avoir ajouté les packages NuGet requis pour utiliser la classe "Serilog", je procède à la configuration des journaux (logs). Cette configuration est intégrée dans la méthode "Access()".

```
private Access()
{
   string authenticationString;
   try
   {
      authenticationString = GetConnectionStringByName(connectionName);
      Log.Logger = new LoggerConfiguration()
         .MinimumLevel.Information()
         .WriteTo.Console()
         .WriteTo.File(new JsonFormatter(), "logs/log.txt", rollingInterval:
RollingInterval.Day)
         .CreateLogger();
      api = ApiRest.GetInstance(uriApi, authenticationString);
   }
   catch (Exception e)
   {
      Log.Fatal("Access.Access catch connectionString={0} erreur={1}", connectionName,
e.Message);
      Environment.Exit(0);
   }
}
```

Dans la méthode `Access()`, je configure les logs en utilisant la classe `Serilog`. J'essaie d'abord de récupérer la chaîne de connexion à partir du fichier de configuration. Ensuite, je définis le niveau minimal des logs à "Information" et je configure deux destinations pour les logs : la console et un fichier journal qui sera sauvegardé dans le dossier "logs" avec une rotation quotidienne. En cas d'erreur lors de la configuration ou de l'initialisation de l'API, j'utilise `Serilog` pour enregistrer un message d'erreur critique dans les logs et je quitte l'application.

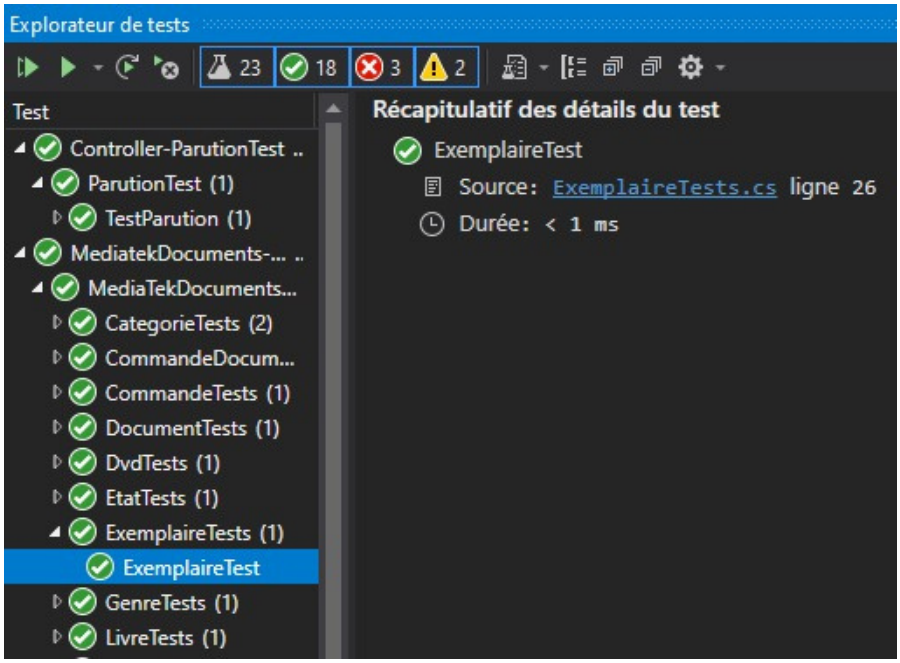
```
public bool SupprimerDocument(string Id)
{
   string jsonIdDocument = "{ \"id\" : \"" + Id + "\"}";
   Console.WriteLine("jsonIdDocument" + jsonIdDocument);
   try
   {
      List<Document> liste = TraitementRecup<Document>(DELETE, "document/" +
jsonIdDocument);
      return (liste != null);
   }
   catch (Exception ex)
   {
      Console.WriteLine(ex.Message);
      Log.Error("Access.SupprimerDocument catch jsonIdDocument={0} erreur={1}",
jsonIdDocument, ex.Message);
   }
}
```

MISSION 6 : TESTER ET DOCUMENTER

TACHE 1: GÉRER LES TESTS

Pour créer des tests unitaires dans un projet C# sous Visual Studio, je commence par ajouter un nouveau projet de type "Test" à la solution en faisant un clic droit sur la solution puis en sélectionnant "Ajouter un nouveau projet". Je choisis ensuite le type de projet "Test" et je lui donne un nom approprié, comme "MediatekDocuments-ModelTests". Une fois le projet créé, je crée des classes de test pour chaque classe du dossier "Model" que je souhaite tester. Par exemple, pour tester la classe "Exemplaire", je crée une classe de test nommée "ExemplaireTests".

```
public class AbonnementTests
{
    private const string id = "00050";
    private static readonly DateTime dateCommande = new DateTime(2023, 3, 22);
    private const double montant = 50;
    private static readonly DateTime dateFinAbonnement = new DateTime(2023, 3, 30);
    private const string idRevue = "10003";
    private const string titre = "Challenges";

    private static readonly Abonnement abonnement = new Abonnement(id, dateCommande, montant, dateFinAbonnement, idRevue, titre);

    [TestMethod()]
    0 références
    public void AbonnementTest()
    {
        Assert.AreEqual(id, abonnement.Id, "devrait réussir : id valorisé");
        Assert.AreEqual(dateCommande, abonnement.DateCommande, "devrait réussir : date de commande valorisée");
        Assert.AreEqual(montant, abonnement.Montant, "devrait réussir : montant valorisé");
        Assert.AreEqual(dateFinAbonnement, abonnement.DateFinAbonnement, "devrait réussir : date de fin d'abonnement valorisée");
        Assert.AreEqual(idRevue, abonnement.IdRevue, "devrait réussir : idRevue valorisé");
        Assert.AreEqual(titre, abonnement.Titre, "devrait réussir : titre valorisé");
    }
}
```

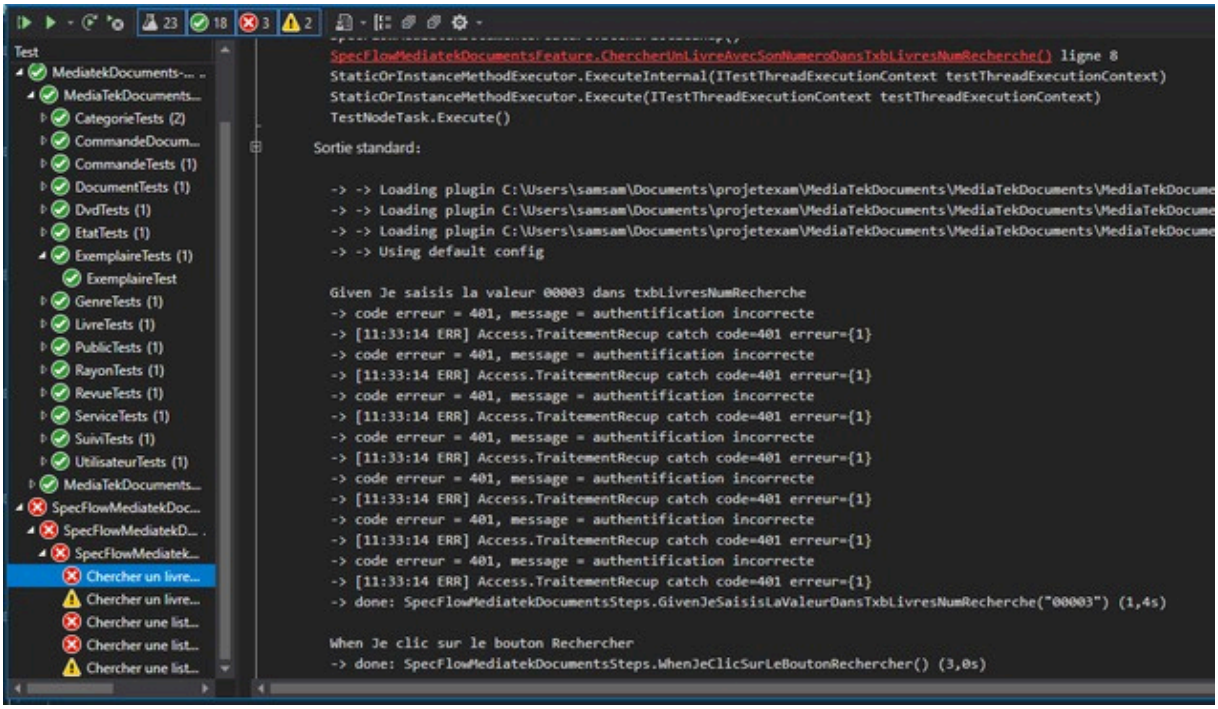


Le paragraphe décrit le processus de création des tests fonctionnels dans un projet C# sous Visual Studio. Il commence par l'ajout d'un nouveau projet de type "Specflow" dans la solution, auquel est attribué le nom "SpecFlowMediatekDocuments". Ensuite, les scénarios du fichier "SpecFlowMediatekDocuments.feature" dans le dossier "Features" sont modifiés pour générer les méthodes d'exécution des tests de recherche d'un livre.

```
Feature: SpecFlowMediatekDocuments
Recherche d'un livre
@rechercheLivreNumero
Scenario: Chercher un livre avec son numero dans txbLivresNumRecherche
   Given Je saisis la valeur 00003 dans txbLivresNumRecherche
   When Je clic sur le bouton Rechercher
   Then Les informations détaillées affichent le titre 'Et je danse'

Scenario: Chercher une liste de livres avec le genre dans cbxLivresGenres
   Given Je sélectionne la valeur 'Policier' dans cbxLivresGenres
   Then Le résultat est 8 livres dans dgvLivresListe

Scenario: Chercher une liste de livres avec le public dans cbxLivresPublics
   Given Je sélectionne la valeur 'Adultes' dans cbxLivresPublics
   Then Le résultat est 12 livres dans dgvLivresListe

Scenario: Chercher une liste de livres avec le rayon dans cbxLivresRayons
   Given Je sélectionne la valeur 'Sciences' dans cbxLivresRayons
   Then Le résultat est 0 livre

Scenario: Chercher un livre avec son titre dans txbLivresTitreRecherche
   Given Je saisis la valeur 'Et je danse' dans txbLivresTitreRecherche
   When Je clic sur le bouton Rechercher
   Then Les informations détaillées affichent le titre 'Et je danse'
```

J'ai ensuite effectué un clic droit puis "generate steps definitions", ce qui a créé la classe de tests "SpecFlowMediatekDocumentsSteps.cs" , dans le dossier "Steps" du projet Specflow. Puis j'ai modifié les méthodes de tests.

```
[Given(@"Je sélectionne la valeur '(.*)' dans cbxLivresGenres")]
public void GivenJeSelectionneLaValeurDansCbxLivresGenres(string listeAttendue)
{
   ComboBox cbxLivresGenres =
(ComboBox)frmMediatek.Controls["tabOngletsApplication"].Controls["tabLivres"].Cont
rols["gr
pLivresInfos"].Controls["cbxLivresGenres"];
   cbxLivresGenres.SelectedItem = listeAttendue;
}
```

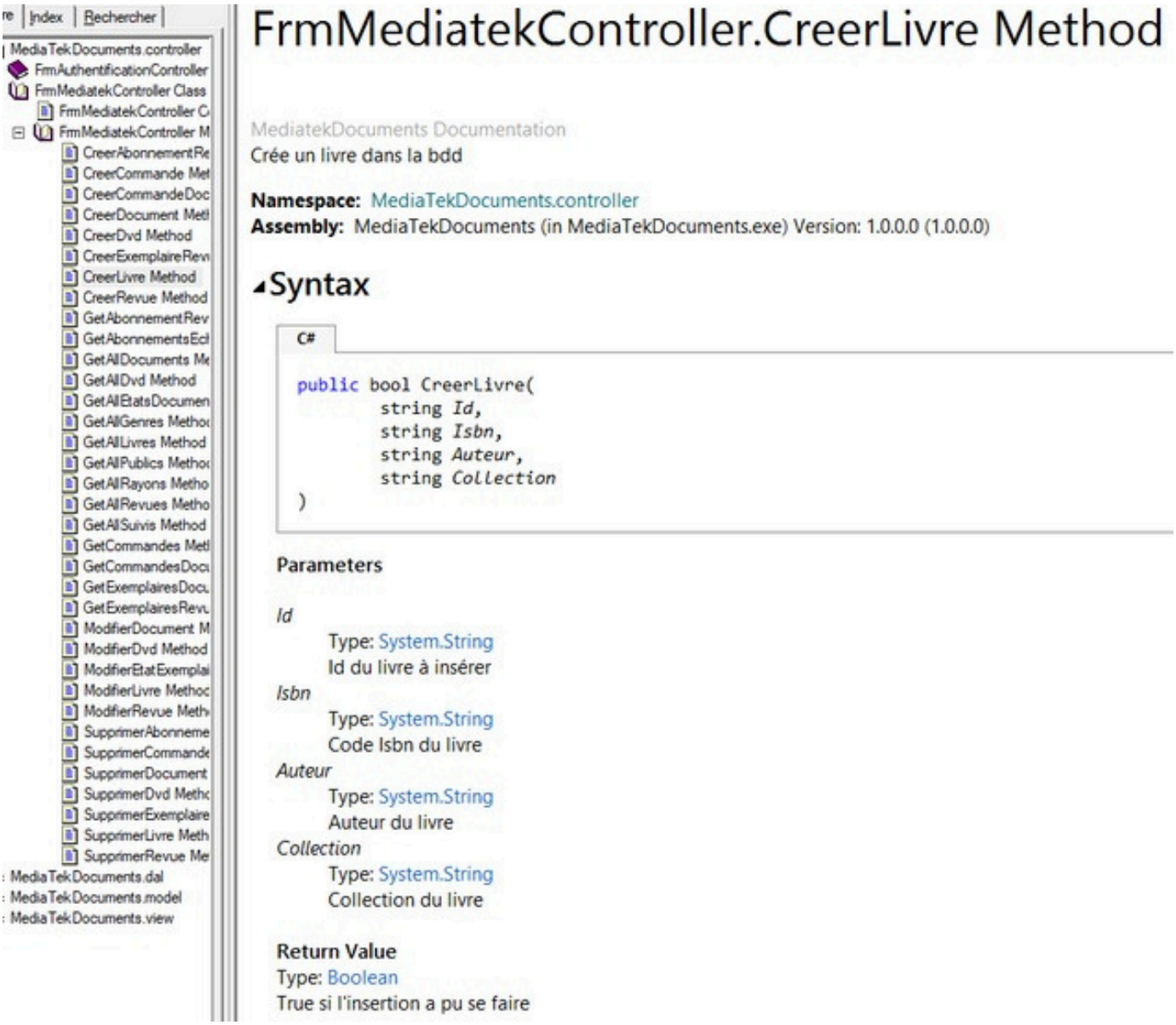


J'ai procédé à la réalisation des tests sur Postman en ouvrant simplement le logiciel. J'ai créé une collection de tests que j'ai nommée "MediatekDocumentsTests". Dans cette collection, j'ai ajouté les requêtes GET correspondant à la classe "AccessBDD" de l'API REST "rest_mediatekdocuments". Ensuite, j'ai effectué les tests en utilisant la commande suivante dans l'onglet "Tests" de chaque requête :

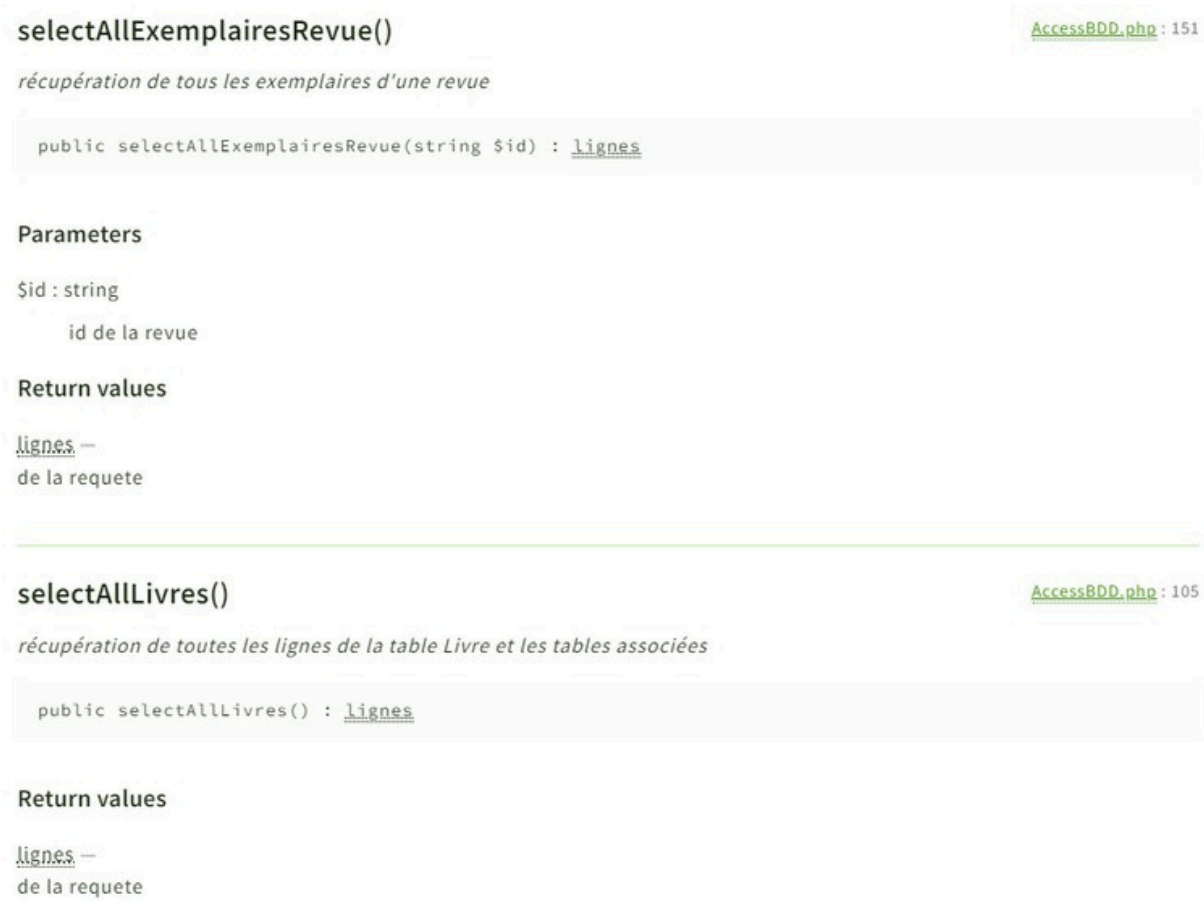
TACHE 2: DOCUMENTATIONS TECHNIQUES

Il était nécessaire de créer une documentation au format HTML Help 1 (chm), pour laquelle j'ai utilisé l'outil SandCastle. Après avoir repris tous les commentaires normalisés de l'application C#, j'ai généré le build sous cette forme. Un exemple de commentaire normalisé dans la classe "FrmMediatekController" est le suivant :

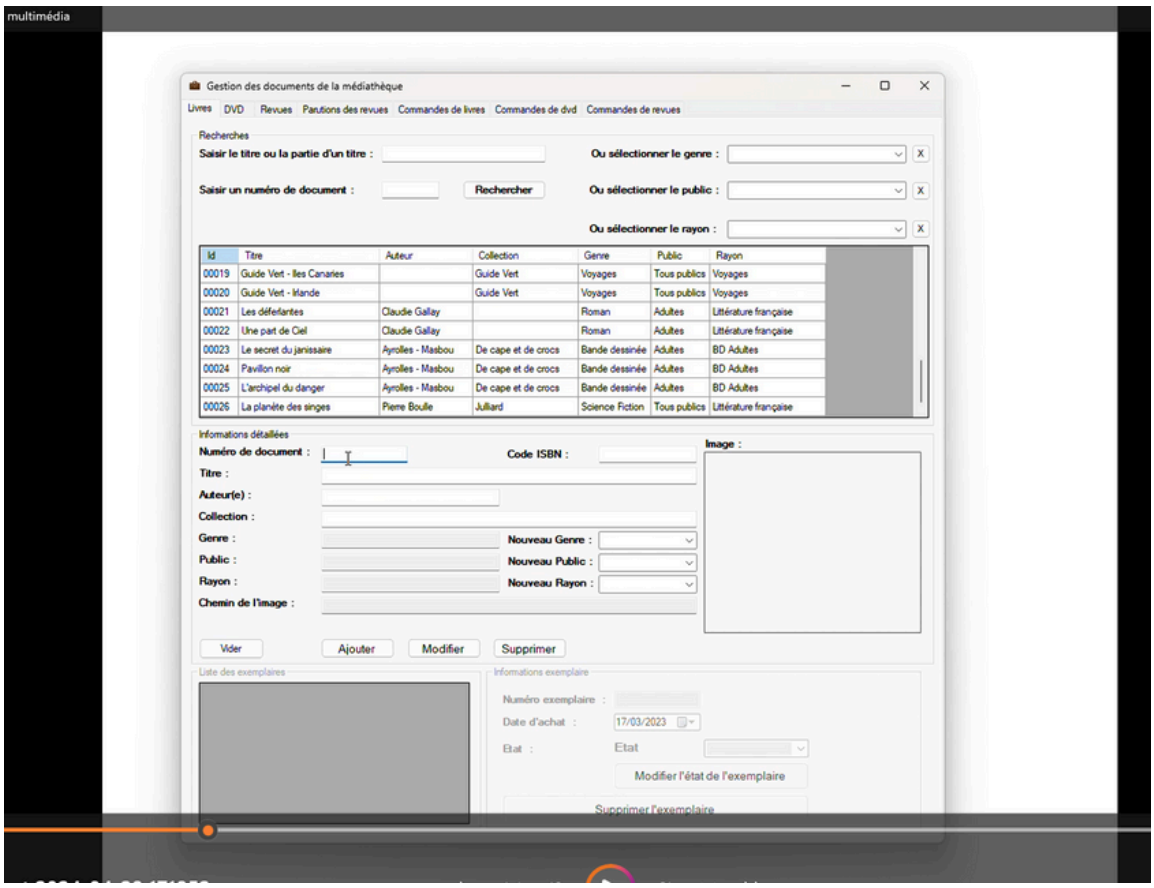
```
/// <summary>
/// Crée un livre dans la base de données.
/// </summary>
/// <param name="Id">Id du livre à insérer.</param>
/// <param name="Isbn">Code ISBN du livre.</param>
/// <param name="Auteur">Auteur du livre.</param>
/// <param name="Collection">Collection du livre.</param>
/// <returns>True si l'insertion a réussi.</returns>
public bool CreerLivre(string Id, string Isbn, string Auteur, string Collection)
{
   return access.CreerLivre(Id, Isbn, Auteur, Collection);
}
```



La documentation technique de l'API "rest_mediatekdocuments" est produite dans NetBeans à l'aide de "phpDocumentor". Elle est stockée dans le dossier "Documentation Technique". Pour générer cette documentation, j'ai lancé l'invite de commandes en mode administrateur et exécuté la commande suivante :



TACHE 3: DOCUMENTATION UTILISATEUR



MISSION 7 : DÉPLOYER ET GÉRER LES SAUVEGARDES DE DONNÉES

TACHE 1: DÉPLOYER LE PROJET

J'ai déployé la base de données en récupérant le script de la base locale depuis phpMyAdmin de mon serveur WampServer. Ensuite, dans le tableau de bord de mon hébergeur, j'ai créé une nouvelle base de données et un nouvel utilisateur avec tous les droits. J'ai importé le script SQL dans cette nouvelle base de données via phpMyAdmin.

Ensuite, j'ai mis à jour les informations de connexion dans la classe "AccessBDD" de mon API REST, remplaçant les données locales par celles de la base de données en ligne.

Quant à l'API REST elle-même, je l'ai importée sur mon hébergeur en créant un nouveau nom de domaine. J'ai ensuite ajouté le dossier de l'API dans le répertoire "public_html" de mon nouveau domaine.

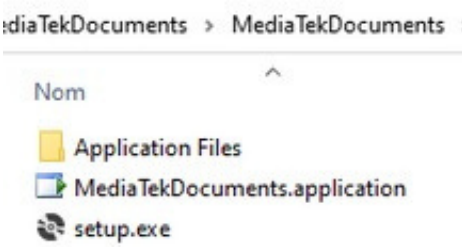
Pour finaliser la configuration, j'ai modifié l'adresse de l'URI de l'API dans la classe "Access" de mon application C#, lui affectant le domaine que j'avais créé pour héberger l'API.

Maintenant que les deux applications sont correctement configurées, je procède à la création de l'installateur dans l'application C# sur Visual Studio 2019.

Pour ce faire, je démarre par un clic droit sur le projet "MediatekDocuments", puis je me rends dans "Propriétés" et "Publish". Je spécifie le chemin vers l'emplacement où je souhaite que l'installateur soit généré, par exemple "Installation MediatekDocuments", et je clique sur le bouton "Publier maintenant".

Les fichiers d'installation sont ainsi créés dans le dossier spécifié. Pour installer l'application, je lance le fichier "setup.exe". Lors de l'installation, un raccourci vers l'application est automatiquement créé sur le bureau de l'utilisateur.

Pour lancer l'application une fois installée, l'utilisateur peut soit cliquer directement sur le fichier "mediatekdocuments.application", soit utiliser le raccourci de bureau correspondant, avec l'icône associée.



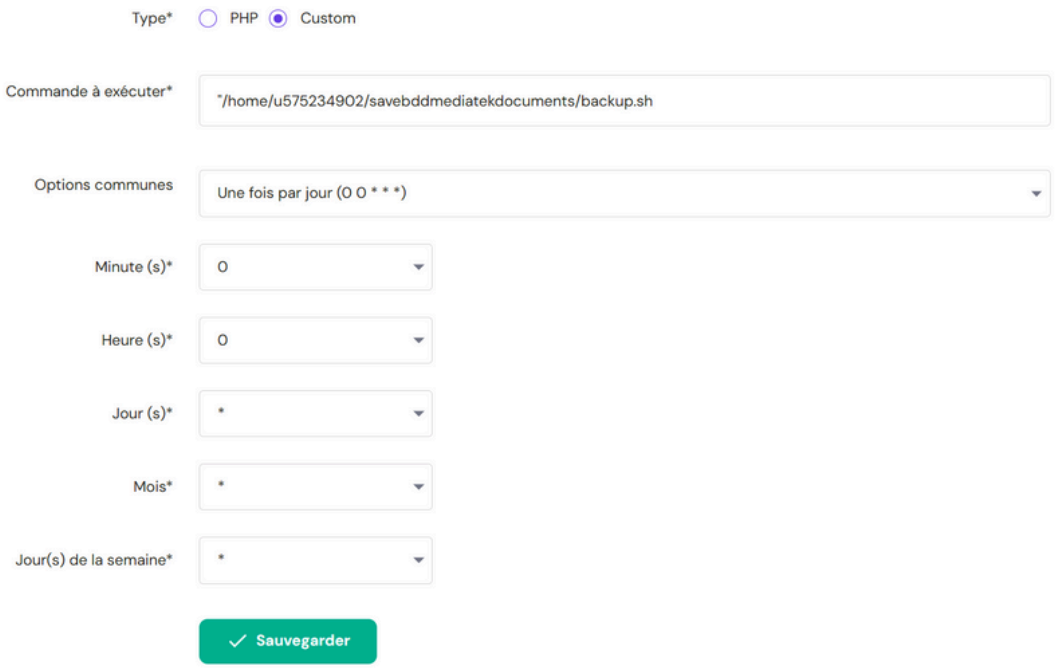
TACHE 2: GÉRER LES SAUVEGARDES DES DONNÉES

Pour instaurer des sauvegardes quotidiennes, je commence par créer un nouveau répertoire nommé "savebddmediatekdocuments" dans la structure de fichiers de mon hébergement sur Hostinger, situé à la racine du dossier "/home". Dans ce répertoire, je génère un fichier nommé "backup.sh" et y configure les informations de connexion à la base de données. Ensuite, je me rends dans le répertoire "doc2unix" de mon ordinateur pour convertir le fichier "backup.sh" récupéré depuis Hostinger.

Après avoir déplacé ce fichier vers le dossier “C:\Users\louiz\Downloads\dos2unix-7.5.2-win64\bin>” , j'exécute la commande suivante dans l'invite de commandes : "C:\Users\louiz\Downloads\dos2unix-7.5.2-win64\bin>dos2unix.exe backup.sh".

```
C:\Users\louiz\Downloads\dos2unix-7.5.2-win64\bin>dos2unix.exe backup.sh
dos2unix: converting file backup.sh to Unix format...

C:\Users\louiz\Downloads\dos2unix-7.5.2-win64\bin>
```

Je réintègre le fichier dans le dossier "savebddmediatekdocuments" sur Hostinger, puis je vais dans l'onglet "tâches cron". Pour créer une nouvelle tâche Cron, je sélectionne le mode "Custom" et j'ajoute la commande "/home/u575234902/savebddmediatekdocuments/backup.sh" dans le champ "commande à exécuter". Dans les options communes, je choisis "Une fois par jour", puis je clique sur "Sauvegarder". La tâche cron est maintenant visible, et le script de sauvegarde est opérationnel dans le répertoire "savebddmediatekdocuments/".



Il existe cependant un problème avec l'hébergeur Hostinger lors de la récupération du script de sauvegarde. En effet, si on veut récupérer le script, il faut posséder des droits "Super", auxquels on a hélas pas accès sans s'abonner à un autre plan d'hébergement (beaucoup plus cher).

En d'autres cas, si l'on souhaite récupérer le script de sauvegarde de la base de données, il faut se rendre dans le dossier "savebddmediatekdocuments" en racine du dossier "/home", puis télécharger le dossier

BILAN

Dans ce projet, j'ai réussi à atteindre la plupart des objectifs fixés, notamment l'ajout des fonctionnalités demandées, les tests unitaires et la liaison entre SonarQube et GitHub via Jenkins. Cependant, j'ai rencontré des difficultés avec les requêtes "Access" dans l'API REST, ainsi qu'avec les tests fonctionnels utilisant SpecFlow, en raison de problèmes de version et de configuration. Malgré ces obstacles, j'ai réalisé la majorité du projet seule, en excluant certaines missions facultatives. Malgré des frustrations occasionnelles, j'ai trouvé du plaisir dans la réalisation de ces tâches, ce qui a contribué à compenser les défis rencontrés. Dans l'ensemble, je suis satisfaite du résultat final.