Mediatek Documents

Atelier de professionnalisation

Margaux Lachèze
BTS SIO OPTION SLAM

INTRODUCTION 3

CONTEXTE 3
EXISTANT 3
LANGAGES ET TECHNOLOGIES UTILISEES 3
PRECISIONS 3

MISSION : GERER LES COMMANDES 4

TACHE 1 : GERER LES COMMANDES DE LIVRES OU DE DVD 4
KANBAN : 5
DIAGRAMME DE CAS D'UTILISATION : 5
MAQUETTES - FIGMA : 6
EXPLICATIONS SUR LE CODE 8
TACHE 2 : GERER LES COMMANDES DE REVUES 15
KANBAN 16
DIAGRAMME DE CAS D'UTILISATION 16
MAQUETTES – FIGMA 17
EXPLICATIONS SUR LE CODE 18

MISSION : METTRE EN PLACE DES AUTHENTIFICATIONS 25

KANBAN 25
DIAGRAMME DE CAS D'UTILISATION 26
MCD 26
MAQUETTES – FIGMA 26
EXPLICATIONS SUR LE CODE 27

MISSION : ASSURER LA SECURITE, LA QUALITE ET INTEGRER DES LOGS 29

TACHE 1 : CORRIGER DES PROBLEMES DE SECURITE 29
KANBAN 29
ISSUES 29
PULL REQUESTS 30
EXPLICATIONS SUR LE CODE 30
TACHE 2 : CONTROLER LA QUALITE 31
KANBAN 31
EXTRAIT DE LA LISTE DES AVERTISSEMENTS SONARLINT 31
TACHE 3 : INTEGRER DES LOGS 32
KANBAN 32
EXPLICATIONS SUR LE CODE 32

MISSION : TESTER ET DOCUMENTER 33

TACHE 1 : GERER LES TESTS 33

KANBAN 33
EXEMPLE DE CODE DE TEST UNITAIRE C# 33
EXEMPLE DE TEST FONCTIONNEL SOUS POSTMAN 34
TACHE 2 : CREER LES DOCUMENTATIONS TECHNIQUES 34
KANBAN 35
EXEMPLE DE COMMENTAIRE NORMALISE 35
EXTRAIT DE LA DOCUMENTATION TECHNIQUE 35
TACHE 3 : CREER LA DOCUMENTATION UTILISATEUR EN VIDEO 36
KANBAN 37
INFORMATIONS 37

MISSION : DEPLOYER ET GERER LA SAUVEGARDE DES DONNEES 38

TACHE 1 : DEPLOYER LE PROJET 38
KANBAN 38
EXPLICATIONS SUR LES ETAPES DU DEPLOIEMENT 38
TACHE 2 : GERER LES SAUVEGARDES DE DONNEES 39
KANBAN 39
CAPTURE D'ECRAN DE L'ENREGISTREMENT DE LA SAUVEGARDE 39
EXPLICATIONS 39
MODE OPERATOIRE POUR LA RESTAURATION 39

BILAN FINAL 40

OBJECTIFS ATTEINTS 40
DIFFICULTES RENCONTREES 40

Introduction

Contexte

« Vous travaillez en tant que technicien développeur junior pour l'ESN InfoTech Services 86 qui vient de remporter le marché pour différentes interventions au sein du réseau MediaTek86, dont certaines dans le domaine du développement d'application. Un contrat de développement a ainsi été rédigé entre MediaTek86 et InfoTech afin de réaliser 3 applications. »

Dans cet atelier nous travaillons sur l'application Mediatek Document. Cette dernière doit permettre aux personnels des services administratifs et prêts de gérer les documents (livres, DVD, revues) de chaque médiathèque du réseau "Mediatek86".

Existant

Il n'y a pas d'application existante. La partie catalogue (onglets livre, dvd, revue et parution) a déjà été codée par un autre développeur de l'équipe ITS86 (appli C# et API rest)

Langages et technologies utilisées

Langages :

- C# pour l'application de bureau
- PHP pour l'API

SGBD :

- MySQL

Authentification de l'API :

- basic authentication

Précisions

Dans le cadre de cet atelier, nous pouvions travailler seul ou en groupe. Si nous travaillions seul, ce qui est mon cas, les mission 1 et 3 n'étaient pas à rendre. Ces missions correspondent à :

- 1 : Gérer les documents
- 3 : Gérer le suivi de l'état des exemplaires

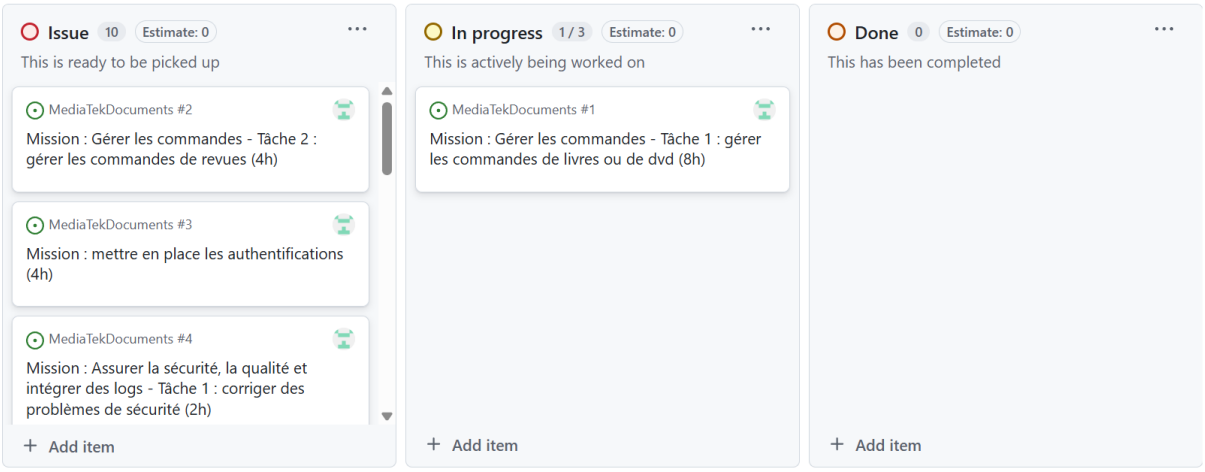
Dans le cadre de ce bilan, elles ne seront donc pas abordées.

Mission : Gérer les commandes

Tâche 1 : gérer les commandes de livres ou de dvd

- Dans la base de données, créer la table 'Suivi' qui contient les différentes étapes de suivi d'une commande de document de type livre ou dvd. Relier cette table à CommandeDocument.
- Créer un onglet (ou une nouvelle fenêtre) pour gérer les commandes de livres.
- La charte graphique doit correspondre à l'existant.
- Dans toutes les listes, permettre le tri sur les colonnes.
- Dans l'onglet (ou la fenêtre), permettre la sélection d'un livre par son numéro, afficher les informations du livre ainsi que la liste des commandes, triée par date (ordre inverse de la chronologie). La liste doit comporter les informations suivantes : date de la commande, montant, nombre d'exemplaires commandés et l'étape de suivi de la commande.
- Créer un groupbox qui permet de saisir les informations d'une nouvelle commande et de l'enregistrer. Lors de l'enregistrement de la commande, l'étape de suivi doit être mise à "en cours".
- Permettre de modifier l'étape de suivi d'une commande en respectant certaines règles (une commande livrée ou réglée ne peut pas revenir à une étape précédente (en cours ou relancée), une commande ne peut pas être réglée si elle n'est pas livrée).
- Permettre de supprimer une commande uniquement si elle n'est pas encore livrée. Faire un trigger qui réalise aussi la suppression dans la classe fille.
- Créer le trigger qui se déclenche si une commande passe à l'étape "livrée" et qui crée autant de tuples dans la table "Exemplaire" que nécessaires, en valorisant la date d'achat avec la date de commande et en mettant l'état de l'exemplaire à "neuf". Le numéro d'exemplaire doit être séquentiel par rapport au livre concerné.
- Créer un onglet pour gérer les commandes de DVD en suivant la même logique que pour les commandes de livres.
- Toutes les sécurités seront mises en place pour éviter des erreurs de manipulation.

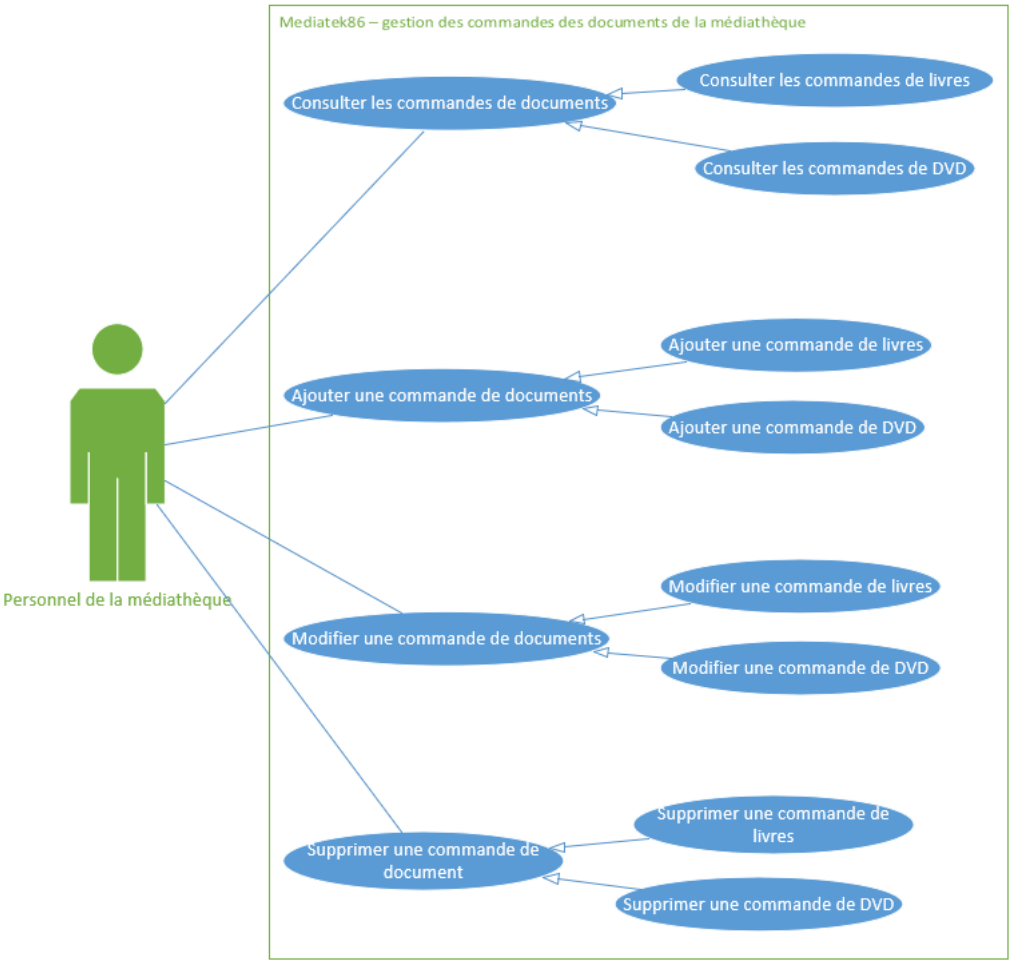
Kanban :



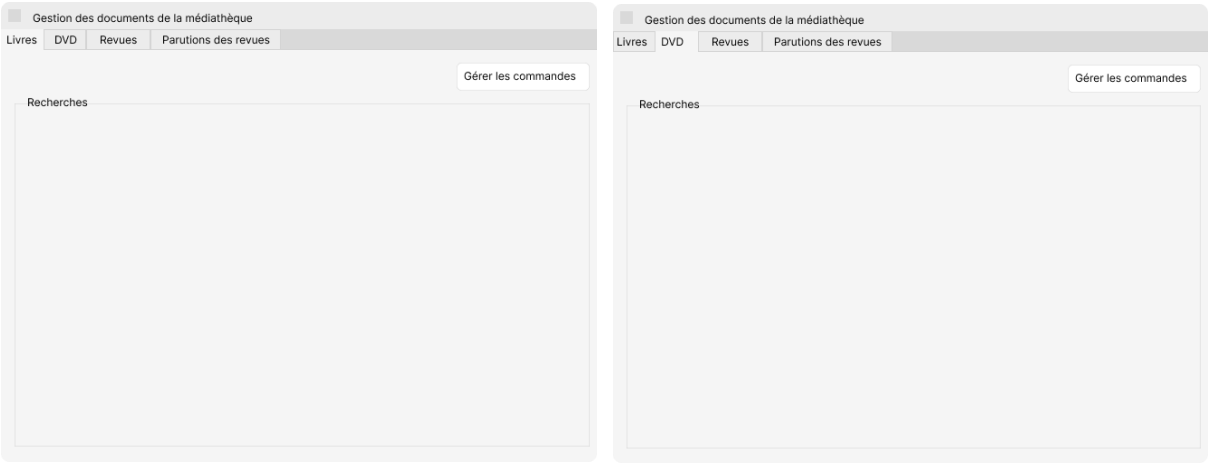
Temps estimé : 8h

Temps réel : 12h

Diagramme de cas d'utilisation :



Maquettes - Figma :



(nda : ces pages de l'applications étaient fournies, les maquettes montrent l'ajout d'un bouton pour gérer les commandes)

Maquette de l'onglet de gestion des commandes de livres

Maquette de l'onglet de gestion des commandes de DVD

Explications des maquettes :

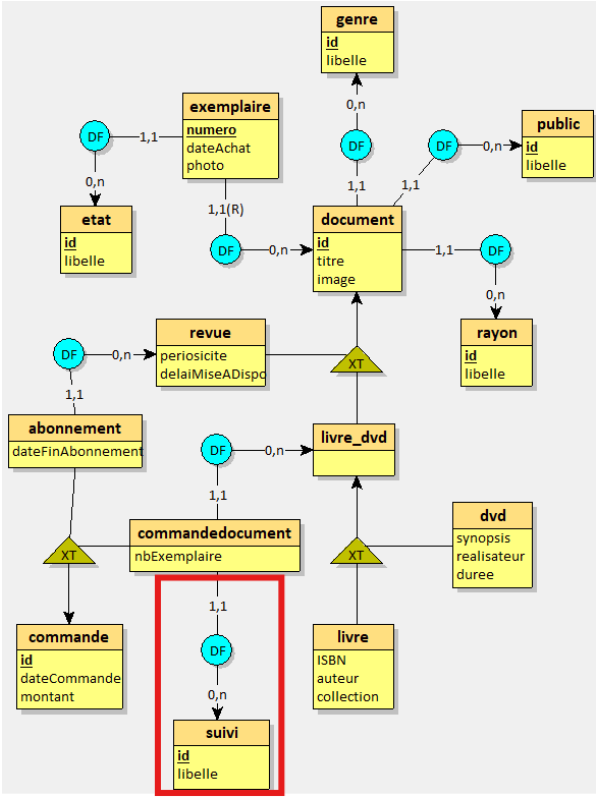
Dans le formulaire déjà conçu, j'ai ajouté des boutons « gérer les commandes » dans les onglets livres et dvd. Cela ouvre un nouveau formulaire, sur l'onglet correspondant à celui sur lequel le bouton a été cliqué.

On retrouve ensuite toutes les informations du document, dont le numéro qui doit être saisi par l'utilisateur pour initier la recherche. Ensuite, s'affichent les commandes inhérentes au livre ou au dvd. L'utilisateur peur modifier une commande, la supprimer si elle n'est pas livrée ou en ajouter une.

L'édition d'une commande se fait en bas de la fenêtre du formulaire où l'utilisateur peut remplir les informations sur la date, le nombre d'exemplaire qu'il veut commander, le montant et l'étape de la commande. A l'ajout, une commande est toujours sur l'étape en cours. L'utilisateur peut ensuite valider ou annuler ses modifications.

Modèle Conceptuel des Données :

La table ajoutée est encadrée en rouge :



Explications sur le code

Gestion du suivi des commandes (base de données) :

Pour assurer le suivi des commandes, j'ai créé la table suivi qui a comme clé primaire id, et comme attribut libelle. Dans la table commandedocument, idSuivi fait référence à id de la table suivi.

Gestion de la commande de documents dans l'application C#:

Du côté de l'application C#, j'ai d'abord créé un nouveau formulaire FrmCommandes.cs ainsi que son contrôleur FrmCommandesController qui est reliée à Access.cs (classe pré-existante à laquelle j'ai ajouté des méthodes). J'ai fait la classe Suivi.cs pour la gestion des stades d'une commande puis la classe Commande.cs, mère de CommandeDocument.cs. Le formulaire FrmCommandes.cs a deux onglets : livres et dvd.

En ce qui concerne l'API, les méthodes ont toutes été ajoutées dans la classe MyAccessBDD.

Suivi.cs :

- public Suivi(string id, string libelle)

Chaque étape de suivi a un identifiant (« 00001 », « 00002 », « 00003 », « 00004 ») et un libelle (« En cours », « Livrée », « Réglée », « Relancée »).

Commande.cs :

- public Commande(string id, DateTime dateCommande, double montant)

Classe mère. Une commande a un id, une date et un montant.

CommandeDocument.cs :

- public CommandeDocument(string id, DateTime dateCommande, double montant, int nbExemplaire, string idLivreDvd, string idSuivi, string suivi) : base(id, dateCommande, montant)

Hérite de Commande.cs. Une commande de document a un nombre d'exemplaire et est reliée à un objet de type LivreDvd (idLivreDvd). Elle est aussi liée à un objet de type suivi via son identifiant (idSuivi) et son Libelle (suivi).

- public bool EstTransitionAutorisee(string nouvelleEtape)

Cette méthode prend en paramètre la nouvelle étape d'une commande. Si celle-ci est la même que l'étape actuelle alors true est renvoyé car rien ne se passe. Si l'étape est différente, alors on vérifie dans le dictionnaire TransitionsAutorisees si le changement peut se faire. Selon le résultat on renvoie true ou false.

Access.cs :

- public List<Suivi> GetAllSuivi()

Récupère tous les tuples suivis en base de données. Il s'agit de la commande GET pour l'API.

- public List<CommandeDocument> GetAllCommandeDocument(string idDocument)

Récupère toutes les commandes liées à un document dont l'id est passé en paramètre. Il s'agit de la commande GET pour l'API.

- public string CreerNouvelIdCommande()

Créé un nouvel id pour une commande en incrémentant l'id le plus élevé trouvé en base de données. Il s'agit de la commande GET pour l'API. Si aucune commande n'existe, l'id commence par défaut à « 00001 ».

- public bool CreerNouvelleCommandeDocument(CommandeDocument commandeDocument)

Enregistre une nouvelle commande de document (dans les tables commande et commandedocument) en base de données, via l'objet passé en paramètre. Il s'agit de la commande POST pour l'API.

- public bool ModifierCommandeDocument(CommandeDocument commandeModifiee)

Modifie une commande présente en base de données via l'objet passé en paramètre. Il s'agit de la commande PUT pour l'API.

- public bool SupprimerCommande(string idCommande)

Supprime de la base de données, la commande dont l'id est passé en paramètre. Il s'agit de la commande DEL pour l'API.

FrmCommandesController.cs :

FrmCommandesController.cs reprend de nombreuses méthodes déjà présentes dans FrmMediatekController.cs. A savoir :

- public FrmCommandesController()
- public List<Categorie> GetAllGenres()
- public List<Livre> GetAllLivres()
- public List<Dvd> GetAllDvd()
- public List<Revue> GetAllRevues()
- public List<Categorie> GetAllRayons()
- public List<Categorie> GetAllPublics()
- public List<Exemplaire> GetExemplairesRevue(string idDocuement)

La première méthode est le constructeur qui appelle une instance de la classe Access. Le reste des méthodes récupèrent tous les tuples d'une table en base de données. En ce qui concerne les méthodes sur les revues, elles ne sont pas utiles pour le moment. Je les ai mises dans le code en prévision de la prochaine tâche de cette mission. Je n'ai pas écrit ou modifié les méthodes ci-dessus, mais pour une meilleure compréhension du reste, je pense qu'il est judicieux de les mettre dans le rapport.

Voici maintenant les méthodes que j'ai écrites, sur lesquelles je ne m'épancherai pas car elles font toutes appel à leurs méthodes homonymes de la classe Access (décrites plus haut) :

- public List<Suivi> GetAllSuivis()
- public List<CommandeDocument> GetCommandesDocument(string idDocument)
- public string CreerNouvelIdCommande()
- public bool CreerNouvelleCommandeDocument(CommandeDocument commandeDocument)
- public bool ModifierCommandeDocument(CommandeDocument commandeDocument)
- public bool SupprimerCommandeDocument(string idCommande)

FrmCommandes.cs :

<u>Région commun :</u>

- internal FrmCommandes()

Il s'agit du constructeur du formulaire. Il appelle le contrôleur FrmCommandesController.cs.

- public void SelectionnerOnglet(int index)

Sélectionne le bon onglet de tabCommandes selon l'index passé en paramètre

- private void RemplirComboSuivi(List<Suivi> lesSuivis, BindingSource bdg, ComboBox cbx)

Rempli le combobox contenant les différentes étapes de suivi d'une commande. L'utilisateur voit le libelle et la valeur des items est l'id du suivi.

- private void tabCommandes_SelectedIndexChanged(object sender, EventArgs e)

Méthode évènementielle au changement d'onglet de tabCommandes : Permet de remplir les combos d'étapes, de supprimer toute information inutile qui aurait persisté et de gérer les visuels des contrôles selon le statut de la commande (en cours d'édition ou non, en modification ou non).

Région commandes livres :

- private void RemplirInformationsDetailleesLivre(Livre livre)

Rempli les contrôles concernant les informations détaillées du livre passé en paramètre.

- private void ViderInformationsLivre()

Vide les informations détaillées du livre qui était en cours de consultation par l'utilisateur et remet le focus sur la zone de saisie de numéro après avoir effacé aussi cette dernière.

- private void AfficherCommandesLivre()

Affiche les commandes liées au livre dont le numéro a été recherché par l'utilisateur en appelant la méthode GetCommandeDocument(idLivre) du contrôleur et la méthode local RemplirCommandesLivre().

- private void RemplirCommandesLivre(List<CommandeDocument> commande)

Rempli le datagridview avec la liste de commande(s) reçue en paramètre, et gère l'affichage de ce dernier. Gère les visuels des boutons selon la commande qui est sélectionnée : le bouton supprimer n'est disponible que si la commande n'est pas livrée. Le bouton modifier est accessible à partir du moment où il y a au moins une commande dans la liste.

- private void CommandeLivreEnCoursModification(Boolean modif)

Gère les visuels de l'interface selon si la commande est en train d'être modifiée ou ajoutée.

- private void CommandeLivreEnCoursEdition(Boolean edition)

Gère les visuels de l'interface selon si la commande est en cours d'édition ou non (accès à l'édition de la commande en bas de l'interface).

- private void tabCommandeLivre_Enter(object sender, EventArgs e)

Charge les informations nécessaires et l'affichage de l'interface à l'entrée sur l'onglet (rempli le combo étapes, récupère la liste des livres disponibles auprès du contrôleur, met les méthodes CommandeLivreEnCoursModification et CommandeLivreEnCoursEdition à false et vide les informations livres pour éviter une éventuelle persistance.

- private void btnCommandeLivreRechercher_Click(object sender, EventArgs e)

Au clic sur le bouton Rechercher, si le numéro saisi par l'utilisateur est valide, affiche les informations ainsi que les commandes du livre et autorise l'accès au bouton d'ajout d'une commande.

- private void txtCommandeLivreRechercheNumero_Enter(object sender, EventArgs e)

Si l'utilisateur clic sur la zone de saisie de numéro, alors toutes les informations liées au livre recherché précédemment sont vidées ainsi que les commandes.

- private void dgvCommandeLivre_ColumnHeaderMouseClick(object sender, DataGridViewCellMouseEventArgs e)

Permet le tri sur les colonnes du datagridview

- private void btnCommandeLivreAjouter_Click(object sender, EventArgs e)

Au clic sur le bouton « Ajouter une commande », met l'édition et la modification à true, interdit le changement du combobox et le met obligatoirement sur l'étape « En cours ».

- private void btnCommandeLivreValider_Click(object sender, EventArgs e)

Clic par l'utilisateur sur le bouton « Valider l'opération » après avoir mis les informations de commande souhaitées. Si cela marche, l'utilisateur à un message de succès. Sinon, un message d'erreur s'affiche. Après validation, l'interface se remet en mode non édition et non modification.

Dans le cas d'une modification, la méthode ModifierCommandeDocument du contrôleur est appelée. Dans le cas d'un ajout c'est la méthode CreerNouvelleCommandeDocument qui est appelée après avoir effectué CreerNouvelIdCommande().

La liste des commandes du datagridview est ensuite mise à jour.

- private void btnCommandeLivreAnnuler_Click(object sender, EventArgs e)

L'utilisateur clique sur le bouton pour annuler l'opération de modification ou d'ajout. Une fenêtre s'ouvre pour confirmer l'annulation puis remet l'interface en mode non édition et non modification.

- private void btnCommandeLivreModifier_Click(object sender, EventArgs e)

L'utilisateur clic sur le bouton pour modifier une commande. La zone d'édition est accessible et remplie avec les informations de la commande sélectionnée dans le datagridview. Si aucune commande n'est sélectionnée, l'utilisateur voit s'afficher un message d'erreur.

- private void cbxCommandeLivre_SelectedIndexChanged(object sender, EventArgs e)

Lorsqu'en cours de modification d'une commande, l'utilisateur essaye de changer l'étape dans le combobox, cette méthode vérifie que la transition entre étape est autorisée. Par exemple, une commande réglée ne peut pas revenir au stade « en cours ». Si la transition est interdite alors l'utilisateur reçoit un message d'erreur.

- private void dgvCommandeLivre_SelectionChanged(object sender, EventArgs e)

Gère les visuels des boutons de l'interface selon la commande sélectionnée : il faut qu'une commande soit sélectionnée pour que le bouton modifier soit accessible. Il faut qu'une méthode ne soit pas encore livrée (donc qu'elle soit en cours ou relancée) pour que le bouton supprimer soit disponible.

- private void btnCommandeLivreSupprimer_Click(object sender, EventArgs e)

Au clic sur supprimer, appel de la méthode SupprimerCommandeDocument(idCommande) du contrôleur. La liste des commandes du datagridview est ensuite mise à jour.

<u>Région commandes de dvd :</u>

Afin de ne pas alourdir le rapport inutilement, je vais faire la liste des méthodes ajoutées à FrmCommandes.cs dans la région DVD sans donner d'explication. En effet, les méthodes sont les mêmes que pour les livres et suivent la même logique. Ce qui change, c'est que les méthodes s'appliquent sur des objets de type DVD qui sont récupérés (de la même façon que les livres) dans une liste liée à un BindingSource auprès du contrôleur grâce à la méthode GetAllDvd().

Les méthodes et les contrôles associés ont les mêmes noms et signatures que les précédentes excepté qu'au lieu de « livre », on trouve « dvd ». Par exemple, AfficherCommandesLivre() deviendra AfficherCommandesDvd(). Les livres et les dvd étant tous deux des commande de

documents, ils font appel aux mêmes méthodes dans le contrôleur pour l'ajout, la modification ou la suppression en base de données.

- private void tabCommandeDvd_Enter(object sender, EventArgs e)
- private void ViderInformationsDvd()
- private void RemplirInformationsDetailleesDvd(Dvd leDvd)
- private void AfficherCommandesDvd()
- private void RemplirCommandesDvd(List<CommandeDocument> lesCommandesDvd)
- private void CommandeDvdEnCoursEdition(Boolean edition)
- private void CommandeDvdEnCoursModification(Boolean modif)
- private void txtCommandeDvdNumero_Enter(object sender, EventArgs e)
- private void btnCommandeDvdRechercher_Click(object sender, EventArgs e)
- private void btnCommandeDvdAjouter_Click(object sender, EventArgs e)
- private void btnCommandeDvdModifier_Click(object sender, EventArgs e)
- private void btnCommandeDvdValider_Click(object sender, EventArgs e)
- private void btnCommandeDvdAnnuler_Click(object sender, EventArgs e)
- private void btnCommandeDvdSupprimer_Click(object sender, EventArgs e)
- private void cbxCommandeDvd_SelectedIndexChanged(object sender, EventArgs e)
- private void dgvCommandeDvd_ColumnHeaderMouseClick(object sender, DataGridViewCellMouseEventArgs e)

FrmMediatek.cs :

Région commun :

- private void OuvrirCommandes(int indexOnglet)

Ouvre un formulaire de FrmCommandes.cs sur l'onglet choisi en index

Région Livre :

- private void btnGestionCommandesLivres_Click(object sender, EventArgs e)

Au clic sur le bouton « gérer les commandes », ouvre le formulaire sur l'onglet de commande des livres en appelant OuvrirCommandes(0).

Région Dvd :

- private void btnGestionCommandeDvd_Click(object sender, EventArgs e)

Au clic sur le bouton « gérer les commandes », ouvre le formulaire sur l'onglet de commande des dvd en appelant OuvrirCommandes(1).

API :

Comme dit précédemment, tous les ajouts se sont faits dans la classe MyAccessBDD.php. Voici les méthodes créées ou modifiées accompagnées de quelques explications :

1- Méthodes créées :

- private function selectCommandesLivreDvd(?array $champs): ?array

Récupère tous les tuples des tables commande et commandedocument dont l'id (idLivreDvd) est passé en paramètre via $champs. Si l'opération est réussie, alors retourne un tableau de commandes.

- private function insertCommandeDocument(?array $champs): ?int

Créé une nouvelle commande de document en base de données en appelant la procédure stockée InsertCommandeDocument. La commande à insérer est passée en paramètre dans $champs. La méthode retourne 1 si l'opération s'est bien déroulée.

- private function selectAllCommandes(): ?array

Sélectionne toutes les commandes présentes en base de données (ainsi que les abonnements quand il y en aura. Cette fonction me sert ensuite à trouver l'id maximum d'une commande dans la classe Access de l'application C# et à l'incrémenter afin d'avoir un nouvel id.

- private function updateCommandeDocument(?array $champs) : ?int

Modifie une commande selon l'id envoyé dans $champs. La modification impacte les tables commande et commandedocument. Pour se faire, la procédure updateCommandeDocument est appelée en base de données.

2- Méthodes modifiées :

- protected function traitementSelect(string $table, ?array $champs): ?array

Traitement select s'occupe du chemin GET de l'API. J'ai rajouté un cas du switch « suivi » qui permet d'appeler la méthode selectTableSimple($table) et de retourner tous les tuples de la table suivi. J'ai rajouté un cas « commandedocument » qui appelle la méthode selectCommandesLivreDvd($champs) décrite plus haut. Enfin, j'ai rajouté un cas commandes qui appelle la méthode selectAllCommandes(), et renvoie tous les tuples de la table commande.

- protected function traitementInsert(string $table, ?array $champs): ?int

Traitement insert s'occupe du chemin POST de l'API. J'ai rajouté un cas dans le switch où la table est « commandedocument », qui permet d'appeler la méthode décrite plus haut insertCommandeDocument.

- protected function traitementUpdate(string $table, ?string $id, ?array $champs): ?int

Traitement update s'occupe du chemin PUT de l'API. J'ai rajouté un cas dans le switch où la table est « commandedocument », qui permet d'appeler la méthode décrite plus haut updateCommandeDocument($champs).

Pour le cas de la suppression, je n'ai pas eu à ajouter ou modifier de méthode. En effet, traitementDelete appelle par défaut la fonction deleteOneTupleOneTable et j'ai géré la suppression dans la base de données grâce à un trigger que nous verrons dans la prochaine section. Par rapport à l'application C#, la classe Access.cs envoie à l'api une demande de suppression en envoyant la table commande ainsi que l'id de la commande à supprimer, le tout séparé par un slash.

Base de données :

Procédures stockées :

Dans ma base de données, j'utilise deux procédures stockées :

- InsertCommandeDocument

Cette procédure me permet d'insérer les informations nécessaires à la création d'une commande dans les tables commande et commandedocument. L'insertion se fait au sein d'une transaction pour assurer que celle-ci se déroule bien dans les deux tables. En effet, en cas de problème, aucune insertion ne se fait.

- updateCommandeDocument

Cette procédure marche de la même façon que la précédente, avec une transaction pour s'assurer que la modification se fait correctement dans les tables commande et commandedocument.

Triggers :

- UpdateNouveauxExemplaires

Après une modification sur la table commandedocument, ce trigger se déclenche si la commande a un statut qui passe à « livré » (c'est-à-dire avec un nouvel id à « 00002 »). Il permet d'ajouter autant d'exemplaires dans la table exemplaire qu'il y a d'exemplaires dans la commande, en valorisant l'id avec l'id du livre ou du dvd commandé, la date d'achat qui correspond à la commande, un numéro qui est séquentiel par rapport aux exemplaires de ce document déjà disponibles et d'initialiser son état à neuf.

- deleteCommande

Après une suppression sur la table commande, ce trigger permet de supprimer dans la table commandedocument le tuple lié à la commande supprimée.

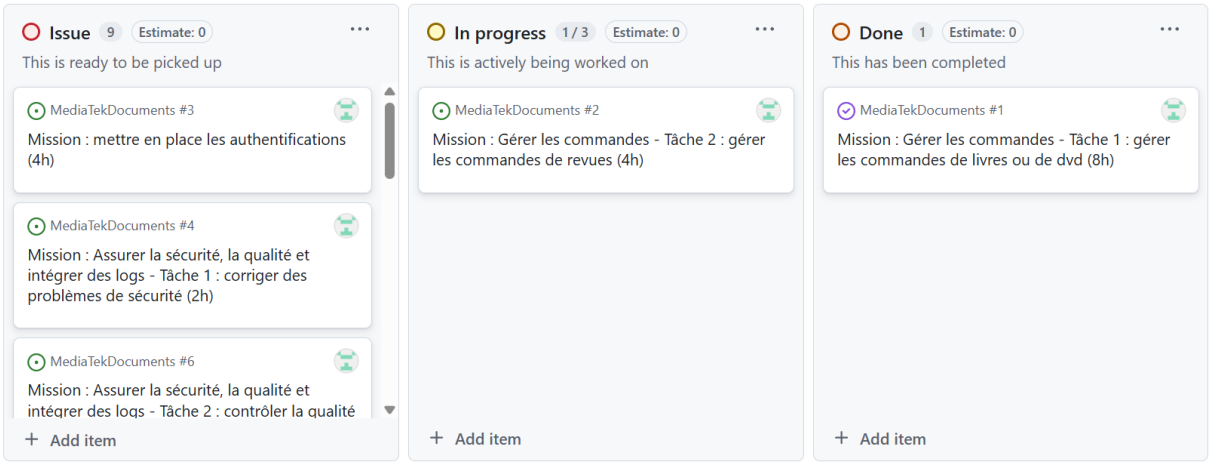
Tâche 2 : Gérer les commandes de revues

- Créer un onglet (ou une fenêtre) pour gérer les commandes de revues : une commande représente un nouvel abonnement ou le renouvellement d'un abonnement. Dans le cas d'un nouvel abonnement, la revue sera préalablement créée dans l'onglet Revues. Donc, dans l'onglet des commandes de revues, il n'y a pas de distinction entre un nouvel abonnement et un renouvellement.
- La charte graphique doit correspondre à l'existant.
- Dans toutes les listes, permettre le tri sur les colonnes.
- Permettre la sélection d'une revue par son numéro, afficher les informations de la revue ainsi que la liste des commandes (abonnements), triée par date (ordre inverse de la chronologie). La liste doit comporter les informations suivantes : date de la commande, montant et date de fin d'abonnement.
- Créer un groupbox qui permet de saisir les informations d'une nouvelle commande (nouvel abonnement ou renouvellement, le principe est identique) et de l'enregistrer.
- Permettre de supprimer une commande de revue uniquement si aucun exemplaire n'est rattaché (donc, en vérifiant la date de l'exemplaire, comprise entre la date de la commande et la date de fin d'abonnement). Pour cela, créer et utiliser la méthode

'ParutionDansAbonnement' qui reçoit en paramètre 3 dates (date commande, date fin abonnement, date parution) et qui retourne vrai si la date de parution est entre les 2 autres dates. Créer le test unitaire sur cette méthode.

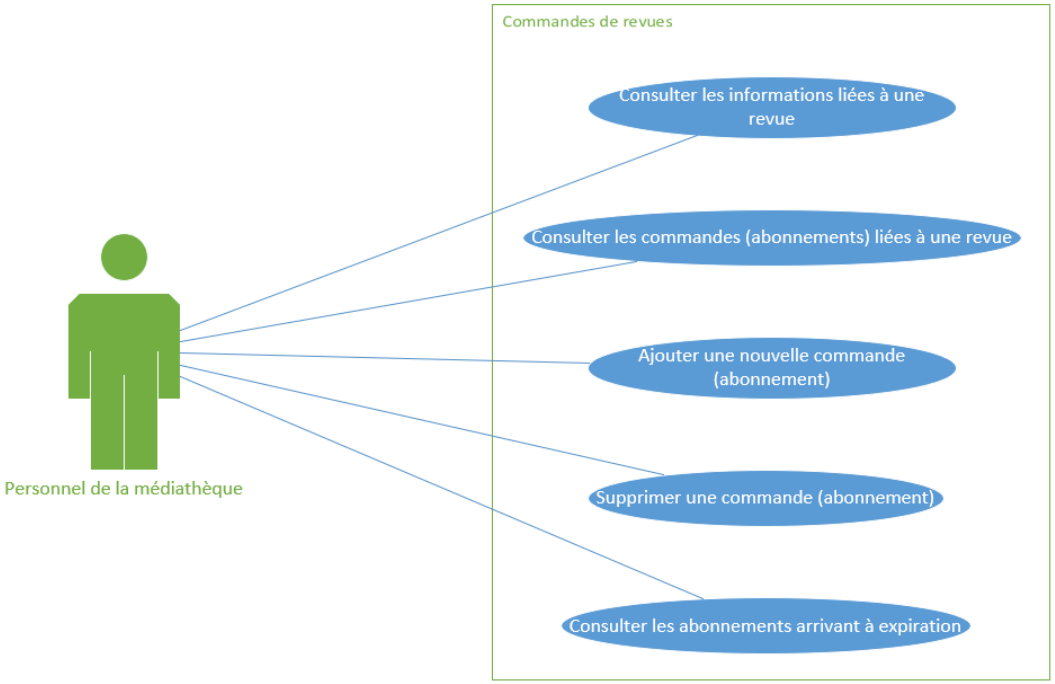
- Toutes les sécurités seront mises en place pour éviter des erreurs de manipulation.
- Créer une méthode qui permet d'obtenir la liste des revues dont l'abonnement se termine dans moins de 30 jours. Dès l'ouverture de l'application, ouvrir une petite fenêtre d'alerte rappelant la liste de ces revues (titre et date de fin abonnement) triée sur la date dans l'ordre chronologique.

Kanban



Temps estimé : 4h00 Temps réel : 5h00

Diagramme de cas d'utilisation



Maquettes – Figma

Maquette de l'onglet de gestion des commandes de revues (abonnements)

Maquette de la fenêtre d'alerte (expiration abonnements)

Explications sur les maquettes :

Maquette de commandes de revues :

Notons tout d'abord que j'ai choisi de parler d'abonnement et non de commandes dans les maquettes. Ce choix est plus cohérent vis-à-vis du modèle (abonnement était une catégorie fille de commande) mais aussi, pour le parcours utilisateur. Il y a en effet une date de début (date de la commande) et une date de fin d'abonnement. Cela permet aussi de mieux trancher avec les commandes de livres et de dvd qui demandent d'autres informations à saisir que celles qui nous intéressent ici.

On retrouve en haut de cette maquette une barre de recherche qui permet à l'utilisateur de rechercher une revue en tapant son numéro et en cliquant sur le bouton « Rechercher ». Dès lors, si le numéro de revue est valide, les informations détaillées la concernant s'affichent.

Ces informations détaillées, présentes en dessous de la barre de recherche, reprennent le même modèle que celles que l'on peut voir dans la consultation des revues : titre, périodicité, délai de mise à disposition, public, genre, rayon, chemin de l'image de la revue et image de la revue. En outre, on retrouve tous les exemplaires (parutions) rattachées à cette dernière dans un datagridview. En sélectionnant un exemplaire dans la liste, l'image de ce dernier s'affiche.

Un groupbox d'édition a été rajouté, à l'image de ce qui a été fait pour les commandes de livres et de dvd. Au chargement, il est inaccessible à l'utilisateur. Pour le rendre utilisable, il faut que l'utilisateur clique sur « Ajouter un nouvel abonnement/renouvellement ». La différence réside dans les informations à donner : si l'on retrouve la date de la commande et le montant, il faut aussi donner la date de fin de l'abonnement. Une fois ces renseignements remplis, l'utilisateur peut soumettre et valider la commande ou annuler l'opération. S'il valide l'opération et que les champs demandés sont correctement renseignés alors le nouvel abonnement s'affiche dans la liste. S'il clique sur annuler alors les informations saisies sont effacées et le groupbox d'édition redevient inaccessible.

L'utilisateur peut aussi cliquer sur supprimer un abonnement. Pour que ce bouton soit accessible, une commande doit être sélectionnée dans la liste des abonnements. Il ne faut pas qu'un exemplaire (une parution) soit liée à la commande sélectionnée (c'est-à-dire qu'un exemplaire ait une date d'achat comprise entre la date de la commande et la date de la fin de l'abonnement).

Maquette d'alerte sur les abonnements arrivant à expiration :

Cette fenêtre s'ouvre au lancement de l'application et contient la liste des revues (titre et la date de fin d'abonnement) dont l'abonnement arrive à expiration dans les 30 jours. Elle se compose simplement d'un label explicatif, d'un datagridview contenant la liste des abonnements et d'un bouton permettant de continuer vers l'application. En cliquant sur ce bouton, la fenêtre permettant de consulter les livres s'ouvre.

Explications sur le code

Gestion des abonnements :

Model :

Au niveau du modèle, j'ai ajouté la classe Abonnement.cs qui correspond à la table abonnement de la base de données. Elle hérite de la classe Commande. Voici son constructeur :

- public bool ParutionDansAbonnement(DateTime dateCommande, DateTime dateFinAbonnement, DateTime dateParution)

Elle contient aussi la méthode ParutionDansAbonnement qui prend en paramètre la date de la commande, la date de la fin de l'abonnement et la date de parution et qui renvoie true si la date de parution est comprise entre la date de début de l'abonnement et la date de fin.

Dal :

Dans la classe Access :

- public bool CreerNouvelAbonnement(Abonnement abonnement)

Cette méthode créé un nouvel abonnement en appelant le chemin POST de l'API, avec « abonnement » en table et un objet de type Abonnement (formaté en json) dans les champs. Si l'ajout réussi alors la méthode retourne true.

- public List<Abonnement> GetAllAbonnement(string idDocument)

Cette méthode retournent tous les abonnements appartenant à une revue passée en paramètre. Elle appelle le chemin GET de l'API avec « abonnement/ » en table et l'id de la revue (formaté en json) en champs. Elle retourne une liste d'objets de type Abonnement.

Contrôleur :

Dans la classe FrmCommandesController, j'ai rajouté les appels aux méthodes précédentes de la classe Access. Voici leurs signatures :

- public bool CreerNouvelAbonnement(Abonnement abonnement)
- public List<Abonnement> GetAllAbonnements(string idDocument)

Vue :

Dans la classe FrmMediatek, région Revue, j'ai ajouté la méthode évènementielle au clic sur le bouton « Gérer les commandes » de l'onglet Revue qui permet d'ouvrir la fenêtre de gestion des commandes directement sur l'onglet revue. Voici sa signature :

- private void btnRevuesCommandes_Click(object sender, EventArgs e)

Dans la classe FrmCommandes, dans la région Commun :

- private void tabCommandes_SelectedIndexChanged(object sender, EventArgs e)

J'ai ajouté une condition à cette méthode qui permet d'initialiser les éléments lorsque la sélection d'onglet change. Elle appelle la méthode GetAllRevues de FrmCommandesController et stocke le résultat dans la liste lesRevues (liste d'objets de type Revue). Elle appelle les méthodes qui gèrent les affichages à l'arrivée sur un onglet et qui seront vues en détail plus loin : AbonnementEnCoursEdition(false), ViderInformationsRevue() et GestionBoutonGestion().

Dans la région Revue :

- private void GestionBoutonGestion()

Cette méthode appelle les deux méthodes suivantes : GestionBoutonAjout et GestionBoutonSuppression. En effet, la possibilité d'ajouter ou de supprimer des éléments étant conditionnées à des contraintes, les boutons ne sont pas toujours utilisables (afin d'éviter toute erreur). J'ai choisi de faire une méthode par bouton afin de les simplifier le plus possible et d'avoir juste à appeler cette méthode de gestion lorsque je dois gérer l'affichage.

- private void GestionBoutonAjout()

Cette méthode rend le bouton pour créer une nouvelle commande accessible si un numéro de revue a été recherché par l'utilisateur et si aucun abonnement est en cours d'édition.

- private void GestionBoutonSuppression()

Cette méthode rend le bouton pour supprimer une commande accessible si une commande est sélectionnée dans le datagridview des abonnements et si aucun abonnement est en cours d'édition. Si ces conditions sont remplies, alors elle fait appel à la méthode ParutionDansAbonnement pour vérifier si aucun exemplaire de revue (parution) est rattaché à la commande. Si cela n'est pas le cas, le bouton supprimer est inaccessible à l'utilisateur.

- private void ViderInformationsRevue()

Cette méthode vide toutes les informations relatives à une revue qui aurait été recherchée par l'utilisateur. Elle met la méthode AbonnementEnCoursEdition à false, appelle la méthode GestionBoutonGestion pour gérer l'affichage des boutons pour ajouter ou supprimer une commande (qui doivent donc devenir inaccessibles) et remet le focus sur la zone de saisie de numéro de revue afin de permettre à l'utilisateur de faire une nouvelle recherche.

- private void AfficheExemplairesRevue()

Cette méthode appelle GetExemplairesRevue du contrôleur en lui passant en paramètre l'id de la revue, c'est-à-dire le numéro de revue qui a été recherché par l'utilisateur. Elle stocke le résultat de cet appel dans la variable lesExemplaires. Cette variable est passée en paramètre de la méthode RemplirExemplairesListe.

- private void RemplirExemplairesListe(List<Exemplaire> exemplaires)

Cette méthode remplit le datagridview avec les exemplaires reçus en paramètre, issus de la méthode AfficheExemplairesRevue.

- private void dgvAbonnementParutions_ColumnHeaderMouseClick(object sender, DataGridViewCellMouseEventArgs e)

Cette méthode permet à l'utilisateur de trier les exemplaires (parutions) du datagridview au clic sur une des en-têtes de celui-ci.

- private void RemplirInformationsRevue(Revue laRevue)

Cette méthode rempli les informations détaillées d'une revue recherchée (titre, périodicité, délai de mise à disposition, genre, public, rayon, chemin de l'image et image). Ensuite, elle appelle la méthode AfficheExemplaireRevue afin de remplir le datagridview des parutions avec les exemplaires de la revue.

- private void AfficheAbonnementsRevue()

Cette méthode appelle GetAllAbonnements du contrôleur passant en paramètre l'id de la revue, c'est-à-dire le numéro de revue qui a été recherché par l'utilisateur. Elle stocke le résultat de cet appel dans la variable lesAbonnements. Elle appelle ensuite la méthode RemplirAbonnementsListe en lui passant en paramètre lesAbonnements.

- private void RemplirAbonnementsListe(List<Abonnement> lesAbonnements)

Cette méthode remplit le datagridview des commandes avec les abonnements reçus en paramètre et issus de la méthode AfficheAbonnementsRevue. Elle gère l'affichage au sein du datagridview (données à afficher, dans quel sens, nom des header).

- private void AbonnementEnCoursEdition(Boolean edition)

Cette méthode gère l'affichage des contrôles selon si un abonnement est en train d'être ajouté ou non afin d'empêcher l'utilisateur de faire une autre opération tant qu'il n'a pas valider ou annuler celle en cours.

- private void tabCommandeAbonnement_Enter(object sender, EventArgs e)

Cette méthode fait la même chose que tabCommandes_SelectedIndexChanged. Elle se déclenche lorsque l'on entre dans l'onglet (alors que l'autre se déclenche lorsque l'on sélectionne un nouvel onglet de la fenêtre de gestion des commandes).

- private void btnAbonnementRechercher_Click(object sender, EventArgs e)

Cette méthode est déclenchée par l'utilisateur lorsqu'il clique sur le bouton « Rechercher ». Elle vérifie qu'un numéro de revue a été saisie et est valide. Si ce n'est pas le cas, alors elle envoie un message d'erreur et remet le focus sur la zone de saisie de numéro en gérant l'affichage des boutons de gestion (ajouter/supprimer). S'il est correct alors elle parcourt la liste lesRevues pour trouver celle recherchée par l'utilisateur. Si la revue est valide, elle l'envoie en paramètre de la méthode RemplirInformationsRevue pour afficher les informations détaillées, AfficheExemplaireRevue afin de remplir le datagridview des parutions, AfficheAbonnementsRevue pour le datagridview des commandes puis GestionBoutonGestion pour gérer l'affichage des boutons ajouter et supprimer.

- private void dgvAbonnementCommandes_ColumnHeaderMouseClick(object sender, DataGridViewCellMouseEventArgs e)

Cette méthode se déclenche au clic sur le header du datagridview des abonnements. Elle permet de trier les abonnements selon leurs propriétés.

- private void btnAbonnementAjouter_Click(object sender, EventArgs e)

Cette méthode se déclenche au clic sur le bouton « Créer un nouvel abonnement / renouvellement ». Elle met la méthode AbonnementEnCoursEdition à true.

- private void btnAbonnementValider_Click(object sender, EventArgs e)

Cette méthode se déclenche au clic sur le bouton « Valider l'opération ». Elle vérifie d'abord si un numéro de revue a été saisi. Si tel n'est pas le cas alors elle envoie un message d'erreur, réinitialise les contrôles en mode « non-édition » et remet le focus sur la zone de saisie de numéro. Si le numéro de commande existe, elle appelle la méthode CreerNouvelIdCommande du contrôleur et valorise les propriétés d'un objet de type Abonnement (id, dateCommande, montante, dateFinAbonnement, idRevue) dans la variable nouvelAbonnement. Elle vérifie si la date de fin d'abonnement est supérieure à la date de commande et envoie un message d'erreur si ce n'est pas le cas. Si tout est correct, elle appelle la méthode CreerNouvelAbonnement du contrôleur en lui passant la variable nouvelAbonnement en paramètre. Si le résultat de cet appel est true, elle montre un message de succès, met à jour le datagridview des commandes et ferme le groupbox d'édition d'un abonnement. Si le résultat est faux, alors un message d'erreur apparaît.

- private void btnAbonnementAnnuler_Click(object sender, EventArgs e)

Cette méthode se déclenche lorsque l'on clique sur le bouton « Annuler l'opération ». Elle met la méthode AbonnementEnCoursEdition à false et gère l'affichage des boutons avec GestionBoutonGestion.

- private void btnAbonnementSupprimer_Click(object sender, EventArgs e)

Cette méthode est déclenchée par le clic sur le bouton « Supprimer l'abonnement sélectionné ». Elle récupère l'abonnement sélectionné dans le datagridview des commandes et le stocke dans la variable abonnementSelected. Ensuite, elle valorise la variable local idCommande avec l'id de celui-ci. Un message d'alerte apparaît invitant l'utilisateur à confirmer la suppression. S'il confirme alors la méthode SupprimerCommande est appelée avec en paramètre l'id de la commande (cette méthode est explicitée plus haut car elle est commune à toutes les commandes qu'elles soient des abonnements ou des documents). Si le résultat de l'appel est true alors le datagridview est mis à jour et la méthode GestionBoutonGestion est appelée pour gérer l'affichage des boutons ajouter et supprimer.

API :

- private function selectAbonnementsRevue(?array $champs) : ?array

Cette méthode renvoie tous les abonnements relatifs à une revue dont l'id est passée dans le paramètre champs.

- private function insertAbonnement(?array $champs): ?int

Cette méthode créer un nouvel abonnement en base de données, avec les informations envoyées dans le paramètre champs. Elle appelle la procédure stockée insertAbonnement de la base de données.

- protected function traitementSelect(string $table, ?array $champs): ?array

Dans cette méthode, qui correspond au chemin GET de l'API, j'ai ajouté un cas dans le switch « abonnement ». Ce cas appelle la fonction selectAbonnementsRevue.

- protected function traitementInsert(string $table, ?array $champs): ?int

Dans cette méthode, qui correspond au chemin POST de l'API, j'ai ajouté un cas « abonnement dans le switch. Ce cas appelle la fonction insertAbonnement.

Base de données :

- dans le trigger deleteCommande, j'ai ajouté une requête sql qui supprime l'abonnement lié à la commande, après une opération delete sur la table commande.
- J'ai ajouté la procédure stockée insertAbonnement qui prend en paramètre : id, dateCommande, montant, dateFinAbonnement et idRevue. Il s'agit d'une transaction car une partie des paramètres concernent la table commande et l'autre la table abonnement. Si la transaction rencontre un problème, alors un rollback est effectué.

Fenêtre d'alerte :

Model :

J'ai créé la classe AbonnementExpiration qui ne correspond pas à une table dans la base de données mais qui va me permettre de créer des objets qui porteront le titre de la revue dont l'abonnement arrive à expiration dans moins de 30 jours. Voici le constructeur :

- public AbonnementExpiration(string titre, DateTime dateFinAbonnement)

dal :

- public List<AbonnementExpiration> GetAllAbonnementExpiration()

Cette méthode appelle le chemin GET de l'api en lui envoyant comme table « expiration » et retourne la liste des abonnements arrivant à expiration.

Contrôleur :

J'ai créé le contrôleur FrmAlerteAbonnementsController qui correspond à la vue FrmAlerteAbonnements. Il ne contient que deux méthodes :

- public FrmAlerteAbonnementsController()

Il s'agit du constructeur de la classe.

- public List<AbonnementExpiration> GetAllAbonnementExpiration()

Cette méthode fait appel à GetAllAbonnementExpiration de la classe access (décrite ci-dessus).

Vue :

- private void FrmMediatek_Load(object sender, EventArgs e)

Cette méthode est présente dans FrmMediatek. Elle instancie un nouveau formulaire FrmAlerteAbonnements au chargement de FrmMediatek et affiche la fenêtre à l'utilisateur.

Les méthodes suivantes concernent la classe FrmAlerteAbonnements :

- public FrmAlerteAbonnements()

Il s'agit du constructeur de la classe qui initialise les composants et appelle le contrôleur FrmAlerteAbonnementsController.

- public void RemplirListeAbonnementsExpiration()

Cette méthode appelle GetAllAbonnementExpiration du contrôleur et stocke le résultat dans la variable lesAbonnementsExpire. Si la liste n'est pas nulle, alors le datagridview affiche les abonnements qui vont expirer dans 30 jours (nom de la revue + date de la fin d'abonnement). En outre, elle met la variable existenceAbonnementExpire à true.

- public void AffichageAbonnements()

Cette méthode vérifie le statut de la variable existenceAbonnementExpire. Si cette dernière est à true, alors elle affiche le datagridview avec un label informant l'utilisateur que des abonnements arrivent à expiration. Si elle est à false, le datagridview n'est pas visible et le label informe qu'aucun abonnement n'arrive à expiration.

- private void FrmAlerteAbonnements_Load(object sender, EventArgs e)

Cette méthode évenementielle se déclenche au chargement du formulaire et appelle les deux méthodes précédentes afin de gérer l'affichage et la liste des abonnements qui arrivent à expiration.

- private void btnAlerteAbonnementsFermer_Click(object sender, EventArgs e)

Cette méthode se déclenche au clic sur le bouton « Continuer vers l'application ». Elle ferme la fenêtre en cours (et par conséquent ouvre la fenêtre de consultation des documents FrmMediatek). Ce bouton n'était pas indispensable car l'on peut fermer la fenêtre en cliquant sur la croix en haut, mais je trouvais que cela apportait une meilleure expérience utilisateur.

<u>API :</u>

- private function selectRevueExpirationMois(): ?array

Cette méthode retourne les titres et la date de fin d'abonnement des revues arrivant à expiration dans moins de 30 jours. Concrètement, elle cherche les revues dont la date de fin d'abonnement est comprise entre aujourd'hui et dans 30 jours.

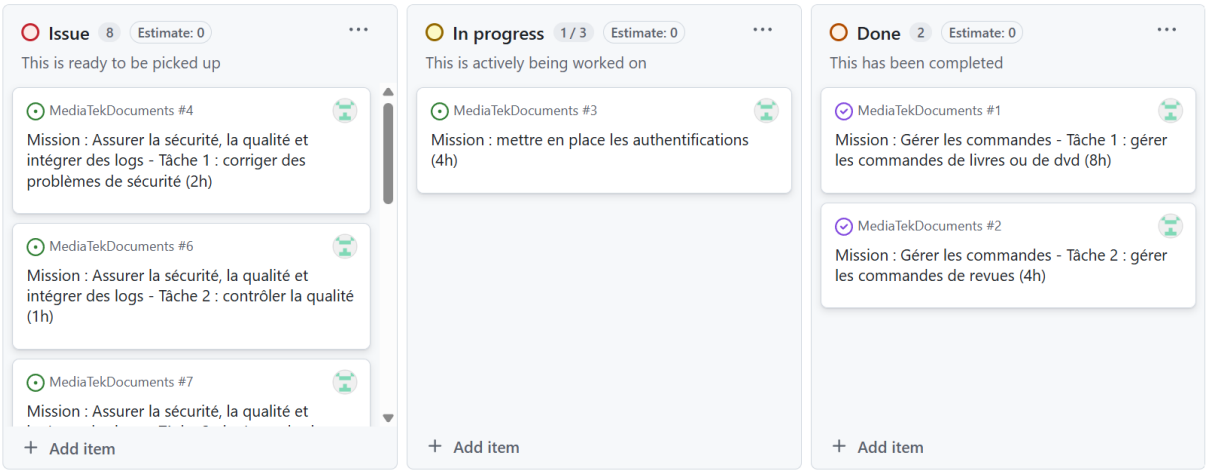
- protected function traitementSelect(string $table, ?array $champs): ?array

J'ai rajouté au switch un test sur « expiration » qui retourne la fonction selectRevueExpirationMois.

Mission : Mettre en place des authentifications

- Dans la base de données, ajouter une table Utilisateur et une table Service, sachant que chaque utilisateur ne fait partie que d'un service. Pour réaliser les tests, remplir les tables d'exemples.
- Ajouter une première fenêtre d'authentification. Faire en sorte que l'application démarre sur cette fenêtre.
- Suivant le type de personne authentifiée, empêcher certains accès en rendant invisibles ou inactifs certains onglets ou objets graphiques.
- Dans le cas du service Culture qui n'a accès à rien, afficher un message précisant que les droits ne sont pas suffisants pour accéder à cette application, puis fermer l'application.
- Faire en sorte que l'alerte de fin d'abonnement n'apparaisse que pour les personnes concernées (qui gèrent les commandes).

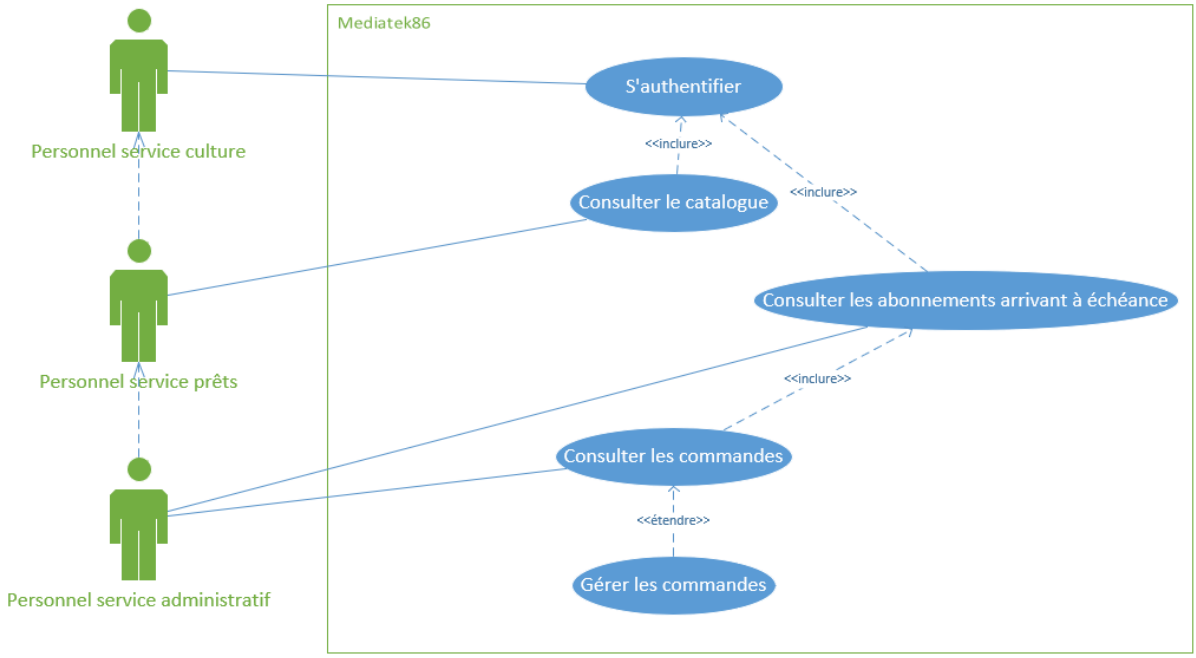
Kanban



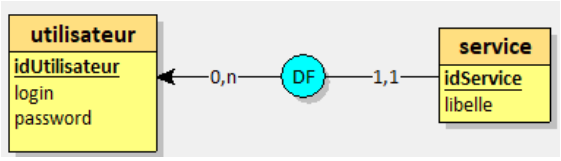
Temps estimé : 4h

Temps réel : 5h

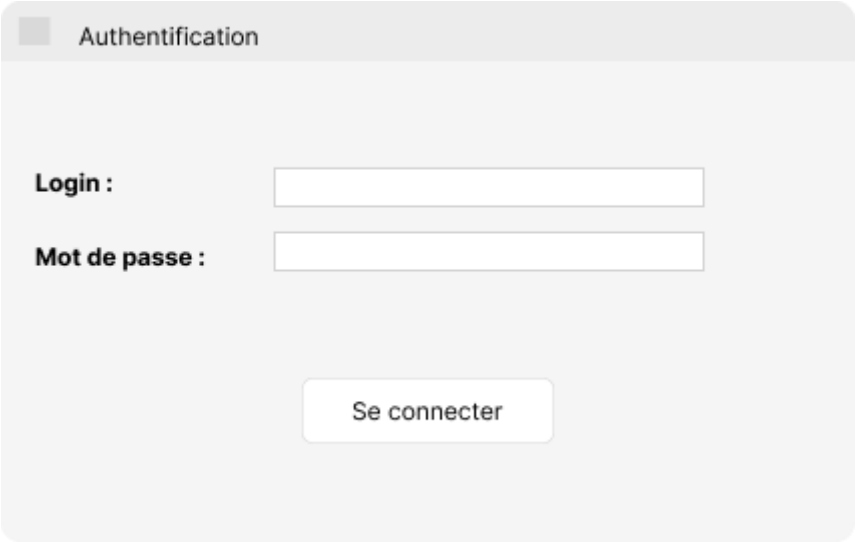
Diagramme de cas d'utilisation



MCD



Maquettes – Figma



Explications sur le code

Base de données :

Deux tables ont été rajoutées service et utilisateur. Service a comme champs idService et libelle. Utilisateur a comme champs idUtilisateur, login, password et idService.

API :

Aucun code n'a été ajouté à l'API car le chemin utilisé est le cas « default » du switch contenu dans la méthode TraitementSelect.

Model :

La classe Utilisateur a été créée. Voici son constructeur :

- public Utilisateur(string login, string password, int idService, string libelle)

La classe Service a été créée et mise en statique afin de pouvoir avoir accès à la propriété IdService qui va nous permettre de restreindre les droits de l'utilisateur selon son appartenance à tel ou tel service. Elle ne contient qu'une propriété :

- public static int IdService { get; set; }

Un nouveau dossier utils avec la classe Utils a été rajouté. Ne sachant pas comment hasher un mot de passe en C#, cette classe a été trouvée sur internet. Le site dont elle vient est dans la description de cette dernière. Au début j'ai voulu m'en inspirer seulement mais cela revenait juste à traduire en français ce qui été donné, ce qui n'a pas beaucoup de sens. Elle est donc identique à celle du site. Cette classe contient deux méthodes :

- public static string HashPassword(string password)

Cette méthode permet de hasher un mot de passe passé en paramètre en SHA256.

- public static bool VerifyPassword(string inputPassword, string storedHash)

Cette méthode permet de comparer deux mots de passe passés en paramètre.

Access :

- public Utilisateur CheckUtilisateur(string login, string password)

Cette méthode prend en paramètre un login et un mot de passe. Elle envoie le login à l'API via le chemin GET et la table « utilisateur ». Si un utilisateur est retourné, elle récupère son mot de passe et appelle la fonction VerifyPassword de la classe Utils, pour comparer le mot de passe en base de données à celui fourni par l'utilisateur (qui a été préalablement hashé avec HashPassword). Si cette comparaison est bonne alors un utilisateur est retourné, sinon la méthode retourne null.

Contrôleur :

Comme dans les autres contrôleurs, la classe est composée d'un constructeur qui appelle une instance d'access :

- public FrmAuthentificationController()

Et de la méthode CheckUtilisateur qui fait référence à la méthode du même nom de la classe Access :

- public Utilisateur CheckUtilisateur(string login, string password)

Vue :

FrmAuthentification :

Le formulaire d'authentification ne contient qu'un constructeur et une méthode. Le constructeur est :

- public FrmAuthentification()

La méthode événementielle btnAuth_Click se déclenche lorsque l'utilisateur clique sur le bouton « Se connecter ». Elle récupère les identifiants saisis par l'utilisateur et appelle la méthode CheckUtilisateur du contrôleur. Si cette dernière retourne un utilisateur, elle valorise IdService de la classe Service. Si l'utilisateur fait partie du service culture, elle affiche un message disant que les droits ne sont pas assez élevés. Si l'utilisateur fait partie d'un des services pouvant se connecter à l'application alors FrmMediatek est lancé.

Si l'utilisateur peut accéder à l'application (service prêts, administratif et administrateur) alors DialogResult est valorisé à OK. Nous verrons le but dans les explications concernant Program.Cs.

Si la méthode CheckUtilisateur n'a rien retourné, c'est qu'aucun utilisateur correspondant au couple login/mot de passe n'a été trouvé. Un message est alors affiché pour informer l'utilisateur.

FrmMediatek :

Deux méthodes ont été ajoutées.

- private void FrmMediatek_Load(object sender, EventArgs e)

Cette méthode permet de gérer le visuel des contrôles avant que le formulaire ne s'affiche à l'utilisateur. Ainsi, si celui-ci fait partie du service des prêts (IdService == 2) alors les boutons permettant de gérer les commandes ne s'afficheront pas.

- private void FrmMediatek_Shown(object sender, EventArgs e)

Cette méthode permet d'afficher le formulaire FrmAlerteAbonnements pour le personnel du service administratif et pour l'administrateur. Elle se lance une fois que le formulaire FrmMediatek est affiché.

Program.cs :

Afin que l'application s'ouvre sur FrmAuthentification sans que celui ne soit le formulaire principal, je créé dans la méthode Main une nouvelle instance de FrmAuthentification. Je stocke dans result, le résultat de l'ouverture de frmAuthentification en tant que fenêtre modale. Dans FrmAuthentification, ce dernier est valorisé à OK si l'utilisateur est identifié. Cela entraîne la création de FrmMediatek comme formulaire principal (et dont la fermeture entraine la fermeture de l'application).

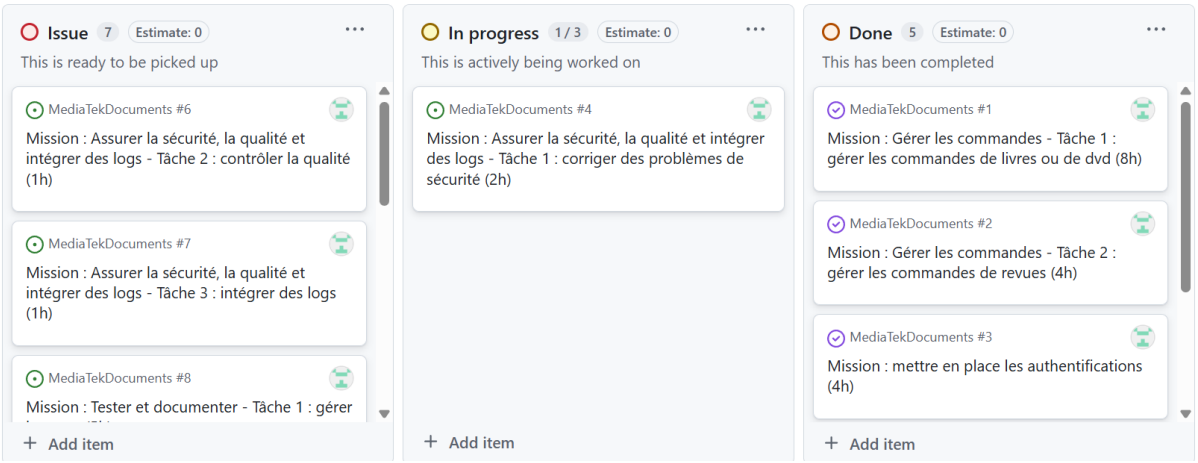
Mission : Assurer la sécurité, la qualité et intégrer des logs

Tâche 1 : Corriger des problèmes de sécurité

Pour chaque problème, créer une "issue" dans le dépôt GitHub correspondant, affecter l'issue à un des développeurs (si vous êtes 2, vous ou votre collègue, sinon vous-même), créer une branche pour proposer une correction de code en expliquant la correction, faire un pull request, accepter le pull request si le résultat correspond bien aux attentes.

- Problème n°1 : Actuellement, l'accès à l'API se fait en authentification basique, avec le couple "login:pwd" en dur dans le code de l'application (dans le constructeur de la classe Access). Le but est de sécuriser cette information.
- Problème n°2 : Si, pour accéder à l'API directement dans un navigateur, on donne juste l'adresse de l'API sans mettre de paramètres : http://localhost//rest_mediatekdocuments/ on obtient la liste des fichiers contenus dans le dossier de l'API. Le but est d'avoir un retour d'erreur.

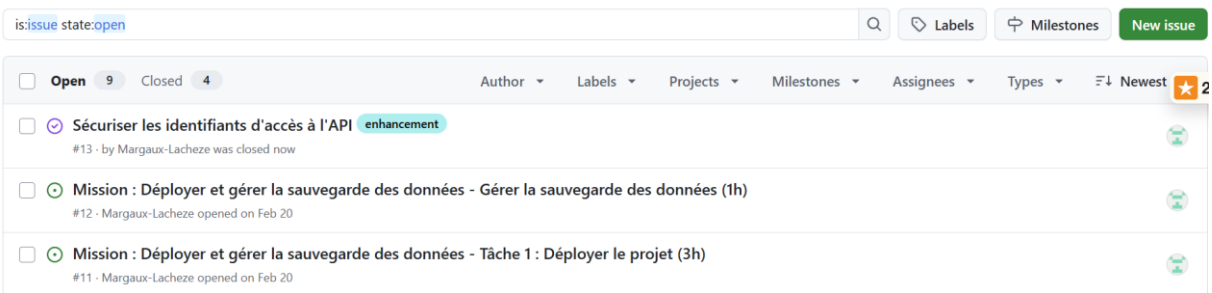
Kanban



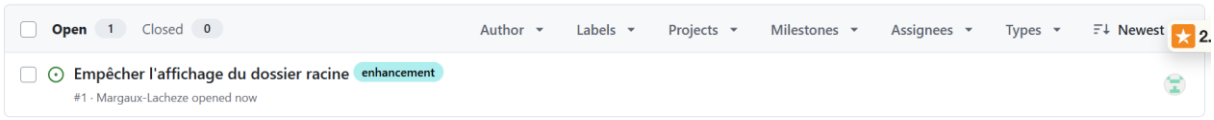
Temps estimé : 2h00 Temps réel : 1h30

Issues

<u>N° 1 : Dépôt Mediatekdocuments :</u>

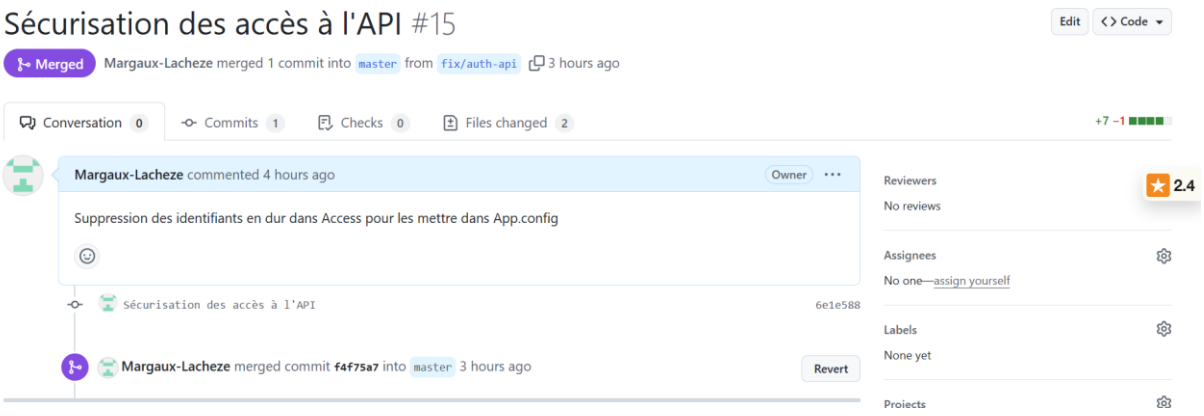


N° 2 : Dépôt rest_mediatekdocuments

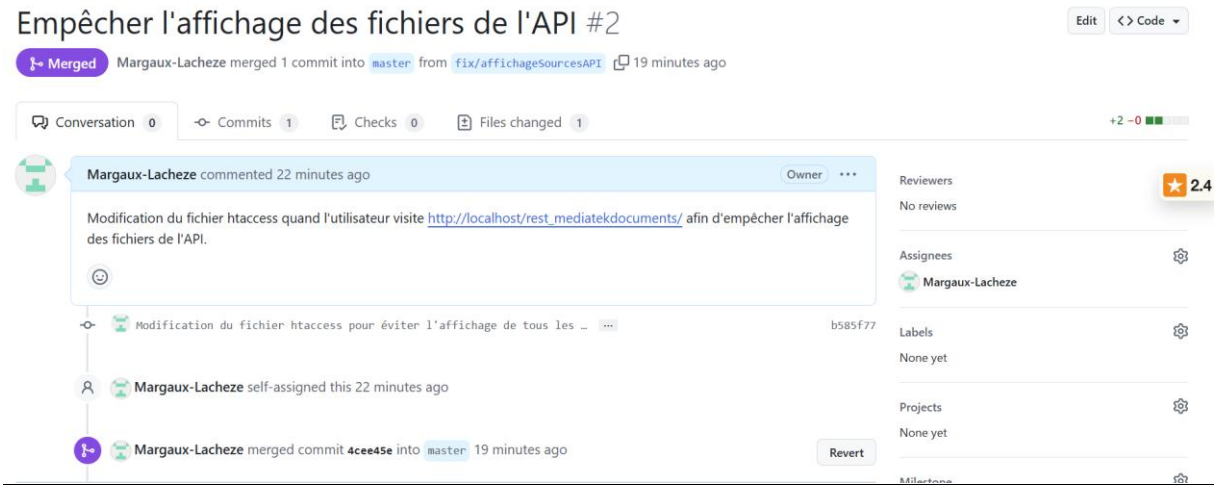


Pull requests

N°1 : Dépôt Mediatekdocuments



N°2 : Dépôt rest_mediatekdocuments



Explications sur le code

Application Mediatekdocuments :

J'ai mis le login et le mot de passe de l'API dans le fichier App.Config. Dans le constructeur de la classe Access, ceux-ci sont récupérés et stockés dans les variables login et password en appelant respectivement les clefs « ApiLogin » et « ApiPassword » du fichier de configuration. Ensuite la logique

conserve l'ancienne version avec une chaîne de connexion qui contient cette fois-ci la concaténation des variables login et password. Ensuite la méthode GetInstance de la classe ApiRest est toujours appelée.

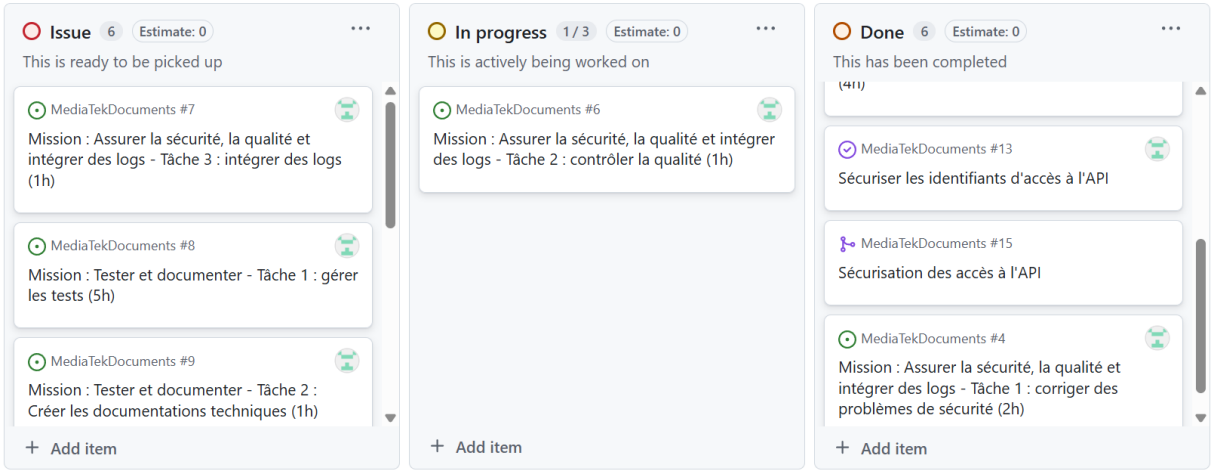
API :

Dans le fichier htaccess, j'ai ajouté une règle qui empêche l'affichage des fichiers. Lorsqu'un utilisateur tape http://localhost/rest_mediatekdocuments/, il tombe dorénavant sur une erreur 400.

Tâche 2 : contrôler la qualité

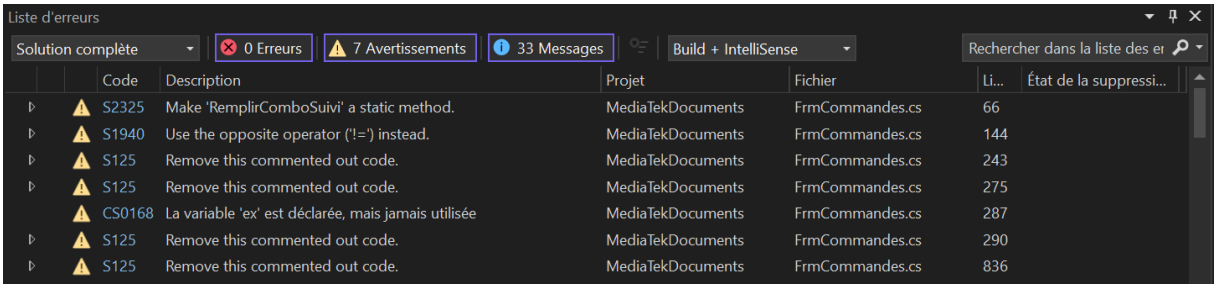
Le code d'origine a été nettoyé en suivant les conseils de Sonarlint/SonarQube et des avertissements de Visual Studio. Le seul point qui n'a pas été corrigé est le nom des méthodes événementielles, qui commencent par une minuscule car les objets graphiques commencent par une minuscule. Excepté ce point, les règles d'écriture de code sont les règles standards. Le but est donc de nettoyer le code ajouté en suivant la même logique. Il faut aussi transférer dabs App.config, les informations d'authentification pour l'accès à l'API en ligne (qui se trouvent actuellement en clair dans Access).

Kanban



Temps estimé : 1h00 Temps réel : 30 minutes

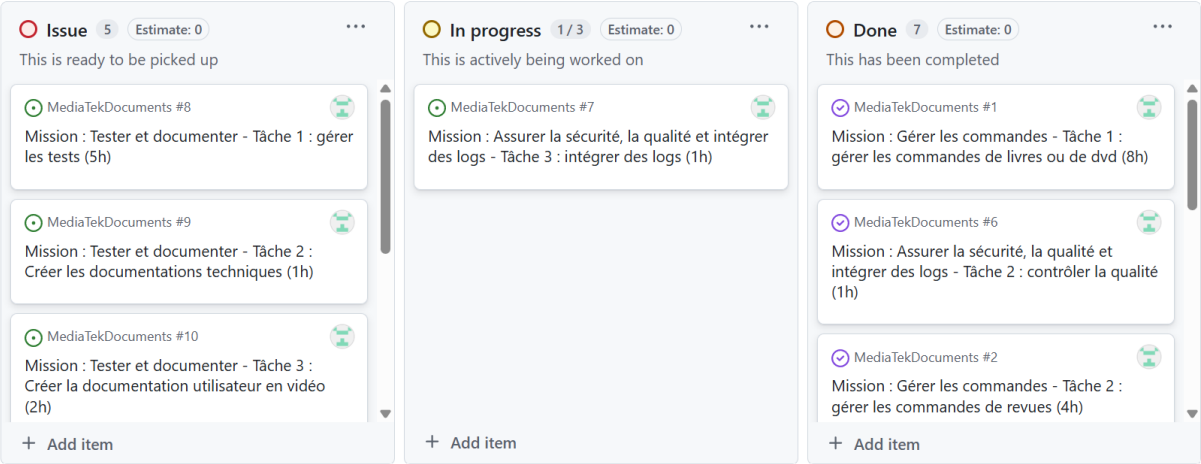
Extrait de la liste des avertissements SonarLint



Tâche 3 : Intégrer des logs

Dans la classe Access, ajouter le code de configuration des logs et des logs au niveau de chaque affichage console (à enregistrer dans un fichier de logs).

Kanban



Explications sur le code

Code permettant de configurer les logs :

Dans le constructeur de la classe Access, en utilisant Serilog j'ai créé un logger. J'ai configuré trois sorties. Une sortie console et une sortie fichier « log.txt » dont le level des logs est verbose. Un nouveau fichier horodaté est créé chaque jour. Les logs, à partir du niveau information, sont enregistrés de la même façon dans un fichier « errorlog.txt ». Les fichiers sont trouvables dans le dossier bin/debug/log de l'application.

Exemple de code d'ajout d'un log :

```
public bool CreerExemplaire(Exemplaire exemplaire)
{
    String jsonExemplaire = JsonConvert.SerializeObject(exemplaire, new CustomDateTimeConverter());
    try
    {
        List<Exemplaire> liste = TraitementRecup<Exemplaire>(POST, "exemplaire", "champs=" + jsonExemplaire);
        return (liste != null);
    }
    catch (Exception ex)
    {
        Log.Error(ex, "Access.CreerExemplaire - Erreur lors de la création d'un exemplaire: {0}", ex.Message);
        Console.WriteLine(ex.Message);
    }
    return false;
}
```

Si un problème survient lors de la création d'un exemplaire, ce log sera à la fois dans la console ainsi que dans les fichiers de logs.

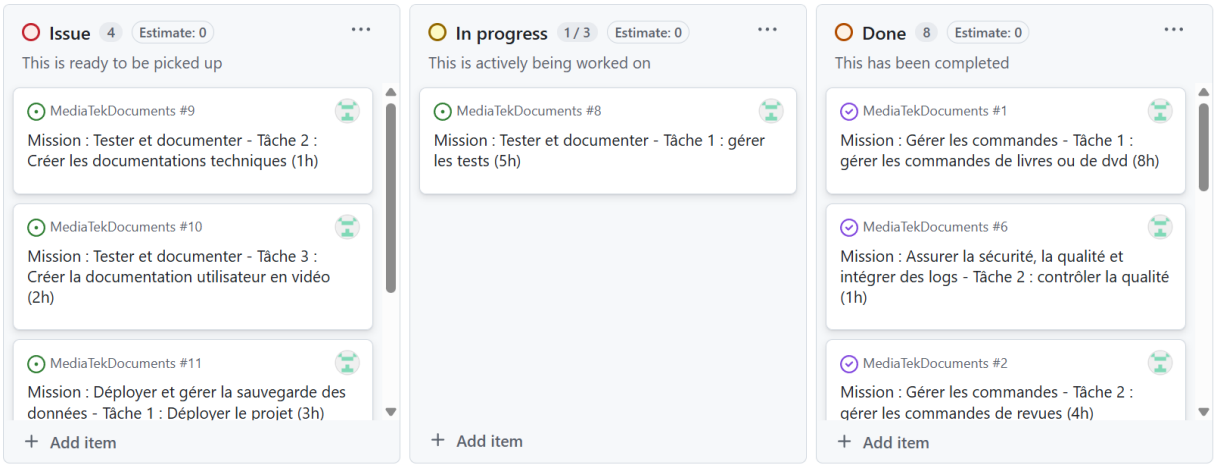
Mission : Tester et documenter

Tâche 1 : Gérer les tests

Écrire les tests unitaires sur les classes du package Model (en plus du test unitaire écrit précédemment).

Construire une collection de tests dans Postman pour contrôler les fonctionnalités de l'API d'accès à la BDD.

Kanban



Temps estimé : 5h00 Temps réel : 4h00

Exemple de code de test unitaire C#

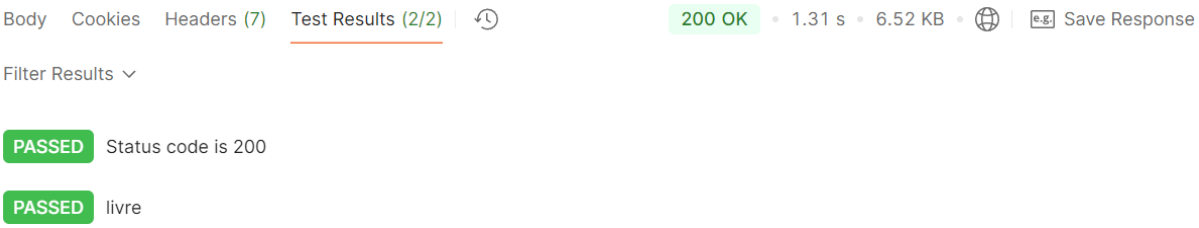
```
[TestInitialize]
0 références | 0 modification | 0 auteur, 0 modification
public void Initialize()
{
    id = "1";
    dateCommande = DateTime.Parse("2025-04-10");
    montant = 10.0;
    nbExemplaire = 2;
    idLivreDvd = "2";
    idSuivi = "00001";
    libelle = "commande";
    commande = new CommandeDocument(id, dateCommande, montant, nbExemplaire, idLivreDvd, idSuivi, libelle);
}

[TestMethod()]
0 références | 0 modification | 0 auteur, 0 modification
public void CommandeDocumentTest()
{
    Assert.AreEqual(idSuivi, commande.IdSuivi);
}

[TestMethod()]
0 références | 0 modification | 0 auteur, 0 modification
public void EstTransitionAutoriseeTest()
{
    // Commande en cours peut être relancée
    Assert.IsTrue(commande.EstTransitionAutorisee("00004"));
    // Commande en cours ne peut pas être tout de suite réglée
    Assert.IsFalse(commande.EstTransitionAutorisee("00003"));
}
```

Exemple de test fonctionnel sous Postman

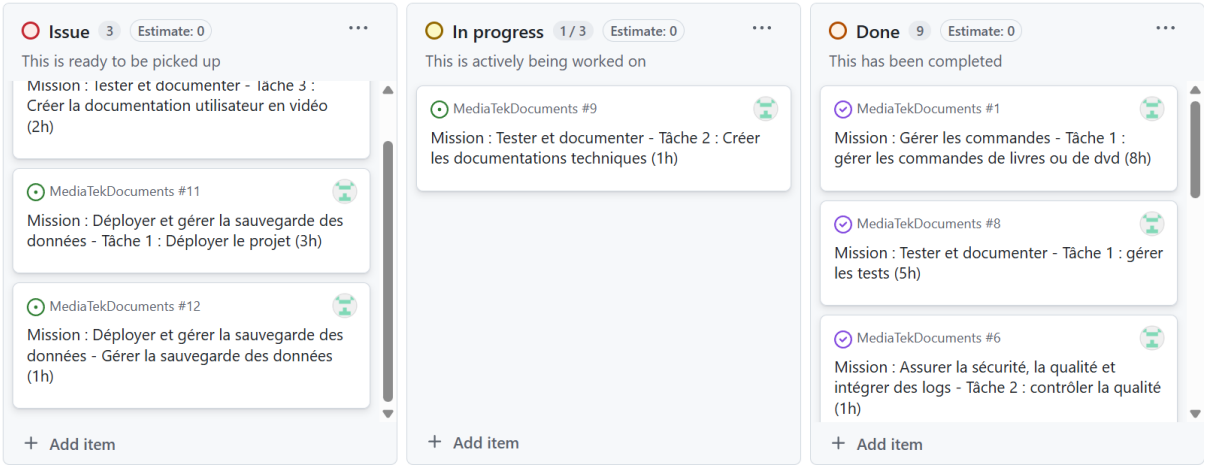
```
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
});
pm.test("livre", function () {
    var jsonData = pm.response.json();
    pm.expect(jsonData["result"][0].id).to.eql("00017");
});
```



Tâche 2 : créer les documentations techniques

- Contrôler, dans chaque application, que les commentaires normalisés sont bien tous ajoutés et corrects.
- Générer la documentation technique de l'application C#, en suivant les explications données dans l'article "Documentation technique sous Visual Studio".
- Générer la documentation technique de l'API REST, en suivant les explications données dans l'article "Génération de la documentation technique sous Netbeans".
- Transférer les documentations dans les dépôts.

Kanban



Temps estimé : 1h00 Temps réel : 1h30

Exemple de commentaire normalisé

Appli C# :

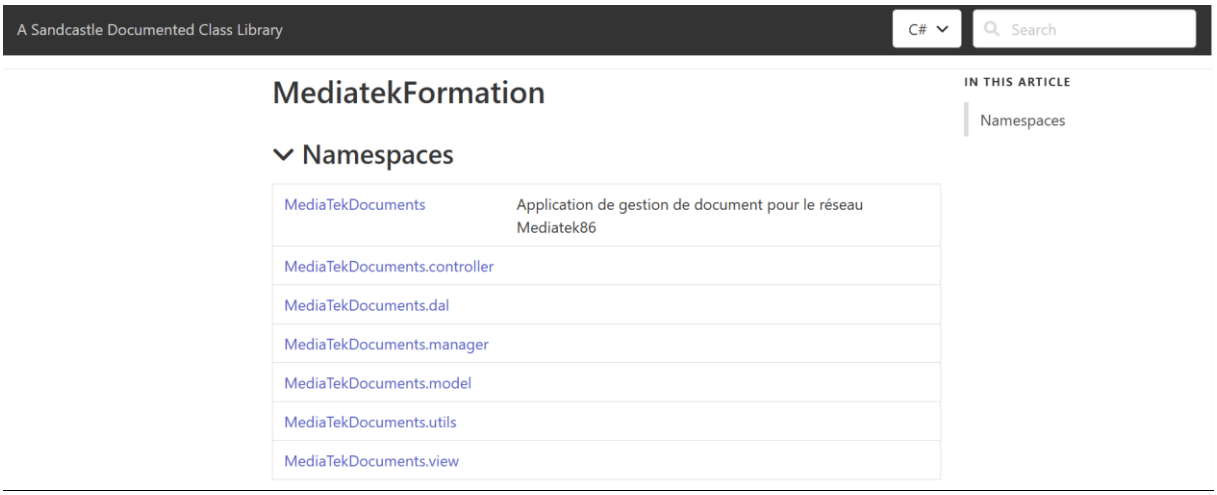
```
/// <summary>
/// Compare deux mots de passe pour vérifier leur correspondance
/// </summary>
/// <param name="inputPassword">le mot de passe saisi sans hash</param>
/// <param name="storedHash">le mot de passe stocké en base de données (hashé)</param>
/// <returns>True si les mots de passe correspondent</returns>
1 référence | Margaux Lacheze, Il y a 1 heure | 1 auteur, 2 modifications
public static bool VerifyPassword(string inputPassword, string storedHash)
```

API :

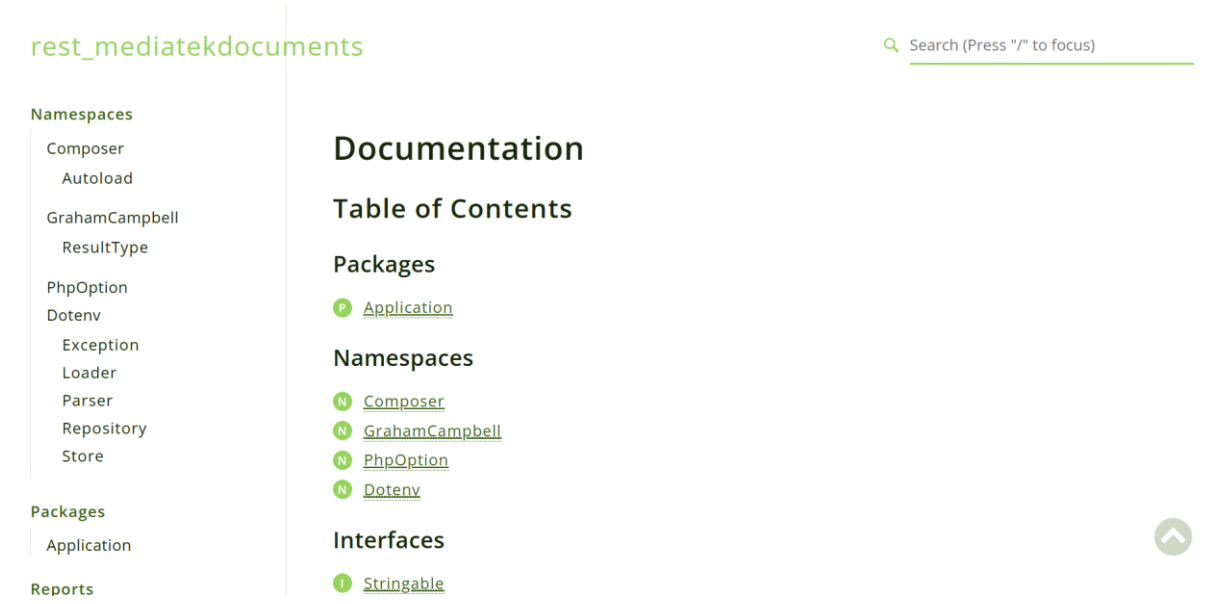
```
/**
 * Sélectionne tous les abonnements de la revue dont l'id est passée en paramètre
 * @param array|null $champs
 * @return array|null
 */
private function selectAbonnementsRevue(?array $champs) : ?array {
```

Extrait de la documentation technique

Appli C# :



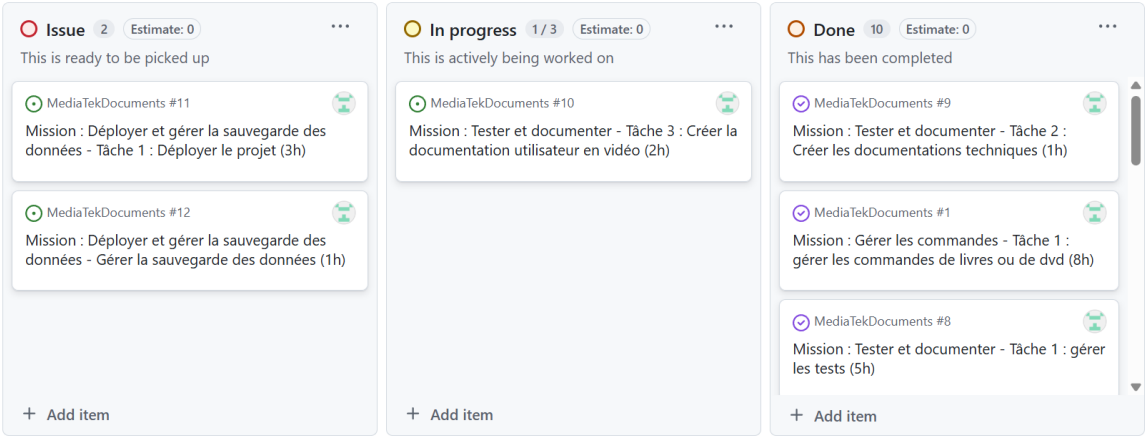
API :



Tâche 3 : Créer la documentation utilisateur en vidéo

Créer une vidéo de 10mn maximum qui présente l'ensemble des fonctionnalités de l'application C#.

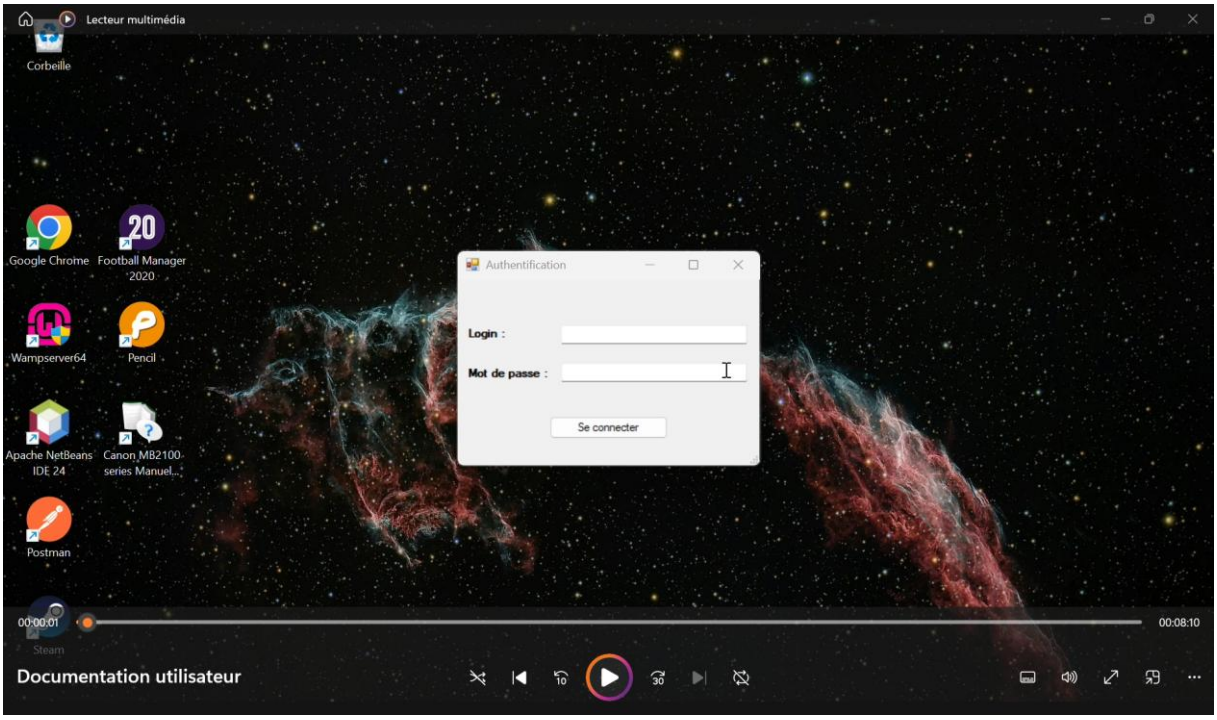
Kanban



Temps estimé : 2h00 Temps réel : 1h30

Informations

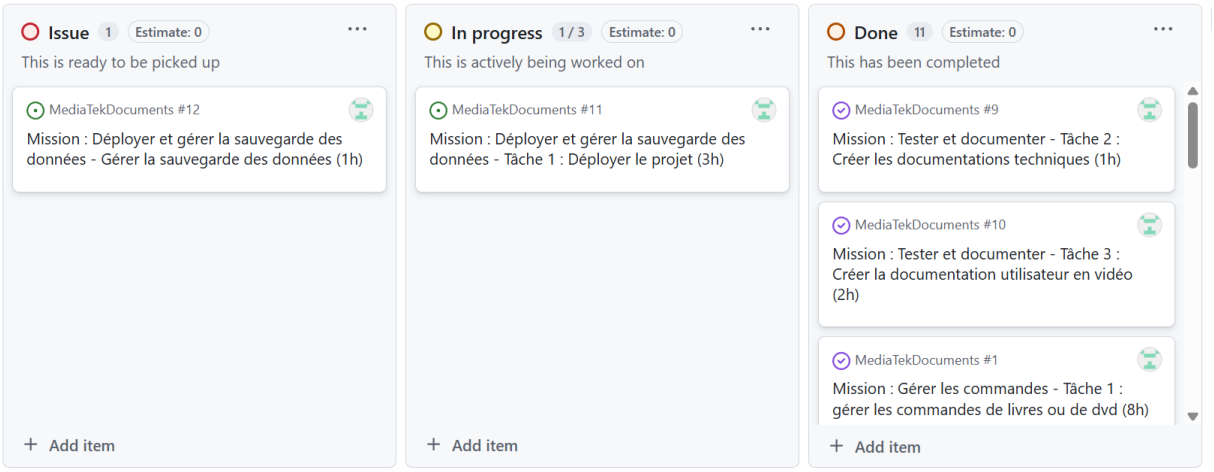
La vidéo a été faite grâce à l'outil de capture d'écran windows. Afin de la fluidifier, quelques cuts ont été effectués grâce à iMovie. Elle dure 8min10.



Mission : déployer et gérer la sauvegarde des données

Tâche 1 : Déployer le projet

Kanban



Temps estimé : 3h00 Temps réel : 6h30

Explications sur les étapes du déploiement

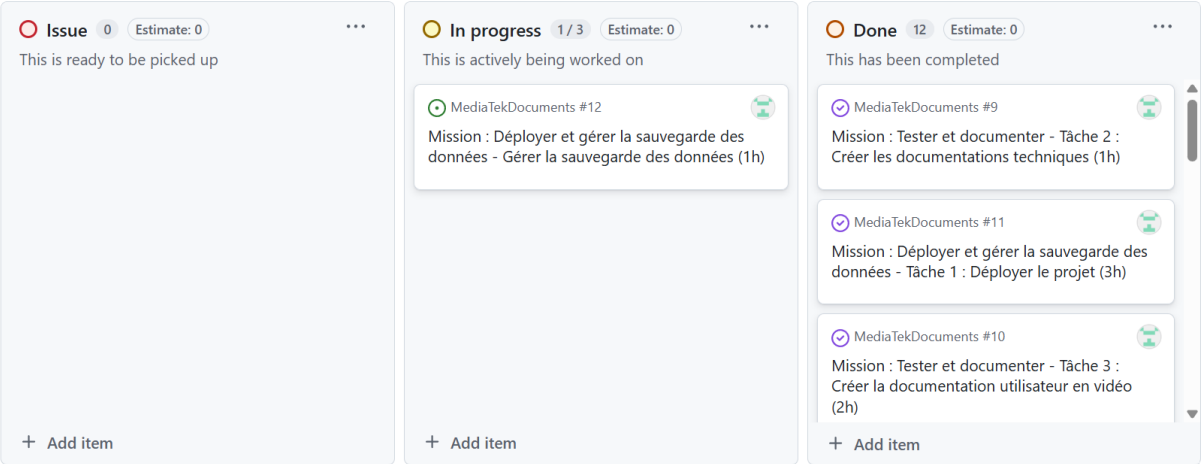
Sur Azure j'ai créé une machine virtuelle mediatek-vm avec une version Windows Server 2019 et avec les ports 80 et 3389 ouverts. En me connectant via le bureau à distance, j'ai installé wamp server. J'ai pu copier mon API et migrer ma base de données pour laquelle j'ai ajouté un utilisateur. J'ai modifié les informations de connexion de l'API dans le fichier .env.

Les tests Postman ont montré un problème avec la méthode PUT. Après de nombreux logs, j'ai pu voir qu'il s'agissait d'un mauvais encodage de ma base de données. Certaines tables étaient encodées en utf8mb4_900_ai_ci et d'autres non. Mes procédures stockées utilisant une comparaison entre deux tables encodées différemment, cela créait une requête invalide bien que le code serveur était 200. J'ai donc harmonisé les tables au niveau de l'encodage, et forcé ma procédure stockée à utiliser utf8mb4_900_ai_ci. Suite à cela, l'API était totalement fonctionnelle en ligne.

J'ai ensuite créé un installeur C# en utilisant l'extension Microsoft Visual Studio Installer Projects 2022 et Setup Wizard. J'ai créé une icône sur Canva qui reprend la direction artistique de Mediatek86. Suite à cela, j'ai pu avoir le fichier .msi et ai installé l'application sur mon propre ordinateur. Après avoir vérifié que tout fonctionnait correctement, j'ai envoyé le fichier .msi sur mon dépôt Github.

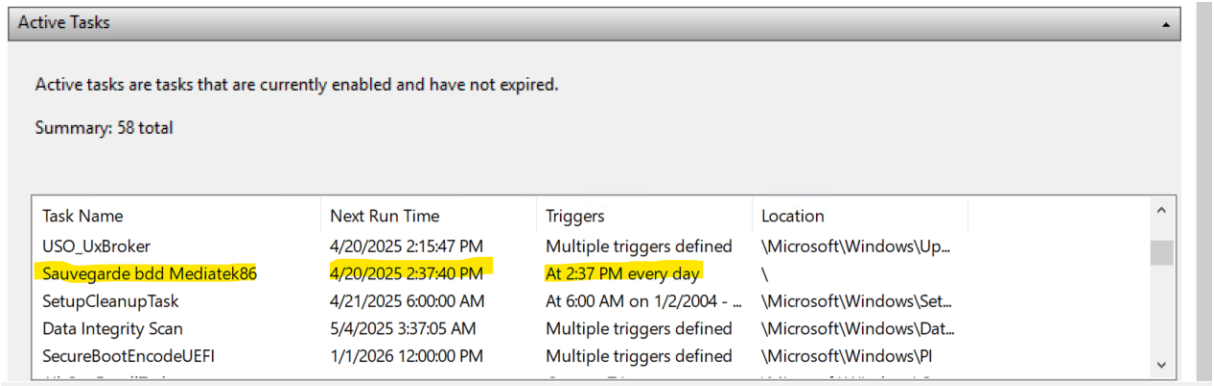
Tâche 2 : Gérer les sauvegardes de données

Kanban



Temps estimé : 1h00 Temps réel : 30 minutes

Capture d'écran de l'enregistrement de la sauvegarde



Explications

J'ai créé un fichier savebdd.bat qui exécute chaque jour mysqldump.exe et qui enregistre le contenu de la base de données dans une fichier .sql horodaté, sauvegardé dans un dossier savebdd. Pour que ce fichier soit automatiquement exécuté, une tâche automatisée windows a été programmée grâce au planificateur de tâches. Elle exécute quotidiennement de façon automatique le fichier .bat

Mode opératoire pour la restauration

Pour restaurer la base de données, il suffit d'aller dans le dossier savebdd et de choisir la date à laquelle vous souhaitez récupérer la base de données pour sélectionner le bon fichier sql.

Il faut ensuite créer une base de données nommée mediatek86 dans phpmyadmin puis soit importer le fichier .sql choisi, soit exécuter les requêtes qu'il contient (en l'ouvrant préalablement dans le bloc note par exemple).

Bilan final

Objectifs atteints

Gestion des commandes :

- commandes de livres et DVD (commander un ou plusieurs exemplaires ; gérer les stades de la commande : "en cours", "relancée", "livrée, "réglée" ; afficher la liste des commandes et leur stade ; lors de la livraison, générer les exemplaires)
- commandes de revues (correspond à un abonnement ou un renouvellement d'abonnement).

Limitation des accès :

- l'administrateur et le personnel du service administratif ont accès à toutes les fonctionnalités
- les employés du services "prêts" n'ont accès qu'aux fonctionnalités de consultation des documents
- les employés du service "culture" (organisation des événements) ne sont pas censés manipuler cette application.

Tests :

- Tests unitaires sur les classes du model
- Tests fonctionnels de l'API sur Postman

Documentation :

- Documentation technique pour l'application et pour l'API
- Documentation utilisateur en vidéo

Déploiement :

- L'API et la base de données sont en lignes
- L'application C# est déployée, fonctionnelle et peut être installée sur n'importe quel ordinateur Windows

Sauvegarde des données :

- Une sauvegarde journalière de la base de données est programmée par le planificateur de tâches Windows

Difficultés rencontrées

J'ai eu beaucoup de mal à développer cette application et je dois dire que je ne suis pas très contente du résultat. En soi elle marche, les objectifs sont atteints mais je trouve que le code que je fournis n'est pas propre. Je pense qu'avec un peu plus de temps, j'aurais pu créer quelque chose de plus maintenable (moins de répétitions de code, MVC mieux respecté). Il y a eu de nombreux moments où j'ai douté de mes choix. D'ailleurs si l'on regarde les commits sur mon dépôt GitHub, on peut voir que j'ai refactorisé certaines choses juste avant la mission sur les tests unitaires. Je ne suis toujours pas satisfaite et je pense que certaines méthodes présentes dans Access ou dans la vue auraient dû aller dans les classes du Model. Par manque de temps avant l'échéance du BTS, je ne vais

pas pouvoir le faire mais ce serait intéressant que je me penche dessus plus tard pour proposer une version améliorée.

J'ai trouvé qu'il y avait une marche assez haute entre ce que nous avons fait en cours et cet atelier. Cela se voit dans le temps réel que j'ai mis à développer l'application. Néanmoins, j'ai beaucoup appris.

Le déploiement de l'application a été semé d'embûches. J'ai d'abord choisi de la déployer l'API dans une VM Azure sous Windows 10 pro. Néanmoins j'ai eu beaucoup de lags qui rendaient les manipulations difficiles. Nous n'avions pas déployé d'API en cours et je ne suis pas particulièrement à l'aise avec les machines virtuelles (au moment de choisir leur configuration). En outre, le guide que l'on nous a fourni (https://github.com/CNED-SLAM/rest_mediatekdocuments/wiki/API-en-ligne) comportait plusieurs solutions mais qui n'ont pas nécessairement été testées avant. Je me suis donc tournée vers un hébergeur payant, chez lequel j'ai déjà un compte pour l'autre atelier professionnel. J'avais créé un sous domaine, migré la base de données et l'api. Je rencontrais des erreurs que je n'arrivais pas à débuguer. Je suis d'abord partie sur des problèmes avec le htaccess et les verbes http car seule la méthode GET marchait. J'ai pris vraiment beaucoup de temps à essayer de trouver une solution et comprendre le pourquoi du comment. Sans y arriver. Vu que nous avons des échéances avec l'examen, j'ai choisi une autre solution mais j'ai tout gardé sur l'hébergeur pour pouvoir m'y replonger plus tard.

J'ai rapidement envisagé des solutions qui n'étaient pas proposées par nos professeurs (comme render), mais j'ai finalement décidé de ne pas abandonner Azure et ses VM. Comme j'ai mis mon premier atelier professionnel chez un hébergeur, je trouvais ça plus intéressant d'essayer quand même la machine virtuelle pour l'API. Cette fois j'ai pris une version Windows Server et un peu plus de Ram. L'API n'avait pas terminé de me donner du fil à retordre car tout marchait sauf ma méthode PUT qui utilise une procédure stockée. Après beaucoup de logs, j'ai enfin pu découvrir que j'avais mal encodé ma base de données. Une partie était en utf8mb4_900_ai_ci et l'autre non. Il se peut que le problème ait été là depuis le début. En résumé ce fut vraiment fastidieux, mais on ne me prendra plus à ne pas faire attention à l'encodage de ma base de données.

Par rapport à la gestion du temps, j'ai mis 7h30 de plus que ce qui était prévu. Les écarts sont plus grands sur les tâches de développement pur et sur la tâche de déploiement de l'application. En revanche, j'ai été plus rapide sur les tests. On peut expliquer cela par les raisons données plus haut. Sur la phase de développement, je suis régulièrement revenue en arrière pour améliorer le code. Et j'ai eu beaucoup de mal à mettre l'application en ligne. Au-delà de cela, je passe le BTS avec des soucis de santé et bénéficie d'un tiers-temps pour les épreuves. Au cours de l'année, cela s'est traduit par l'obligation de m'arrêter en plein milieu de certaines missions et de devoir ensuite retrouver le fil. La gestion des contrôles dans la vue de l'application C# demandant une certaine attention, on peut très vite se perdre et avoir des comportements inattendus qui deviennent vite chronophages.