BTS SIO 2023

☐ **Administration des systèmes et des réseaux (E5 – SISR)**

ou

■ **Conception et développement d'applications (E5 – SLAM)**

PAGE DE PRÉSENTATION DU DOSSIER

N° d'inscription[1] : |0_|1_|9_|4_|7_|0_|8_|2_|9_|6_||9_|

NOM : DESCHAMPT

Prénom : Thomas

Date de passage[1] : 31/05/2023	**Heure de passage[1] :** 12**h**00

CATEGORIE CANDIDAT [2] (UNE CASE A COCHER)	
■ Scolaire ❑ Apprenti ❑ Formation professionnelle continue ❑ Expérience professionnelle 3 ans	❑ Ex-scolaire ❑ Ex-apprenti ❑ Ex-formation professionnelle continue

[1] *Informations communiquées sur votre convocation envoyée en mars-avril 2022*

[2] *Informations communiquées sur votre confirmation d'inscription*

Tampon de l'établissement

BTS SERVICES INFORMATIQUES AUX ORGANISATIONS	SESSION 2023
Épreuve E5 - Conception et développement d'applications (option SLAM)	
ANNEXE 7-1-B : Fiche descriptive de réalisation professionnelle (recto)	

DESCRIPTION D'UNE RÉALISATION PROFESSIONNELLE	**N° réalisation : 01**
Nom, prénom : Deschampt Thomas	**N° candidat :** 01947082969
Épreuve ponctuelle ☒ **Contrôle en cours de formation** ☐	**Date :** 31/05/2023

Organisation support de la réalisation professionnelle
Laboratoire GSB

Intitulé de la réalisation professionnelle
PROJET Service Web

Période de réalisation : 15/03/2023 au 25/04/2023 **Lieu :** EPSI Lyon
Modalité : ☒ **Seul(e)** ☐ **En équipe**

Compétences travaillées

☒ Concevoir et développer une solution applicative
☐ Assurer la maintenance corrective ou évolutive d'une solution applicative
☒ Gérer les données

Conditions de réalisation[5] (ressources fournies, résultats attendus)

-Ressources fournies : description du contexte, expression des besoins, exemple de requête

-Résultat attendus : création, conception et remplissage d'une base de données - développement d'une application Back-end appelant la base de données en utilisant la norme RESTFULL - sécurisation des API via un token JWT

Description des ressources documentaires, matérielles et logicielles utilisées[6]

Modélisation : Looping
SGBD : SQL Server et SQL Management Studio
Environnement de développement : Visual Studio
Bibliothèque de développement : .NET, Entity Framework (ORM)
Langages : SQL, C#, TransactSQL
Gestion de version : Github
Tests des comportement API : Talend API Tester

Modalités d'accès aux productions[7] et à leur documentation[8]

Base de données et application Backend hébergées en local via Visual Studio
https://github.com/ThomasDeschampt/BTS_PPE

- Le répertoire Back_PPE_SLAM comporte tous les contrôleurs et donc les API
- Le répertoire Model au modèle qui décrit les tables de la base de données
- Le répertoire ORM_PPE_SLAM qui fait le lien entre la base de données et le Backend
- Le répertoire BDD comporte le script pour créer la base de données et le fichier Looping

[5] En référence aux *conditions de réalisation et ressources nécessaires* du bloc « Conception et développement d'applications » prévues dans le référentiel de certification du BTS SIO.

[6] Les réalisations professionnelles sont élaborées dans un environnement technologique conforme à l'annexe II.E du référentiel du BTS SIO.

[7] Conformément au référentiel du BTS SIO « *Dans tous les cas, les candidats doivent se munir des outils et ressources techniques nécessaires au déroulement de l'épreuve. Ils sont seuls responsables de la disponibilité et de la mise en œuvre de ces outils et ressources. La circulaire nationale d'organisation précise les conditions matérielles de déroulement des interrogations et les pénalités à appliquer aux candidats qui ne se seraient pas munis des éléments nécessaires au déroulement de l'épreuve.* ». Les éléments peuvent être un identifiant, un mot de passe, une adresse réticulaire (URL) d'un espace de stockage et de la présentation de l'organisation du stockage.

[8] Lien vers la documentation complète, précisant et décrivant, si cela n'a été fait au verso de la fiche, la réalisation professionnelle, par exemples service fourni par la réalisation, interfaces utilisateurs, description des classes ou de la base de données.

Contexte et besoins :

Dans le cadre de ses activités, le laboratoire Galaxy Swiss Bourdin a pour habitude de travailler avec des médecins implantés dans différents départements. Ainsi, le GSB utilise une base de données comportant tous les médecins pour faciliter ses différents travaux. Cette base est vouée à évoluer en permanence compte tenu de l'arrivée de nouveaux médecins et du départ en retraite de certains d'entre eux mais aussi à cause des déplacements de médecins vers de nouveaux départements.

Pour répondre à ses besoins, le GSB a contacté une entreprise de services informatiques qui sera chargée de fournir au laboratoire un Service Web et une nouvelle base répondant à tous leurs besoins et contraintes.

Solution à mettre en place :

Pour satisfaire les attentes du GSB, le Web Service (que l'on appellera désormais WS) va devoir respecter un certain nombre de contraintes et prérequis.

Tout d'abord, il faudra mettre en place une base de données MySQL qui regroupe différentes informations relatives aux médecins telles que leur nom, leur prénom, leur spécialité, leur département, etc…

Pour le WS, son fonctionnement repose sur la technologie .NET version 4.8 ainsi que le Framework Entity Framework version 6.2.0, accompagné d'un ORM (*Object-Relational Mapping*). Les requêtes de récupération et envoi de données vers la base se feront par le biais d'API (*Application Programming Interface*) qui utilise le header HTTPS.

L'application permettra aussi de retrouver une liste de médecins en fonction d'un département ou bien en fonction de son nom ou de sa spécialité.

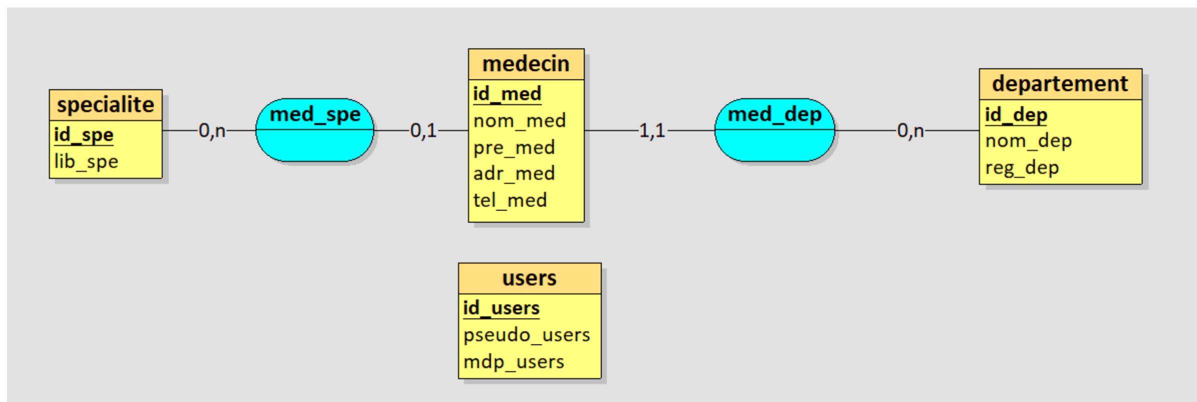
Enfin, du côté de la sécurisation, l'utilisation des API requiert d'avoir un Token valide qui ne pourra seulement être fourni aux utilisateurs valides, sinon l'utilisateur sera renvoyé vers une erreur 403 “Accès refusé”.

Réalisations :

Base de données :

Avant de commencer à travailler sur le Web Service en lui-même, il a fallu commencer par la conception et la création de la base de données que l'on hébergera ensuite sur SQL Server.

La première étape est donc de réaliser le modèle conceptuel des données de notre base (le MCD). Pour cela, on va utiliser le logiciel Looping qui nous permet non seulement de conceptualiser une base de données mais aussi de générer un script SQL à partir de notre MCD, qui nous servira ensuite à créer la base.

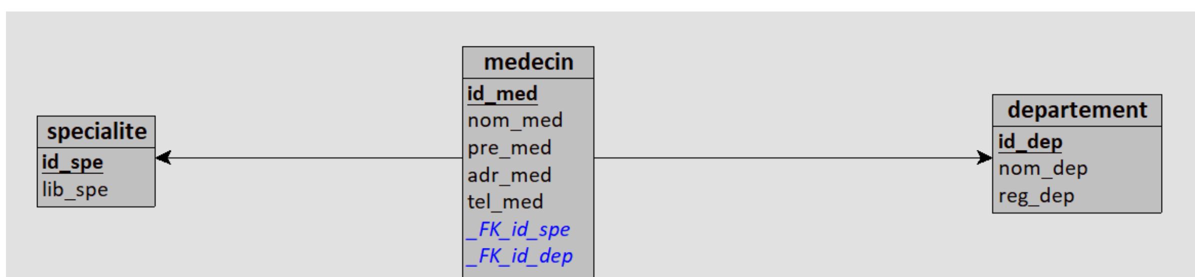


Ainsi, dans le MCD que j'ai réalisé, on retrouve 4 tables qui réunissent l'ensemble des informations demandés par le GSB. La table "medecin" regroupe toutes les informations "uniques" à un médecin c'est-à-dire son nom, prénom, adresse et téléphone ainsi que son identifiant auto incrémenté qui fera office de clé primaire dans la table.

Cette table est en relation avec deux autres tables qui comportent quant à elles des informations qui peuvent être communes à plusieurs médecins telles que le département et la spécialité.
L'intérêt est de pouvoir faire appels aux données d'autres tables et donc éviter les redondances, réduire la taille de la base de données tout en facilitant la gestion des données.

Tout d'abord, la table "spécialité" qui correspond à la spécialité complémentaire que certains médecins généralistes possèdent. Etant donné que tous les médecins n'ont pas obligatoirement de spécialités, la cardinalité qui relie medecin à specialite est de 0,1 (un médecin peut avoir une spécialité ou aucune spécialité).
Ensuite la table "departement" qui correspond au département dans lequel le médecin exerce son activité. Un médecin ne peut avoir qu'un seul département d'où la cardinalité 1,1.
Dans les deux tables, les cardinalités qui les relient vers la table medecin sont de 0,n car un département et une spécialité peuvent aussi bien être associés à un seul médecin, qu'à plusieurs médecins ou même à aucun.

Enfin, la table users nous permettra de stocker les utilisateurs (et leur mot de passe) que nous utiliserons plus tard lors de la sécurisation des API via un Token.



Si on visualise notre base en mode MLD (Modèle Logique des Données), on voit que les relations ont disparu au profit de l'ajout de deux clés étrangères dans la table medecin : _FK_id_spe et _FK_id_dep correspondant respectivement à la spécialité et au département.

```sql
CREATE TABLE departement(
  id_dep INT IDENTITY,
  nom_dep VARCHAR(250) NOT NULL,
  reg_dep VARCHAR(250),
  PRIMARY KEY(id_dep)
);

CREATE TABLE specialite(
  id_spe INT IDENTITY,
  lib_spe VARCHAR(100) NOT NULL,
  PRIMARY KEY(id_spe)
);

CREATE TABLE medecin(
  id_med INT IDENTITY,
  nom_med VARCHAR(50) NOT NULL,
  pre_med VARCHAR(50) NOT NULL,
  adr_med VARCHAR(250) NOT NULL,
  tel_med VARCHAR(50) NOT NULL,
  _FK_id_spe INT,
  _FK_id_dep INT NOT NULL,
  PRIMARY KEY(id_med),
  FOREIGN KEY(_FK_id_spe) REFERENCES specialite(id_spe),
  FOREIGN KEY(_FK_id_dep) REFERENCES departement(id_dep)
);
```

La dernière étape que j'ai eu à réaliser dans Looping a été de générer le script SQL associé au MCD. Pour ce faire, il faut simplement préciser dans quel type de SGBD (Système de Gestion de Base de Données) on souhaite utiliser le script, ici SQL Server.
Après, le logiciel nous ressort automatiquement un script qui correspondant au modèle conceptuel des données que l'on vient de mettre en place.
Avant de copier ce code dans notre SGBD, j'ai juste rajouté (1,1) après les IDENTITY pour préciser que l'auto-incrémentation se fera de 1 en 1.

<u>Web Service :</u>

Pour le WS, j'ai travaillé sur le logiciel Visual Studio en utilisant la technologie .NET. La première étape a été de créer un projet web ASP.NET ; ce projet constitue le Backend de notre application. C'est donc depuis ce projet que l'on peut faire appel à nos API. Ainsi, on va tout d'abord commencer par créer les 4 contrôleurs correspondant à nos tables "medecin, specialite, departement et users".

Une fois les contrôleurs créer, on retrouve dans ceux-ci toutes les actions que l'on peut être amené à effectuer sur nos données, c'est-à-dire : les afficher, les modifier, les supprimer ou en ajouter des nouvelles (chacune de ces actions correspondant respectivement aux opérations du CRUD avec le protocol HTTP : GET, PUT, DELETE, POST).

Dans un second temps, j'ai créé un ORM (*Object-Relational Mapping*) reposant sur le Framework Entity Framework c'est lui qui fera le lien entre la base de données et les contrôleurs. Ainsi l'ORM nous permettra de récupérer les données de la base directement via des objets et donc de ne pas avoir besoin de passer par des requêtes SQL.
L'ORM est composé d'un fichier principal data_model.cs :

```csharp
public virtual DbSet<departement> departements { get; set; }
7 références
public virtual DbSet<medecin> medecins { get; set; }
6 références
public virtual DbSet<specialite> specialites { get; set; }
7 références
public virtual DbSet<user> users { get; set; }
```

Au début du fichier, chaque table est associée à un dbset (classe d'Entity Framework qui permet de représenter et de manipuler une table par le biais d'un objet).

```
{
    modelBuilder.Entity<departement>()
        .Property(e => e.nom_dep)
        .IsUnicode(false);

    modelBuilder.Entity<departement>()
        .Property(e => e.reg_dep)
        .IsUnicode(false);

    modelBuilder.Entity<departement>()
        .HasMany(e => e.medecins)
        .WithRequired(e => e.departement)
        .HasForeignKey(e => e.C_FK_id_dep)
        .WillCascadeOnDelete(false);

    modelBuilder.Entity<medecin>()
        .Property(e => e.nom_med)
        .IsUnicode(false);

    modelBuilder.Entity<medecin>()
        .Property(e => e.pre_med)
        .IsUnicode(false);

    modelBuilder.Entity<medecin>()
        .Property(e => e.adr_med)
        .IsUnicode(false);

    modelBuilder.Entity<medecin>()
        .Property(e => e.tel_med)
        .IsUnicode(false);

    modelBuilder.Entity<specialite>()
        .Property(e => e.lib_spe)
        .IsUnicode(false);

    modelBuilder.Entity<specialite>()
        .HasMany(e => e.medecins)
        .WithOptional(e => e.specialite)
        .HasForeignKey(e => e.C_FK_id_spe);
}
```

La suite du fichier va renseigner les propriétés de chacune de nos entités, mais aussi les propriétés des relations qui mettent en lien nos entités.

Par exemple, pour la table “departement”, on va configurer nos champs nom_dep et reg_dep de tel sorte à ce qu’ils n’acceptent pas les caractères Unicode. Ensuite, dans le 3ème bloc, on va configurer la relation entre departement et medecin : un département peut avoir plusieurs médecins associés mais un médecin ne peut avoir qu’un seul département ("many-to-one").
On définit la clé étrangère C_FK_id_dep pour la relation entre ces deux entités puis on indique que la suppression en cascade n’est pas activée entre les deux entités (la suppression d’un département n'entraînera donc pas la suppression des médecins qui lui sont associés).

Enfin pour finaliser la structure du projet, le dernier élément que j’ai rajouté est le modèle. C’est lui qui va représenter la structure des données et qui fera ainsi la liaison entre l’ORM et le Backend. Ainsi, le modèle sera composé de 4 fichiers, un pour chaque entité.

Le code présent à droite est celui du fichier représentant l’entité medecin correspondant à la table du même nom.
Dans le fichier on va décrire toutes les propriétés de la classe medecin c’est-à-dire toutes les colonnes qui composent notre table.
On définit ici le type de données (string, int, …) des colonnes, leur longueur maximale et si les champs sont obligatoires ou non.
[Key] permet de préciser que le champ est la clé primaire de la table alors que [Column] précise que le champ correspondant à une clé étrangère de la table.
À la fin du code, on va ajouter deux nouvelles propriétés qui ne renseignent pas de colonnes mais indiquent une relation avec une autre entité. Cela va permettre d’accéder à l'enregistrement correspondant dans les tables spécialités et départements.

```
[Table("medecin")]
24 références
public partial class medecin
{
    [Key]
    3 références
    public int id_med { get; set; }

    [Required]
    [StringLength(50)]
    1 référence
    public string nom_med { get; set; }

    [Required]
    [StringLength(50)]
    1 référence
    public string pre_med { get; set; }

    [Required]
    [StringLength(250)]
    1 référence
    public string adr_med { get; set; }

    [Required]
    [StringLength(50)]
    1 référence
    public string tel_med { get; set; }

    [Column("_FK_id_spe")]
    4 références
    public int? C_FK_id_spe { get; set; }

    [Column("_FK_id_dep")]
    4 références
    public int C_FK_id_dep { get; set; }

    1 référence
    public virtual departement departement { get; set; }

    1 référence
    public virtual specialite specialite { get; set; }
}
```

Pour synthétiser, dans un premier temps l'ORM fait le lien entre la BDD et l'application, il représente la base de données et la structure des tables et les relations qui la composent. C'est lui qui permet les interactions avec la base.
Dans un second temps, le modèle va quant à lui décrire les tables en elles-mêmes, en détaillant leurs différents champs. Pour finir, les contrôleurs du Backend vont récupérer les données en s'appuyant sur le modèle et les faire transiter par le biais des API.

Avant de passer à la sécurisation du WS, j'ai mis en place la méthode qui permettra de récupérer une liste de médecin en fonction de leur nom. Avant toute chose, j'ai déclaré la propriété "nom" qui servira lors de la recherche des correspondances depuis le fichier medecin du modèle.

Dans le contrôleur médecin du backend, j'ai créé une nouvelle API qui va prendre en paramètre la variable "nom" de type string. Le rôle de l'API est de récupérer tous les médecins qui auront une correspondance dans leur nom avec le nom pris en paramètre. De plus, j'ai ajouté plusieurs vérifications qui renverront l'utilisateur vers une erreur bad Request si le nom recherché ne comporte qu'un seul caractère et vers une erreur NotFound si aucune correspondance n'est trouvée dans la table.

```
var medecins = db.medecins
    .Where(m => m.nom_med.Contains(nom))
    .ToList();
```

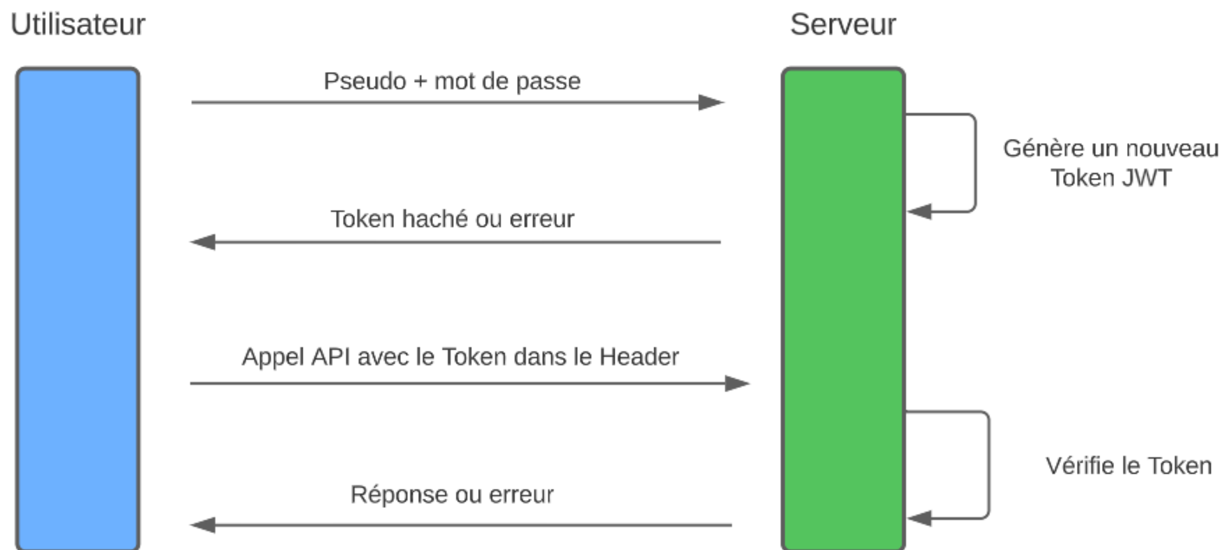
Sécurisation :

Enfin, la dernière étape que j'ai mise en place lors de ce projet a été la sécurisation de l'appel des API, l'objectif est de limiter leur accès aux personnes possédant un token valide dans leur header HTTPS. Cela permettra de maximiser la sécurité de notre application de manière à protéger nos données.

Pour ce faire, j'ai utilisé un token JWT, le fonctionnement d'un tel token fonctionne de la manière suivante : tout d'abord l'utilisateur va envoyer son pseudo et son mot de passe au serveur. En réponse, ce dernier va générer et lui envoyer un token haché, si les informations sont valides. Ensuite, l'utilisateur qui dispose désormais du token va pouvoir faire appel aux API en ajoutant le token au sein de son header HTTPS ; le serveur valide finalement ou non la demande en fonction de la validité du token présent puis renvoie soit une erreur soit la réponse attendue par l'utilisateur.

De plus, pour renforcer la sécurité du Web service, le token expire automatiquement au bout d'une certaine durée.

Le schéma présent ci-dessous permet de mieux visualiser les différentes opérations qui composent le fonctionnement de la vérification grâce au token.



J'ai commencé par créer deux nouveaux fichiers au sein du modèle : token_reponse.cs et token_request.cs. L'objectif est de définir les classes que l'on va utiliser lors de la demande et lors de la réponse à la génération d'un nouveau token. On va aussi pouvoir renseigner les propriétés que l'on sera amené à utiliser au sein de ces classes.

Ainsi pour la demande de token on aura besoin d'un pseudo et d'un mot de passe alors que pour la réponse on aura justement besoin du token et de la réponse HTTP renvoyé par le serveur.

Ensuite, j'ai créé un nouveau contrôleur dans le Backend : TokenController.cs, le rôle de ce contrôleur est triple ; il va dans un premier temps vérifier si le mot de passe et le pseudo sont valides. Puis dans un second temps, il va créer un nouveau token pour finalement terminer par le stocker sur la machine du client.

Pour la vérification des informations, on va commencer par faire appel à la méthode GetUser qui permet d'associer les informations d'identification envoyées à leur correspondance dans la table users. S'il n'y a pas de correspondance, la méthode renvoie une erreur et la génération de token n'aura pas lieu.

```
private user GetUser(token_request login)
{
    user user = null;

    try
    {
        user = db.users.Where(U => U.pseudo_user.Equals(login.pseudo) && U.mdp_user.Equals(login.mdp)).FirstOrDefault();
    }
    catch (Exception e)
    {
        return null;
    }
    return user;
}
```

Mais dans le cas où les données vérifient bien tous les contrôles, alors la méthode de vérification appelle la méthode de création.

Le token que l'on va créer est auto-suffisant et contient ainsi toutes les informations nécessaires à l'authentification d'un utilisateur ; le token se compose de 3 sections : le header (qui contient le type de token et l'algorithme utilisé ici HS256), le payload (avec les informations de l'utilisateur mais aussi la date d'émission et d'expiration du token) et la signature (qui est une chaîne caractères cryptée à partir de la clé secrète connue seulement par le web service).

Ainsi la méthode de création commence par renseigner toutes les variables qui seront utilisées pour créer le token comme la date d'émission, la date d'expiration mais aussi la clé secrète et la signature.

On terminera par créer le token (avec toutes les variables qui le forment) puis par le convertir en une chaîne de caractères.

```
var token =
    (JwtSecurityToken)
        tokenHandler.CreateJwtSecurityToken(issuer: "https://localhost:44345", audience: "https://localhost:44345",
            subject: claimsIdentity, notBefore: issuedAt, expires: expires, signingCredentials: signingCredentials);
var tokenString = tokenHandler.WriteToken(token);
```

Enfin la dernière méthode du contrôleur permet de stocker le token dans fichier en local. Après avoir défini le chemin d'accès du fichier où l'on renseigne le token, on va vérifier si le fichier existe déjà pour le supprimer si c'est le cas. Par la suite on recrée le fichier puis on écrit dans ce dernier le token généré précédemment.

```
private void StoreToken(string token)
{
    string fileName = @"C:\tmp\token.txt";

    // on supprime le fichier s'il existe deja
    if (File.Exists(fileName))
        File.Delete(fileName);

    // on le cree
    using (FileStream fs = File.Create(fileName))
    {

        // on insere le token dans le fichier
        Byte[] title = new UTF8Encoding(true).GetBytes(token);
        fs.Write(title, 0, title.Length);

    }

}
```

Pour finir la sécurisation, il faut désormais vérifier si le token envoyé par l'utilisateur est valide. C'est donc au sein d'un autre contrôleur que l'on va procéder à cette vérification. Le but de ce contrôleur sera d'extraire le token du header HTTPS puis de vérifier la validité du token pour répondre positivement ou non à la demande de l'utilisateur.

Pour vérifier la validité du token, on va mettre en place de nombreuses vérifications comme : si l'audience et l'émetteur du token sont corrects mais aussi si la date d'expiration n'est pas arrivée à terme. Puis on termine par la vérification de la signature du Token, pour ce faire le serveur va extraire les informations nécessaires au calcul de la signature puis il va recréer la signature en utilisant ces informations et les données du token avec la clé secrète qu'il est le seul à connaître. Dans le cas où les deux signatures sont les mêmes alors le token est authentique, le serveur répondra ainsi à la demande de l'utilisateur.

Pour utiliser la sécurisation via token, il suffira de faire appel à la méthode Authorize dans les fichiers contrôleurs aux niveaux des API désirées avec la ligne de code présente ci-dessous. On pourra par exemple exiger le token pour une demande d'ajout, de modification ou de suppression de données (comme dans l'exemple de code) mais pas pour une demande d'affichage.

```
// DELETE: api/medecins/5
[System.Web.Http.Authorize]
```