

Bienvenue sur l’exercice Clavardons.

Il s’agit tout simplement d’un système de tchat WebSocket un peu particulier, puisqu’il n’utilise pas de base de données, l'authentification s'effectuant exclusivement via JWT, mécanisme dont nous parlerons plus loin dans ce document.

Le tchat Clavardons, lorsqu’il est complètement fonctionnel, permet :

* De se connecter avec un nom
* De pouvoir se reconnecter (après rechargement de la page) sans avoir à remettre son nom
* D’envoyer des messages à tous les autres utilisateurs connectés en temps réel
* D’éviter le spam de messages
* De voir en temps réel la liste des utilisateurs connectés au tchat
* De se déconnecter

Une image contenant capture d’écran

Description générée automatiquement

Votre mission, si vous l’acceptez, sera de rendre la partie serveur du tchat complètement fonctionnelle. Mais avant… un peu de background technique !

# Le JWT

Le JSON Web Token est un standard ouvert de jeton d’accès. Il permet à un parti de s’authentifier auprès d’une autre entité lui ayant préalablement délivré un JWT (typiquement, un serveur).

Un JWT ressemble à ça :

Une image contenant objet

Description générée automatiquement

Une image contenant capture d’écran

Description générée automatiquementUne image contenant capture d’écran

Description générée automatiquement

Un JWT est composé de 3 parties, séparées par un « . » et encodées en base64 URL-encoded. La première partie, ici en rouge, représente le « header » du token, qui indique le type du jeton, et l’algorithme de signature utilisé (nous en reparlerons).

La deuxième partie du jeton est le « payload ». Le payload est la charge utile, celle qui permettra au serveur d’identifier qui est l’utilisateur utilisant le jeton. Typiquement, on y met l’identifiant utilisateur, potentiellement un identifiant unique de jeton, ou encore une date d’expiration.

La troisième partie, la plus intéressante, est la signature. Celle-ci est générée à partir du header et du payload, grâce à une fonction de hashage. Dans notre cas, « HS256 » représente un hashage de type HMACSHA256. Pour générer une signature, le serveur a besoin d’une clé secrète, que lui seul connaît.

Une fois que le serveur a créé le JWT, il le renvoie au client. Le client peut alors s’en servir pour s’authentifier auprès du serveur. Pour cela, le serveur décode le jeton, lit les données à l’intérieur, signe ces données à l’aide de la clé secrète, et la compare à la signature du JWT. Si elle est correcte, il accepte et authentifie l’utilisateur. Si la signature est incorrecte, c’est que le JWT a été altéré. Par exemple, si un utilisateur mal intentionné change le « sub » par une autre valeur pour se faire passer pour quelqu’un d’autre, la signature ne sera pas la même, et le serveur va le détecter et le rejeter. Notre hacker ne pourra pas générer une signature valide, puisque seul le serveur connaît la clé secrète…

Grâce à ce mécanisme, pas besoin de base de données pour s’authentifier ! C’est sur ce principe qu’est basé l’authentification de Clavardons.

Vous trouverez plus d’informations sur les JWT à [https://jwt.io](https://jwt.io/)

# SignalR

Les WebSocket sont implémentées grâce à la librairie SignalR, dont vous trouverez une partie de la documentation ici : [https://docs.microsoft.com/fr-fr/aspnet/core/tutorials/signalr](https://docs.microsoft.com/fr-fr/aspnet/core/tutorials/signalr?view=aspnetcore-2.2&tabs=visual-studio)

# A vous de jouer !

Cet exercice est découpé en 3 entités :

1. Le projet **ITI.Clavardons** que vous devez compléter
2. Le projet **ITI.Clavardons.Tests**, qui contient les tests unitaires que vous devez faire passer au vert (pour la planète)
3. Le projet **app**, qui contient le client Web vous permettant de tester votre serveur. Pour le lancer, ouvrez **app/build/index.html.**Par défaut, le client tente de se connecter à *http://localhost:5000*. Si vous souhaitez changer cette valeur, par exemple pour vous connecter au serveur d’un camarade, donnez un paramètre GET *?server\_url=http://localhost:5009*.  
     
   Vous n’avez rien à faire sur cette partie

Voici la liste des tests que vous devez réussir :

#### JWT

**T1\_GenerateJWT**

Ce test attend, pour un payload et une clé secrète, que JWTFactory.Generate renvoie un JWT valide.

*Microsoft.IdentityModel.Tokens.Base64UrlEncoder* et *HMACSHA256* vous seront utiles.

**T2\_VerifyGoodJWT**

Ce test attend qu’un JWT généré par *JWTFactory.Generate* soit reconnu valide par *JWTFactory.Verify.*

**T3\_VerifyBadJWT**

Ce test attend qu’un JWT généré par *JWTFactory.Generate* puis altéré soit reconnu invalide par *JWTFactory.Verify.*

**T4\_ParseJWT**

Ce test attend que *JWTFactory.Parse* renvoie les bonnes valeurs pour un JWT donné.

#### Anti-spam

**T1\_CheckFailAfterAttempt  
T2\_CheckPassWhenWaitingEnough  
T3\_CheckFailWhenZeroMessages**

Ces tests sont assez simples, et leur fonctionnement est documenté dans la classe *AntiSpam*.

#### ChatHub

**T1\_CheckLoginWithName**

Ce test s’attend, quand on appelle *LoginWithName* pour la première connexion :

* A bien renvoyer un JWT valable, avec un Name et un Subject aléatoire
* Lorsqu’on est déjà connecté, à ne pas réaccepter un appel à *LoginWithName*

Pour cela, *Context.ConnectionId* vous sera utile pour identifier chacune des connexions (astuce : utilisez un dictionnaire pour lier chaque connexion à un état d’authentification).

**T2\_CheckLoginWithToken**

Ce test s’attend, quand on appelle *LoginWithToken* pour une authentification ultérieure, à accepter un JWT et à renvoyer les bonnes données.

**T3\_SendMessage**

Ce test s’attend, à chaque appel à *SendMessage*, à ce que tous les clients connectés reçoivent un *ReceiveMessage* avec le message envoyé.

Pour cela, Clients.Group().SendAsync vous sera utile.

**T4\_CheckLogout**

Ce test s’attend à ce qu’un appel à *Logout* révoke le JWT (astuce : grâce à son identifiant unique), et à ce que les authentifications ultérieures avec ce JWT soient refusées.

P.S. : Oui, sachant qu’il n’y a pas de persistance, stockez la liste noire d’ID de JWT en mémoire.

**T5\_CheckSpam**

Ce test s’attend à ce que si on envoie plus de 10 messages en 10 secondes, le message ne soit pas envoyé, et que l’utilisateur fautif reçoive un événement *StopSpamming*.  
  
Utilisez votre classe *AntiSpam* pour cela.

**T6\_SendUserList**

Ce test s’attend à ce que, à la connexion d’un utilisateur, celui-ci reçoive autant de *AddUser* que d’utilisateurs déjà connectés. De la même façon, les autres utilisateurs déjà connectés doivent recevoir un *AddUser* pour signaler le nouvel utilisateur connecté.

**T7\_CheckUserRemove**

Ce test s’attend à ce que, au *Logout* d’un utilisateur, ou lorsque sa connexion est coupée, les autres utilisateurs reçoivent un *RemoveUser* pour signaler la déconnexion de l’utilisateur.

Utilisez le *OnDisconnectedAsync*.