# 1. Introduction

Les établissements de santé échangent quotidiennement des informations cliniques et administratives (ordonnances, résultats de laboratoire, admissions, etc.) au format **HL7 v2.x**, un standard largement utilisé.  
Pour transporter ces messages, le protocole **MLLP** (Minimal Lower Layer Protocol) est couramment employé. Il encapsule les messages HL7 dans des trames TCP non chiffrées.

MLLP a l’avantage d’être léger et largement implémenté dans les moteurs HL7, mais il ne propose ni chiffrement, ni authentification. Les messages HL7 circulant sur le réseau sont donc exposés à plusieurs risques : écoute passive, altération des trames, attaques de type Man in the Middle, etc.

* Objectif du Projet Tunnel :

Le projet **Tunnel** a pour objectif de sécuriser les échanges HL7 **sans modifier le code** des applications existantes (client HL7 émetteur et serveur HL7 récepteur).  
Pour cela, on a introduit deux nouveaux services intermédiaires :

1. **Tunnel-Client** : s’intercale immédiatement après l’application cliente HL7. Il chiffre les messages HL7 reçus en clair, puis les transmet au TunnelServer.
2. **Tunnel-Server** : s’intercale juste avant l’application serveur HL7. Il déchiffre le flux TLS 1.2 reçu de TunnelClient, puis transmet les messages en clair au serveur HL7.

Ces deux services sont **entièrement autonomes,** ce qui permet au client et au serveur HL7 de conserver leur logique initiale, **sans modification de code**, à l’exception éventuelle des ports (selon la configuration).

Cette architecture offre plusieurs bénéfices majeurs. Elle garantit la **sécurité** des échanges grâce au chiffrement TLS 1.2, tout en préservant la **compatibilité** avec les applications HL7 existantes. La solution est **modulaire**, puisque Tunnel-Client et Tunnel-Server sont des services Java indépendants et reconfigurables, et **simple à déployer**, que ce soit sur des environnements distincts ou via Docker.

Conteneurisation avec Docker :

Pour simplifier l’installation et le déploiement, l’ensemble du projet a été conteneurisé :

* Chaque composant (App-Client, Tunnel-Client, App-Server, Tunnel-Server) est distribué sous forme d’**images Docker**.
* Les applications web (App-Client et App-Server) s’exécutent dans des conteneurs Tomcat, tandis que les tunnels fonctionnent dans des conteneurs Java légers.
* Le déploiement est orchestré via **docker-compose**, séparément sur la machine client (Machine A) et la machine serveur (Machine B).
* Les **certificats, paramètres et journaux** sont montés dans les conteneurs via des volumes, garantissant la persistance des données et une configuration simplifiée.

Grâce à Docker, le déploiement devient plus rapide, reproductible et indépendant de l’environnement d’hébergement, tout en conservant les mêmes garanties de **sécurité** et de **compatibilité** que l’architecture initiale.

Contenu de la documentation :

La suite de cette documentation détaillera :

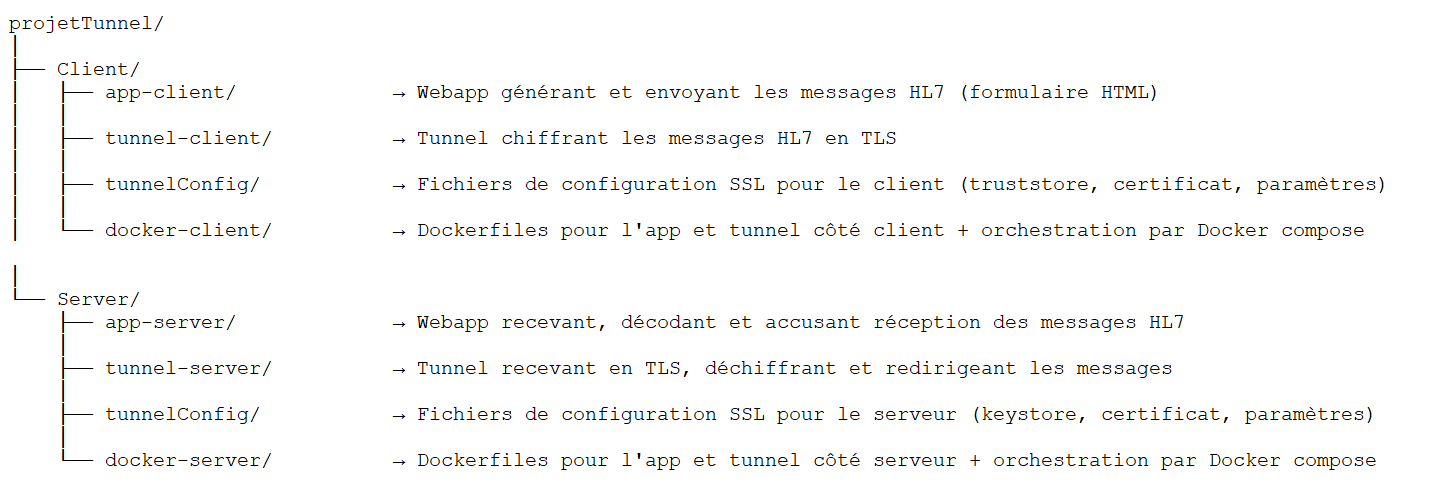
1. L’architecture générale du système.
2. Les étapes d’installation, de compilation et de déploiement.
3. Les paramètres de configuration si besoin.
4. Les principes de fonctionnement et les protocoles utilisés.
5. Le flux d’utilisation du projet.
6. Les détails techniques d’implémentation des différents services (App-Client,  
   App-Server, Tunnel-Client et Tunnel-Server).

# 2. Architecture générale

La section Architecture générale présente le rôle, le placement et le fonctionnement des différents composants du système Tunnel.

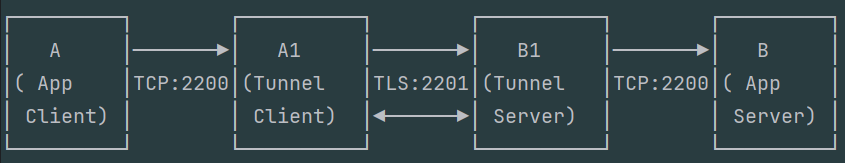
## 2.1 Structure globale du projet

Le projet est désormais organisé en **deux branches distinctes** : **Client** (sur la machine client) et **Server** (sur la machine serveur). Chacune contient une application web déployée sur Tomcat émettrice ou réceptrice, un point de tunnel pour relayer le message, les fichiers de configuration SSL (keystore / truststore), et finalement un répertoire docker-\*/ contenant les Dockerfile et le fichier docker-compose pour déployer ces services.



Chaque branche (Client / Server) dispose de son propre fichier compose.\*.yaml permettant de déployer les deux services locaux dans des conteneurs isolés.

## 2.1. Vue d’ensemble du flux (A → A1 → B1 → B)



1. **App-Client (A) = Émetteur HL7** **:** générateur de message HL7 à partir d’un formulaire, et envoie MLLP non chiffré sur 2200.
2. **Tunnel-Client (A1)** : écoute sur 2201, reçoit le flux MLLP, établit un **SSLSocket** vers B1.
3. **Tunnel-Server (B1)** : serveur SSL (SSLServerSocket), accepte la connexion chiffrée, déchiffre le flux, extrait la trame MLLP.
4. **App-Server (B)** = **récepteur HL7 :** servlet cible sur Tomcat, reçoit MLLP non chiffré sur 2200 et lance l’accusé de réception.

*Architecture initiale:*

ClientServer  
  **--------------- ---------------**

**A** **---------** *MLLP* **---------> B**

*Architecture finale:*

Client Server  
**-------------- --------------**

**A** **B**  
| ↑  
| *MLLP*  
*MLLP* |  
↓ |  
**A1** **-------** *SSL/TLS* **-------> B**

## 2.2. Protocoles et normes

### HL7 v2.x

HL7 v2.x est une norme de format de données textuelles pour l’échange d’informations médicales.

* **Segments** : lignes commençant par un code à trois lettres (ex. MSH, PID, OBR)
* **Champs** : séparés par le caractère pipe |
* **Composants** : à l’intérieur d’un champ, séparés par le caret ^
* **Sous-composants** : séparées par &
* **Répétitions** : séparées par le tilde ~  
  Le standard définit pour chaque segment l’ordre et le type attendu des champs, ce qui garantit l’interopérabilité entre systèmes. Dans ce projet, *semantsel.xml* formalise cette structure pour générer automatiquement le formulaire et valider la saisie.

### MLLP (Minimal Lower Layer Protocol)

MLLP est le protocole de transport le plus répandu pour faire circuler des messages HL7 v2.x sur TCP. Un message HL7 brut est encapsulé entre deux délimiteurs :

* **VT** (Vertical Tab, 0x0B) au début
* **FS** (File Separator, 0x1C) suivi de **CR** (Carriage Return, 0x0D) à la fin  
  Cette simple trame permet au récepteur de repérer exactement où commence et où se termine chaque message, même lorsque plusieurs messages sont envoyés sur la même connexion TCP persistante.

### SSL/TLS (Tunnel A1 → B1)

Le tunnel entre A1 (TunnelClient) et B1 (TunnelServer) repose sur **TLS 1.2** pour garantir confidentialité, intégrité et authentification :

* **Handshake** :

A1 démarre en envoyant un *ClientHello* dans lequel il précise sa version TLS (par exemple TLS 1.2 ou TLS 1.3) et la liste des suites de chiffrement qu’il accepte. B1 lui répond par un *ServerHello* : choisit la version et la suite de chiffrement communes, puis il transmet son certificat X.509 signé par une autorité de confiance et peut demander à A1 de fournir également un certificat client (pour le mutual TLS). Immédiatement après, A1 et B1 négocient une clé de session symétrique. A1 valide ensuite la chaîne de certification du certificat de B1 à l’aide de son truststore et, en cas d’échec de cette vérification, la connexion est immédiatement coupée.

* **Flux chiffré** :

Une fois le handshake terminé et validé (A1 vérifie bien dans son truststore le certificat de B1), le flux MLLP brut est encapsulé et protégé par AES-GCM ou équivalent, rendant impossible la lecture ou la modification des messages.

* **Stores** :

B1 conserve dans un fichier **keystore.jks** sa clé privée et son certificat **X.509**, et de l’autre côté A1 dispose d’un **truststore.jks** contenant soit la certificat racine de l’autorité qui a émis le certificat de B1, soit le certificat de B1 lui‑même. En production, ces stores doivent contenir des certificats émis par une autorité de confiance (Let’s Encrypt, PKI interne).

### TLS handshake protocol | Download Scientific DiagramEtapes détaillées du Handshake :

* **Explication :**

1. **Client Hello :**Le **client** démarre la négociation en envoyant un message **ClientHello** qui contient notamment :
   * La **version TLS** qu’il supporte (par ex. TLS 1.2 ou 1.3)
   * Une **liste de suites de chiffrement** (*cipher suites*) qu’il est capable d’utiliser
   * Un **nombre aléatoire** (*nonce*) généré par le client
   * Un **identifiant de session** (s’il s’agit d’une reprise de session)
   * Le **SNI** (*Server Name Indication*), c’est‑à‑dire le nom de domaine du serveur auquel il souhaite se connecter
2. **Server Hello**Le **serveur** répond avec un message **ServerHello** qui récapitule :
   * La **version TLS** choisie (la plus haute version commune)
   * La **suite de chiffrement** qu’il a sélectionnée
   * Son propre **nonce**
   * Un **identifiant de session** (nouveau ou repris)
   * Le SNI est vide (puisque c’était à l’initiative du client)
3. **Server Certificates**Le serveur envoie ensuite son **certificat X.509**, qui contient sa clé publique. Le client devra vérifier que ce certificat figure dans son **truststore** ou qu’il est signé par une autorité de certification (CA) de confiance. Dans mon cas, comme je suis dans la phase de développement j’utilise simplement un certificat auto-signé, mais en production, il est nécessaire d’utiliser un certificat signé par une autorité de certification reconnue pour garantir la sécurité et la confiance des connexions.
4. **Server Hello Done**Un simple message pour indiquer que le serveur a terminé sa phase de hello et de présentation des certificats.
5. **Client Key Exchange Message**Le client envoie à son tour un message ***ClientKeyExchange***, qui contient, selon la méthode de négociation : ***RSA key‑exchange*** (un **pré‑master secret** chiffré à la clé publique du serveur) ou **(EC)DHE** : ses paramètres Diffie‑Hellman (ou elliptic‑curve DH) pour établir un secret partagé.
6. **Key Generation**À partir du **pré‑master secret** (ou du secret DH) et des deux nonces (client et serveur), les deux parties dérivent :les **clés de chiffrement** symétriques et les **clés d’intégrité** (MAC keys).
7. **Change Cipher Spec**Chaque côté envoie un message ***ChangeCipherSpec*** pour passer **en mode chiffré** avec les clés qui viennent d’être dérivées.
8. **Finished (Client)**Le **client** envoie un message ***Finished*** chiffré contenant un **hash** de tous les messages échangés jusque‑là pour prouver qu’il a bien les mêmes clés et qu’aucune altération n’a eu lieu.
9. **Cipher Spec + Finished (Server)**Le **serveur** retourne lui aussi un ***ChangeCipherSpec*** puis son propre ***Finished*** chiffré, avec le même mécanisme de hash.
   * **Record Protocol (Application Data)**Une fois le handshake achevé, le **Record Protocol** prend le relais :toutes les données sont chiffrées et authentifiées avec les clés négociées.

## 2.3. Composants et rôles

|  |  |  |  |
| --- | --- | --- | --- |
| Composant | Classes clès | Ports | Rôle principal |
| App-Client (A) | Formulaire web patients  +  Servlet socketSenderSt.java | HTTP 8080 → socket socketSenderSt | Génère le message HL7 depuis le formulaire |
| TCP 2200 (vers A1) | Ouvre une connexion TCP vers TunnelClient (A1) pour envoyer la trame MLLP |
| Tunnel‑Client (A1) | TunnelClient.java | TCP 2200 (listen MLLP) | Écoute la trame MLLP en clair sur 2200 |
| TLS 2201 (vers B1) | Crée un SSLSocket (via SSLContext) vers TunnelServer (B1) sur 2201 et transmet octet par octet le flux HL7 dans le tunnel TLS |
| Tunnel‑Server (B1) | TunnelServer.java | TLS 2201 (listen) | Accepte connexions TLS sur 2201 et déchiffre la couche TLS pour lire le flux MLLP |
| TCP 2200 (vers B) | Ouvre un socket TCP vers le récepteur final (B) sur 2200 et relaye en clair le message HL7 |
| App‑Server (B) | Servlet  zetaSockHl7.java + socketSrv.java | TCP 2200 (listen) | Écoute sur 2200 la trame MLLP, la dé‑encapsule et parse le HL7.  Log et génération d’un ACK |

# Structure détaillée du projet :

Voici la structure globale du projet :

|  |  |
| --- | --- |
| * la structure détaillée côté client : | * la structure détaillée côté serveur : |

# 4. Installation et Déploiment

Cette section décrit pas à pas comment préparer l’environnement, compiler et déployer les différents modules du projet Tunnel.

Deux modes de déploiement sont possibles :

* **Mode classique (sans Docker)** : compilation avec Maven + déploiement manuel dans Tomcat et en lançant directement les JAR.
* **Mode Docker (recommandé)** : chaque composant est embarqué dans un conteneur isolé et orchestré par docker-compose.

## 4.1. Mode classique (sans Docker)

### 4.1.1 Prérequis

Avant de compiler et déployer les quatre modules (app-client, tunnel-client, tunnel-server et app-server), assurez-vous d’avoir :

* **Java 8u321** installé (JDK 1.8)   
  Vérification : java -version # doit afficher : java version "1.8.0\_321"
* **Maven 3.6+** (pour compiler et générer les artefacts)  
  Vérification : mvn -version
* **Apache Tomcat 9.0.102** (pour déployer manuellement les app-client et app-server)
* **keytool** (fourni avec le JDK)

### 4.1.2. Compilation

Vous avez désormais deux projets distincts (**Client** et **Server**). Il faut compiler chacun séparément sur une machine :

* Côté Serveur (=Machine B)
  + Copiez le dossier projetTunnel\Server\ sur la **Machine B** (Server).
  + cd \chemin\vers\projetTunnel\Server\app-server
  + mvn clean package  
    **→ génère: target\app-server.war**
  + cd ..\tunnel-server
  + mvn clean package  
    **→ génère: target\tunnel-server-1.0.0.jar**
* Côté Client (=Machine A)
  + Copiez le dossier projetTunnel\Client sur la **Machine A** (Client).
  + cd \chemin\vers\projetTunnel\Client\app-client
  + mvn clean package  
    **→ génère: target\app-client.war**
  + cd ..\tunnel-client
  + mvn clean package  
    **→ génère: target\tunnel-client-1.0.0.jar**

À la fin du build, vous aurez les 4 fichiers :

* projetTunnel \ Client \ app-client \ target \ **app-client.war**
* projetTunnel \ Client \ tunnel-client \ target \ **tunnel-client-1.0.0.jar**
* projetTunnel \ Server \ app-server \ target \ **app-server.war**
* projetTunnel \ Server \ tunnel-server \ target \ **tunnel-server-1.0.0.jar**

### 4.1.3. Déploiement

Avant le déploiement, vous devez configurer les certificats si nécessaire ; reportez-vous à la section 5 « [5. Configuration](#_4._Configuration) » pour plus de détails.

* Côté Serveur(=Machine B)

Depuis le répertoire : projetTunnel\Server\

* **App-Server :**Copiez le war **’**app-server\target\**app-server.war’** vers le webapps de Tomcat.Puis, démarrez Tomcat avec : \chemin\vers\tomcat\bin\startup.bat  
  Pour vérifier le déploiement, voir les logs de Tomcat (catalina.log).→ Écoute en clair sur localPort 2200.
* **Tunnel-Server :**Lancez le jar **’tunnel-server-1.0.0.jar’**en exécutant :

java -DconfigDir=".\tunnelConfig" -jar "tunnel-server\target\tunnel-server-1.0.0.jar"

→ Écoute SSL sur localPort 2201 (vérifier C:\def\tunnel-logs\logs\_ tunnel-server.log).

* Côté Client (=Machine A)

Depuis le répertoire : projetTunnel\Client\

* **App-Client :**Copiez **’**app-client\target\**app-client.war’** vers le webapps de Tomcat.Puis, démarrez Tomcat avec : \chemin\vers\tomcat\bin\startup.batPour vérifier l’interface frontend, allez :   
  <http://localhost:8080/app-client/formulaire/index.html>
* **Tunnel-Client :**

Lancez le jar **’tunnel-client-1.0.0.jar’**, en exécutant :

java -DconfigDir=".\tunnelConfig" -jar "tunnel-client\target\tunnel-client-1.0.0.jar"

→ Écoute en clair sur localPort 2200 (vérifier C:\def\tunnel-logs\logs\_ tunnel-client.log).

* **Remarque :**  
  Si vous voulez personnaliser le chemin du répertoire de configuration, il suffit de modifier l’option -DconfigDir=".\tunnelConfig" par  
  -DconfigDir="chemin\vers\tunnelConfig"

Ces quelques étapes configurent et démarrent l’ensemble des quatre modules.

## 4.2. Mode Docker (recommandé)

### 4.2.1. Prérequis

**1. Installer Docker Desktop :**Avant de compiler et déployer les quatre modules via Docker, assurez-vous d’avoir installé la dernière version stable de [Docker Desktop](https://docs.docker.com/desktop/setup/install/windows-install/) (incluant Docker Engine ≥ 20.10 et Docker Compose ≥ 2.0), ce qui pourrait nécessiter les droits administrateur.

* Vérification : docker -v # version de Docker Engine  
   docker compose version # version de Docker Compose

Contrairement au mode classique, **aucun Java/Maven ni Tomcat** n’est requis sur la machine hôte : ils sont intégrés automatiquement dans les images Docker.

**2. Libérer les ports :**Assurez-vous que les ports suivants soient libres et ouverts dans le pare-feu : **- côté client** : 8080, 2200 **- côté serveur** : 2200, 2201

**3. Création des dossiers sur l’hôte et partage avec Docker Desktop :**Avant de lancer les conteneurs, il faut d’abord créer les répertoires nécessaires :

* C:\def\tunnel-logs pour stocker les logs, dans les machines **client** et **serveur**.
* C:\def\messageStore pour stocker les messages HL7 reçus, dans la machine **serveur**.

Ensuite, il faut donner à Docker Desktop les droits de lecture et d’écriture dans ces dossiers. Pour cela ouvrir **Docker Desktop → Settings → Resources → File Sharing,** et ajouterchacun d’eux, ou même le dossier générique C:\def\

⚠️ Remarque : placez vos certificats et fichiers de configuration **avant le build**, car ils sont montés comme volumes dans les conteneurs. Pour cela, référez-vous à la section 5   
« [5. Configuration](#_4._Configuration) »

### 4.2.2. Build des images et lancement des conteneurs

Le projet est découpé en deux parties distinctes : **Client/** côté client et **Server/** côté serveur.  
Chacune possède son propre docker-compose et ses Dockerfile multi-stage qui embarquent Maven et Tomcat.

* Côté Serveur (Machine B)

Depuis le répertoire projetTunnel/Server/docker-server, exécutez :

* + docker compose -f compose.server.yaml build  
    → Cela compile automatiquement app-server et tunnel-server, puis génère deux images Docker prêtes à l’emploi.
  + docker compose -f compose.server.yaml up -d  
    → Cela démarre les conteneurs app-server et tunnel-server en arrière-plan.

Ou bien avec une seule commande (pas recommandé) :

* + docker compose -f compose.server.yaml up -d –build

**Remarque :** Assurez-vous de lancer Docker Desktop avant d’exécuter ces commandes.

* + Ce qui a pour effet :
* **Conteneur app-server** démarré: construit et déploie le fichier app-server.war dans un Tomcat embarqué → exposé sur ports **8080** et **2200**
* **Conteneur tunnel-server démarré**: construit le jar tunnel-server-1.0.0.jar et le lance avec java -DconfigDir=/tunnelConfig → exposé sur port **2201**
* Les logs des services sont stockés à l’intérieur de chaque conteneur (dans /var/log/tunnel-logs à accéder par la commande docker excec…), mais aussi dans C:\def\tunnel-logs pour une raison de traçabilité et d’accessibilité.
* Le keystore (keystore.jks) est monté dans /tunnelConfig du conteneur tunnel-server
  + **Vérification :**
* Exécuter docker ps pour vérifier si les conteneurs sont bien lancés
* Pour vérifier le déploiment, vous pouvez consulter les logs dans C:\def\tunnel-logs (monté en /var/log/tunnel-logs dans les conteneurs), ou accéder aux logs du docker : docker log <nom\_conteneur>
* Côté Client (Machine A)

Depuis le répertoire projetTunnel/Client/docker-client, exécutez :

* + docker compose -f compose.client.yaml build

→ Cela compile automatiquement app-server et tunnel-server, puis génère deux images Docker prêtes à l’emploi

* + docker compose -f compose.client.yaml up -d  
    → Cela démarre les conteneurs app-client et tunnel-client en arrière-plan.

Ou bien avec une seule commande :

* + docker compose -f compose.client.yaml up -d –build
  + Ce qui a pour effet :
* **Conteneur app-client** démarré: construit et déploie le fichier app-client.war dans un Tomcat embarqué → exposé sur ports **8080**
* **Conteneur tunnel-client démarré**: construit le jar tunnel-client-1.0.0.jar et le lance avec java -DconfigDir=/tunnelConfig → exposé sur port **2200**
* Les logs des services sont stockés à l’intérieur de chaque conteneur (dans /var/log/tunnel-logs à accéder par la commande docker excec…), mais pour une raison de traçabilité et d’accessibilité, ils sont aussi stockés dans C:\def\tunnel-logs. De la même façon, les messages HL7 reçus sont stockés dans C:\def\messageStore
* Le truststore (truststore.jks) est monté dans /tunnelConfig du conteneur tunnel-client
  + **Vérification :**
* Exécuter docker ps pour vérifier si les conteneurs sont bien lancés
* Pour vérifier le déploiment, vous pouvez consulter les logs dans C:\def\tunnel-logs (monté en /var/log/tunnel-logs dans les conteneurs), ou accéder aux logs du docker : docker log <nom\_conteneur>
* Pour tester l’interface web du service app-client :  
  <http://localhost:8080/app-client/formulaire/index.html>

### 4.2.3. Résultat attendu

À la fin du build, vous disposez alors de **4 conteneurs actifs** :

* app-client (Tomcat, formulaire HL7 → port 8080)
* tunnel-client (écoute MLLP clair sur 2200, sortie TLS vers serveur)
* app-server (Tomcat, réception HL7 → port 2200)
* tunnel-server (écoute TLS sur 2201, sortie MLLP clair vers app-server)
* Vérification rapide : docker ps  
  # doit afficher les 4 conteneurs avec les ports exposés.
* **Remarque :**

Les docker-compose montent automatiquement certains volumes entre la machine hôte et les conteneurs :

* tunnelConfig/ → /tunnelConfig (certificats, parameters.xml)
* C:\def\tunnel-logs → /var/log/tunnel-logs (logs des services)
* C:\def\messageStore → /var/messageStore (messages HL7 reçus)

⚠️ Si vous souhaitez personnaliser les chemins, il suffit de modifier la section volumes: dans le fichier compose.client.yaml ou compose.server.yaml.

# 5. Configuration

L’architecture Dockerisée du projet repose sur plusieurs niveaux de configuration à réaliser au besoin :

1. **Sécurité (certificats SSL)**: pour chiffrer la communication TLS entre client et serveur
2. **Paramètres de connexion (parameters.xml ou web.xml)** : pour indiquer ports, hôtes et chemins de logs.
3. **Conteneurisation (Docker Compose & Dockerfile)** : pour définir et assembler les services.
4. **Persistance des données (volumes)** : pour que les logs et messages HL7 restent accessibles en dehors des conteneurs.

Chacun de ces points est détaillé ci-dessous.

## 5.1. Certificats SSL (Keystore & Truststore)

La sécurité du tunnel repose sur TLS :

* **Côté serveur** : un keystore.jks contient la clé privée et le certificat X.509 du Tunnel-Server.
* **Côté client** : un truststore.jks embarque le certificat public du serveur afin de valider son identité.

Ces fichiers se trouvent respectivement dans Server/tunnelConfig et Client/tunnelConfig, puis sont **montés dans les conteneurs** à l’exécution.

Pour chiffrer la liaison A1 ↔ B1, vous devez générer et échanger des certificats, voici la procédure utilisée pour les certificats auto-signés :

* Côté Serveur (*Tunnel-Server*) :

Allez au dossier : Server\tunnelConfig\

1. **Générer le keystore serveur (B1)**

keytool -genkeypair \  
 -alias server \  
 -keyalg RSA -keysize 2048 \  
 -keystore keystore.jks \  
 -storepass changeit \  
 -keypass changeit \  
 -validity 365 \  
 -dname "CN=example.com, OU=IT, O=Example, L=Paris, C=FR" \  
 *-*ext "SAN=dns:example.com"

- Dans cette phase de développement, j’ai pris juste: -dname "CN=localhost"

*- Exemple de test* : si votre machine serveur s’appelle `def-s9`, vous devez prendre :

-dname "CN=def-s9, OU=IT, O=Example, L=Paris, C=FR" \  
 *-*ext "SAN=dns:def-s9"

Ou avec une adresse IP :

-dname "CN=192.168.0.5, OU=IT, O= Example, L=Paris, C=FR" \  
-ext SAN=IP:192.168.0.5

1. **Exporter le certificat public client**

keytool -exportcert \  
 -alias server \  
 -keystore keystore.jks \  
 -storepass changeit \  
 -rfc \  
 -file cert.pem

Vérifiez que : les deux fichiers `keystore.jks` (contenant la clé privée + certif) et `cert.pem` sont bien générés dans le répertoire `Server\tunnelConfig\`.

* Côté Client (*Tunnel-Client*) :

Allez au dossier : Client\tunnelConfig\

1. **Copier le certificat du serveur généré (dans** Client\tunnelConfig\**).**
2. **Créer un truststore pour A1 et y importer ce certificat :**

keytool -importcert \  
 -alias server \  
 -file cert.pem \  
 -keystore truststore.jks \  
 -storepass changeit \  
 -noprompt

Vérifiez que le fichier `truststore.jks` (contenant le certif du serveur) est bien généré dans le répertoire `Client\tunnelConfig\`.

* **Remarque 1 :**  
  En production, il faut remplacer les certificats auto-signés par des certificats délivrés par une autorité de confiance (CA interne ou publique). N’oubliez également pas d’adapter les mots de passe, alias, CN, SAN et chemins à votre cas.
* **Remarque 2 :**  
  Pour activer l’authentification mutuelle (TLS bidirectionnel), c’est-à-dire ajouter l’authentification du client, il suffit de répéter les mêmes commandes ci-dessus en inversant simplement les rôles client et serveur.

## 5.2. Fichiers de configuration (parameters.xml et web.xml)

Chaque instance Tunnel lit son fichier parameters.xml pour indiquer ses points d’écoute, de destination et ses stores SSL.

* **Exemple côté client :** Client\tunnelConfig\parameters.xml :

Détails :

* + **localPort** : port TCP local d’écoute de tunnel-client (A1).
  + **remoteHost, remotePort** : hôte du seveur SSL (le tunnel-server B1) et son port.
  + **truststoreFile**, **truststorePassword :** chemin vers le fichier truststore.jks contenant le certificat CA du serveur + mot de passe du truststore.
  + **logDirectory, logFileName, logLevel :** les paramètres de logging du tunnel-client. Ici logDirectory = /var/log/tunnel-logs (monté en volume à l’intérieur du conteneur, et relié ensuite à l’hôte par C:\def\tunnel-logs)
* **Optionnel (**pour l’authentification mutuelle)**:**

**keystoreFile, keystorePassword :** chemin vers le fichier keystore.jks contenant la clé privée et le certificat du client + mot de passe du keystore.

* **Exemple côté serveur :** Server\tunnelConfig\parameters.xml :  
  ****

Détails :

* + **localPort** : port TCP local d’écoute SSL (de tunnel-server B1).
  + **remoteHost, remotePort** : hôte final de app-server (B) et son port. Ici remoteHost = app-server (nom du conteneur).
  + **keystoreFile, keystorePassword :** chemin vers le fichier keystore.jks contenant la clé privée et le certificat du serveur + mot de passe du keystore.
  + **logDirectory, logFileName, logLevel :** les paramètres de logging du tunnel-server. Ici logDirectory = /var/log/tunnel-logs (monté en volume à l’intérieur du conteneur, puis relié à l’hôte par C:\def\tunnel-logs).
* **Optionnel (**pour l’authentification mutuelle)**:**

**truststoreFile**, **truststorePassword :** chemin vers le fichier truststore.jks contenant le certificat CA du client + mot de passe du truststore.

* **Remarques :**
* Le champ **remoteHost** peut utiliser soit un nom DNS, soit le nom du conteneur docker (ex. app-server), grâce au réseau Docker interne.
* Les chemins **logDirectory** doivent pointer vers /var/log/tunnel-logs, qui est un chemin dans le système de fichier du conteneur, et sera monté en volume depuis l’hôte.
* Vous pouvez placer les dossiers **tunnelConfig** où vous le souhaitez ; il suffit de passer leur chemin via l’option -DconfigDir au lancement du JAR dans les dockerfiles.
* **App-Client :** web.xml :  
  ****
* **App-Server :** web.xml :  
  

## 5.3. Configuration des conteneurs (Docker Compose)

Le projet s’appuie sur **Docker Compose** pour orchestrer les services. En effet, chaque machine possède son fichier docker-compose spécifique (compose.client.yaml et compose.server.yaml), et plutôt de lancer manuellement chaque conteneur avec docker run, ce fichier définit l’ensemble :

* **Les services** (app-client et tunnel-client, app-server et tunnel-server)
* **Leur build** (Dockerfile utilisé, contexte de compilation)
* **Les ports exposés** (8080 pour l’UI, 2200 pour MLLP clair, 2201 pour TLS)
* **Les volumes montés** pour la traçabilité des données
* **Les dépendances** (depends\_on) pour garantir l’ordre de démarrage
* **La politique de redémarrage** (restart: unless-stopped)
  + **Exemple côté serveur** (compose.server.yaml) :

services:

  app-server:

    build:

      context: ../..

      dockerfile: Server/docker-server/Dockerfile.app-server

    container\_name: app-server

    expose: ["2202","8080"]

    volumes:

      - C:\\def\\messageStore:/var/messageStore

    restart: unless-stopped

  tunnel-server:

    build:

      context: ../..

      dockerfile: Server/docker-server/Dockerfile.tunnel-server

    container\_name: tunnel-server

    ports: ["2201:2201"]

    volumes:

      - ../tunnelConfig:/tunnelConfig:

      - C:\\def\\tunnel-logs:/var/log/tunnel-logs

    depends\_on: [app-server]

    restart: unless-stopped

* + **Points configurables :**
    - Ports exposés (2201 = TLS, 8080 = UI, 2200 = MLLP clair)
    - Volume de stockage des messages HL7 (/var/messageStore dans le conteneur et C:\def\messageStore dans l’hôte)
    - Volume de logs (/var/log/tunnel-logs en conteneur et C:\def\tunnel-logs dans l’hôte)

L’intérêt de Docker Compose est double :

* **Simplification** : un seul fichier décrit toute la stack, plus besoin de gérer les paramètres un par un.
* **Reproductibilité** : le même fichier peut être utilisé sur une autre machine pour recréer l’environnement à l’identique.

## 5.4. Construction des images (Dockerfile)

Chacun des quatre module (client ou serveur) dispose de son propre Dockerfile.  
Ils utilisent chacun une approche **multi-stage build** :

1. **Étape build (Maven)** : compile le code source et génère un artefact JAR/WAR.
2. **Étape runtime (Tomcat ou JRE léger)** : seul l’artefact final est copié dans une image légère contenant uniquement un environnement d’exécution (Tomcat pour les apps ou JRE pour les instances tunnel).

Cette méthode présente plusieurs avantages. Elle permet de produire des conteneurs beaucoup plus légers, car ils ne contiennent pas d’outils de build superflus (comme Maven). Elle garantit également que toutes les dépendances sont intégrées dès la compilation, assurant ainsi un environnement cohérent pour l’exécution. Enfin, le processus est standardisé et reproductible : le même Dockerfile peut être utilisé pour reconstruire l’image de manière identique sur n’importe quelle machine, ce qui facilite la portabilité et la maintenance.

* + **Exemple :** Dockerfile.tunnel-server

FROM maven:3.9-eclipse-temurin-8 AS build

WORKDIR /src

COPY Server/tunnel-server/pom.xml ./

RUN mvn -q -DskipTests dependency:go-offline

COPY Server/tunnel-server/ ./

RUN mvn -q -DskipTests clean package

FROM eclipse-temurin:8-jre

WORKDIR /app

COPY --from=build /src/target/tunnel-server-1.0.0.jar ./tunnel-server.jar

ENV CONFIG\_DIR=/tunnelConfig

EXPOSE 2201

CMD ["sh","-c","java -DconfigDir=${CONFIG\_DIR} -jar /app/tunnel-server.jar"]

* + **Points configurables:**
    - **Version de Java/Tomcat** : eclipse-temurin:8-jre**,** tomcat:9.0-jdk8
    - **Répertoires de travail** : par défaut /app pour les JAR et /usr/local/tomcat pour les WAR

## 5.5. Volumes et gestion des logs

Par défaut, un conteneur Docker est **éphémère** : toute donnée écrite à l’intérieur (dans son système de fichier) est perdue lorsqu’il est arrêté ou supprimé.  
Pour garantir la traçabilité et éviter toute perte, le projet s’appuie sur des **volumes montés** : des répertoires partagés entre l’hôte et les conteneurs.

Les volumes utilisés sont :

* **Logs** :  
  /var/log/tunnel-logs dans le conteneur **↔** C:\def\tunnel-logs sur l’hôte  
  - ils contiennent les fichiers logs :  
  logs\_app-client.log et logs\_tunnel-client.log côté client  
  logs\_tunnel-server.log et logs\_app-server.log (fichier actuellement vide car niveau de debug mis à 0) côté serveur.
* **Messages HL7 reçus** :  
  /var/messageStore dans le conteneur **↔** C:\def\messageStore sur l’hôte  
  - ils contiennent les messages en .hl7 et fichiers .ok de validation.
* **Certificats & paramètres** :  
  /tunnelConfig dans le conteneur **↔** /tunnelConfig sur l’hôte  
  - ils contiennent: parameters.xml + keystore.jks ou truststore.jks, cert.pem.  
  ainsi, à l’exécution du conteneur, les tunnels accèdent directement à leurs certificats.

Cela présente plusieurs avantages. Les fichiers (messages et logs) sont **persistants**, ce qui permet de les conserver même après un redémarrage ou un redéploiement. Ils sont également **accessibles** directement depuis la machine hôte, sans avoir besoin d’entrer dans les conteneurs avec des commandes Docker (docker exec). De plus, la configuration reste **flexible** : il suffit de modifier ou remplacer les fichiers de configuration (de tunnelConfig) côté hôte pour que le changement soit pris en compte au prochain redémarrage, sans avoir à reconstruire l’image du conteneur.

* + **Accès aux logs :**

Le chemin /var/log/tunnel-logs est accessible :

* **Directement sur l’hôte** via les volumes montés (C:\def\tunnel-logs), plus simple et persistant.
* **Depuis un conteneur** avec la commande :  
  pour le tunnel-client : docker exec -it tunnel-client cat /var/log/tunnel-logs/logs\_tunnel-client.log

Les sorties standards des conteneurs restent accessibles via :  
 docker logs app-client  
 docker logs app-server

# 6. Utilisation

Voici le flux d’utilisation du projet :

## 6.1. Démarrage des services

Vérifier que les services/conteneurs *app-server* et *tunnel-server* sont bien démarrés sur la machine « serveur », et les *app-client* et *tunnel-client* sur la machine « client ». Pour cela, référez-vous à la section « [4. Compilation & Déploiment](#_4._Compilation_&) »

* Vérification des conteneurs actifs : docker ps

## 6.2. Accès et génération du formulaire

1. **Ouvrir l’interface**

Une fois les 4 conteneurs lancés, accéder à l’interface web côté client :

<http://localhost:8080/app-client/formulaire/index.html>

1. **Chargement automatique**

À l’ouverture, la page charge les champs HL7 avec leurs options; vous retrouvez immédiatement la liste des segments (MSH, PID, …) et les menus déroulants préremplis.

1. **Remplir les informations Patient**

Saisissez ou sélectionnez les données du patient et du message.  
Un bouton “Auto-compléter” permet de remplir automatiquement le formulaire avec des valeurs fictives. C’est utile pour tester rapidement sans ressaisie manuelle.

**Explication technique :**

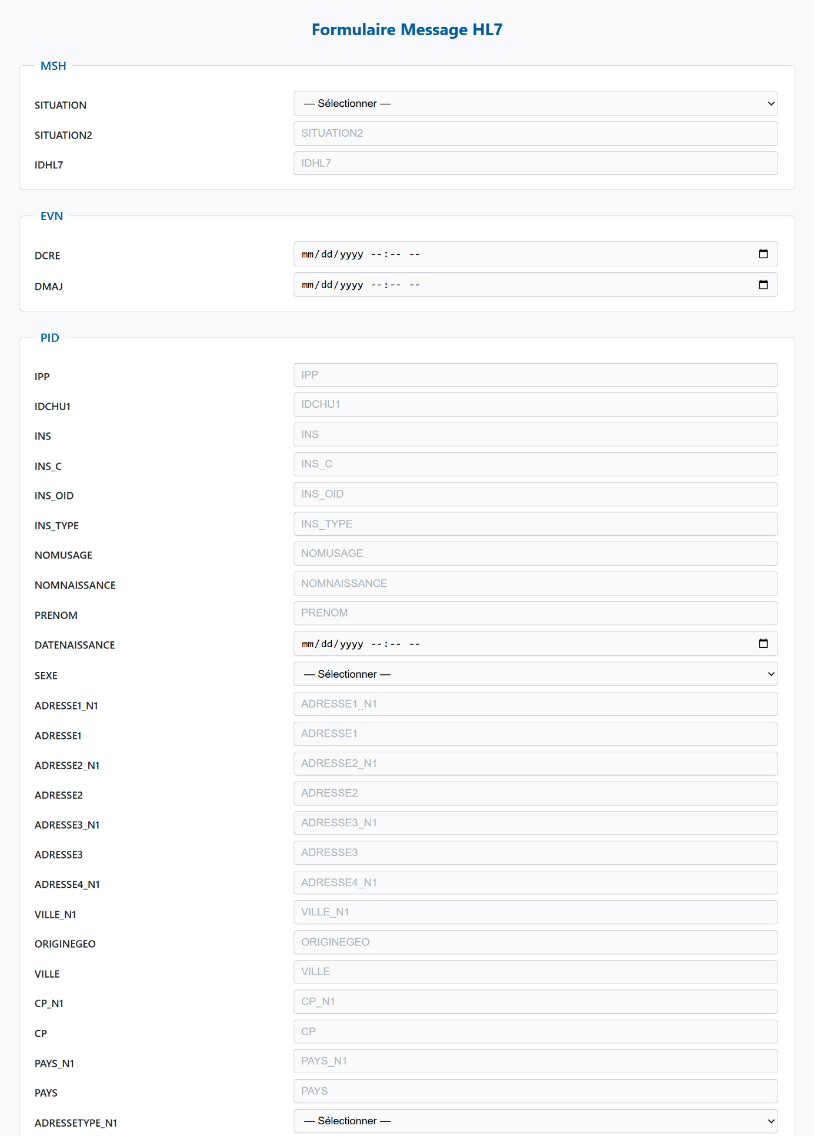
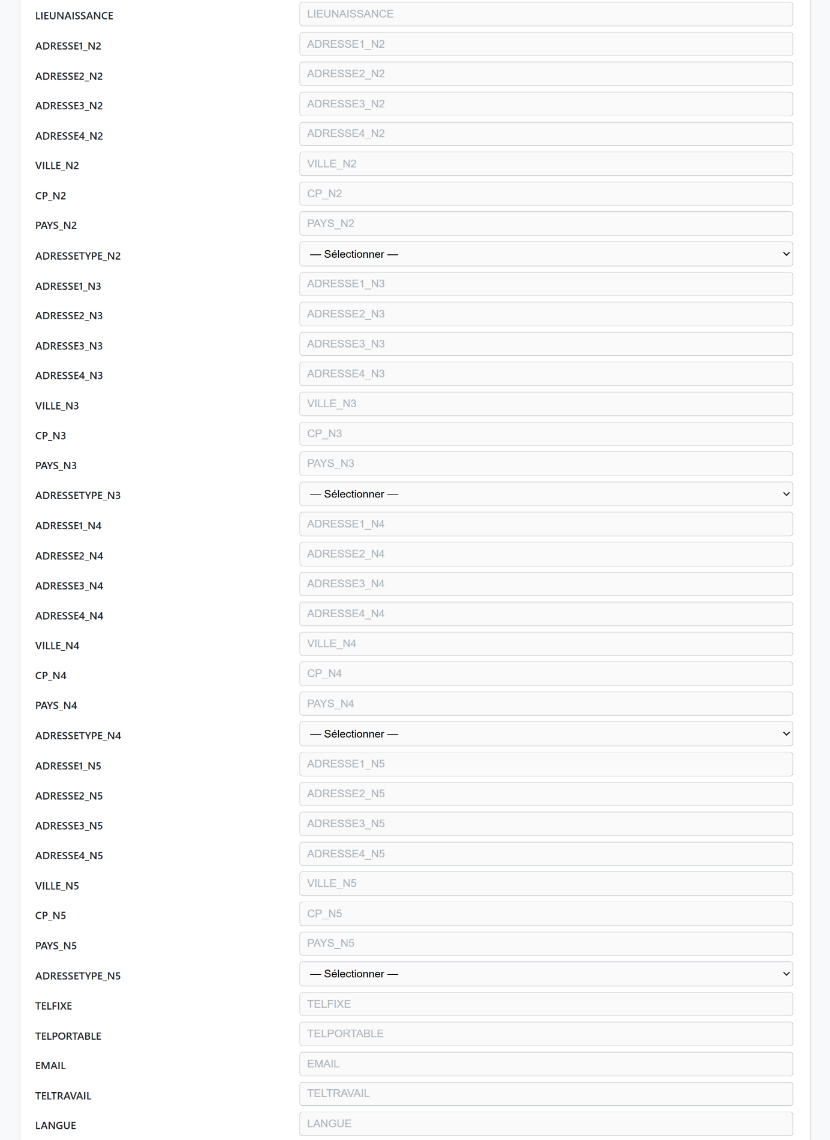
Le formulaire dynamique repose sur des fichiers clés:

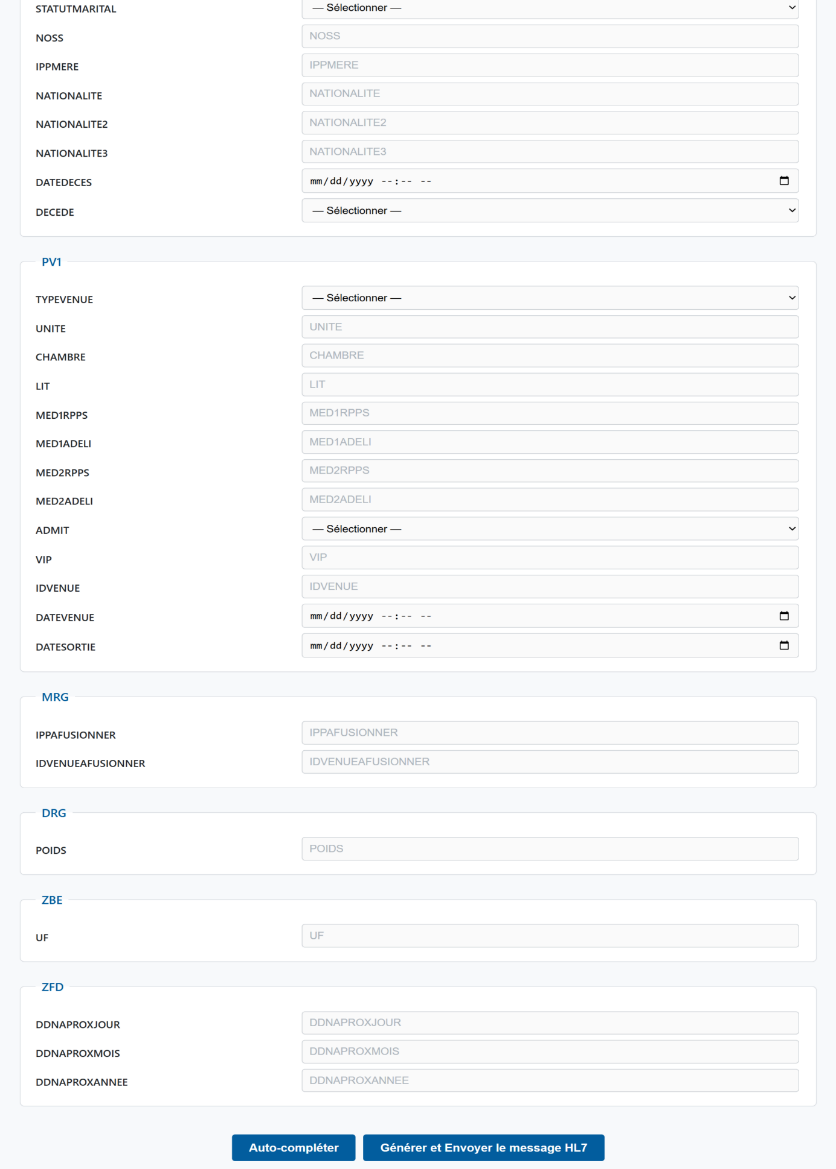
* **semantsel.xml** : fichier définissant la structure des champs HL7, leurs segments (PID, MSH, etc.), et leurs positions (champ, composant, sous-composant).
* **fieldOptions.json** : options pour les champs de type liste déroulante (ex : sexe, pays).
* **generateForm.js** : génère le HTML dynamiquement à partir des définitions.
* **generateHL7.js** : construit un message HL7 et l’envoie via HTTP.

Au chargement de la page index.html, le fichier generateForm.js est exécuté. Voici les étapes principales :

1. Il charge semantsel.xml et le parse avec dataLoader.js.
2. Organise les champs par segments HL7 (MSH, PID, PV1, etc.) grâce à segmentBuilder.js.
3. Crée dynamiquement les balises <input> ou <select> en fonction du type de champ.
4. Remplit les options si disponibles dans fieldOptions.json.

Cela permet de générer le formulaire entièrement à partir du fichier semantsel.xml, ce qui centralise la logique dans les données plutôt que dans le code. Il devient ainsi facile de modifier, ajouter ou supprimer des champs en ajustant uniquement semantsel.xml, sans avoir à toucher au code, rendant le formulaire modulable et plus simple à maintenir.

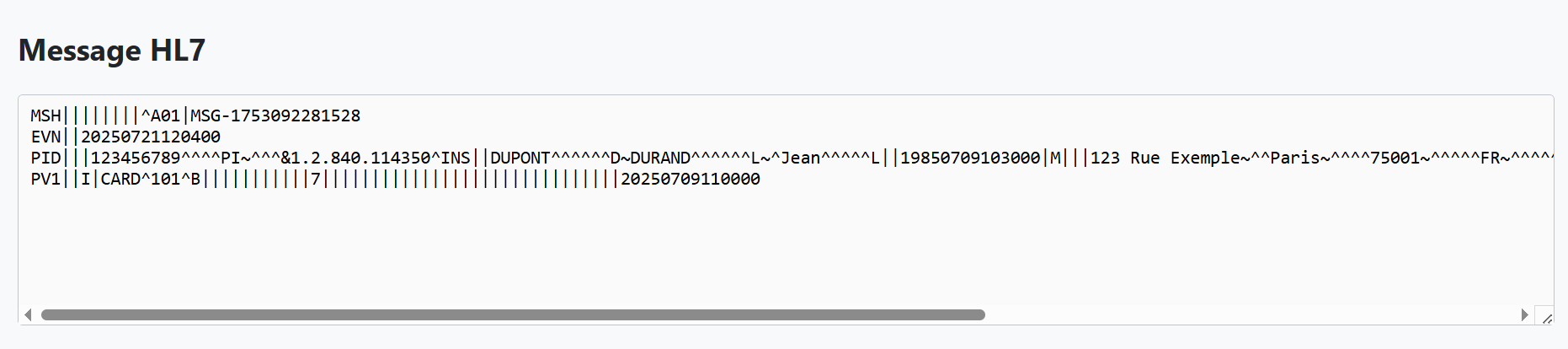
Voici le formulaire de simulation généré :



## 6.3. Génération du message & Envoi au servlet java

Quand l'utilisateur clique sur “Générer et Envoyer le message HL7” :

1. Le script generateHL7.js lit toutes les valeurs du formulaire.
2. Il construit les segments HL7 un par un en fonction des définitions XML :
   * gestion des séparateurs |, ^, &, ~,
   * conversion des dates en format YYYYMMDDhhmmss,
   * regroupement des champs par r,f,s (champ, composant, sous-composant).
   * Ajout d'attributs supplémentaires si définis (f1,s1,val1)
3. Le message construit est affiché dans une zone de texte <textarea> pour vérifier la génération.



Enfin, generateHL7.js envoie le message HL7 construit via une requête HTTP POST au servlet socketSenderSt.java, avec l’URL suivante :

/app-client/servlet/socketSender?zs:fnc=send

## 6.4. Transport du message : Étape Servlet socketSenderSt → TunnelClient

Le servlet **socketSenderSt.java** reçoit le message HL7, l’encode au format MLLP (encapsulation 0x0B…0x1C0D) et l’envoie localement au port 2200, où écoute TunnelClient (A1).

## 6.5. Transport du message : Étape TunnelClient → TunnelServer (SSL/TLS)

TunnelClient initie une connexion TLS vers remoteHost:remotePort (ex. localhost:2201), en validant le certificat du serveur B1 via son truststore.jks. Le flux MLLP brut est ensuite encapsulé dans ce canal sécurisé.

## 6.6. Transport du message : Étape TunnelServer → app-server

TunnelServer (B1) accepte la connexion SSL entrante sur le port 2201, déchiffre le message HL7, puis ouvre un socket TCP non-chiffrée vers le serveur final B (ex. localhost:2200). L’intégralité de la trame MLLP est transmise telle quelle.

## 6.7. Réception finale (app-server)

La classe socketSrv.java agit comme office de serveur HL7 côté réception : écoute sur le port 2200 les frames MLLP (0x0B…0x1C 0x0D), enregistre chaque message brut dans un fichier .hl7 (et crée un fichier vide .ok), puis renvoie immédiatement un ACK sur la même connexion TCP dans le chemin inverse (B → B1 → A1 → A).

Le fichier .hl7 est accessible dans le répertoire /var/messageStore (monté en volume depuis l’hôte C:\def\messageStore),

## 6.8. Surveillance des logs :

Les logs permettent de tracer le flux tout au long de transmission des messages, et sont à consulter pour régler tout dysfonctionnement probable. La sortie standard des conteneurs peut être consultée avec :

docker logs tunnel-client

docker logs tunnel-server

docker logs app-client

docker logs app-server

Ils sont également stockés dans les volumes montés :

/var/log/tunnel-logs **→** C:\def\tunnel-logs côté hôte.

Les fichiers de log eux-mêmes sont stockés dans les volumes montés et restent accessibles côté hôte : /var/log/tunnel-logs **→** C:\def\tunnel-logs

Pour consulter un fichier de log spécifique depuis un conteneur :  
docker exec -it tunnel-client cat /var/log/tunnel-logs/logs\_tunnel-client.log

Exemple de fichiers de logs :

* + **Logs côté app-client (par socketSenderSt) :** *logs\_app-client****.****log*



* + **Logs côté tunnel-client :** *logs\_tunnel-client.log*



* + **Logs côté tunnel-server :** *logs\_tunnel-server.log*



* + **Logs côté app-server :** *logs\_app-server.log (avec un debug = 5)*



Actuellement, le niveau de debug est mis à 0, à changer si besoin

# 7. Détails d’implémentation

Cette section plonge au cœur du code, en expliquant le fonctionnement de chaque module. Les extraits de code sont fournis à titre d’illustration ; reportez-vous aux fichiers sources pour l’intégralité.

## app-client :

Le **app-client** est une application web Java (WAR) exposant une partie front-end (formulaire HTML/JS) pour saisir et générer des messages HL7, et une partie back-end MLLP (le servlet socketSenderSt) qui encapsule ces messages et les transmet via TCP au **tunnel-client**.

### 1. Sources de données

* **data/semantsel.xml** :  
  Décrit la structure du message HL7 (segments, champs, composants, éventuelles valeurs fixes) selon les mappings SQL.  
  Chaque définition est un élément <sqlelement> qui a ces attributs clés :
  + sqlname : nom du champ en base (et ID dans le DOM)
  + type : type de saisie attendu (string, date, bool, sexe, etc.)
  + line : segment HL7 (MSH, PID, PV1, …)
  + r, selm, f, s : position HL7 (champ, répétition, composant, sous‑composant)
  + (pour les cas complexes) extraAttributs : liste de couples {f,s,val} à injecter systématiquement

**Exemple** (champ simple):

<sqlelement sqlname="TYPEVENUE" type="string" prio="0" line="PV1" r="2" selm="0" f="0" s="0" rname="PatientClass" fname="PatientClass" sname="PatientClass"/>

**Exemple (**champ complexe, càd avec des extraAttributes) :

<sqlelement sqlname="IPP" type="string">  
 <element prio="0" line="PID" r="3" f="0" s="0" rname="PatientIdentifierList" fname="IDNumber" sname="IDNumber" f1="4" s1="0" val1="PI"/>  
</sqlelement>

* **data/fieldOptions.json :**  
  Contient la liste des options {value, text} pour chaque sqlname qui est champ à choix. Elles sont à s’afficher dans un <select>.



* **data/autocomplete.json** **:**  
  Contient des valeurs de test permettant de préremplir le formulaire en un clic sur le bouton **Auto-compléter** :



### 2. Modules utilitaires

* **dataLoader.js :**

**- loadDefinitions()** lit semantsel.xml et construit un tableau d’objets defs décrivant pour chaque champ les coordonnées HL7 ainsi que les couples d’extraAttributes s’il y en a :

* **Remarque :**  
  Si prio="0" sur l’élément, on en extrait directement {sqlname, type, line, r, selm, f, s}. Sinon, on trie ses <element> internes par prio, on prend le premier, puis on collecte aussi les éventuels f1/val1, f2/val2,… en extraAttributs.

**-** **loadFieldOptions()** charge simplement fieldOptions.json et renvoie l’objet JavaScript.

* **segmentBuilder.js :**

**prepareSegments(defs)** regroupe les définitions defs par segment line, trie les items (r, selm, f, s) puis renvoie en objets :



L’ordre par défaut des segments est : MSH, EVN, PID, PV1, MRG, DRG. Ceux non listés sont ajoutés en queue.

### 3. generateForm.js (Génération du formulaire)

1. Chargement initial des options et définitions (fieldOptions + defs).
2. Pour chaque segment : création d’un <fieldset> ; pour chaque champ :
   * Si des options sont disponibles → <select> rempli.  
     Sinon, choix de l’élément en fonction de type.
   * sqlname devient le texte du <label> et l’id de l’input.
3. Le bouton **Auto‑compléter** remplit avec les champs avec autocomplete.json.

### 4. generateHL7.js (Génération du message)

1. **Pré-traitement** : remplit le champ IDHL7 (MSG‑<timestamp>) avec la date de clic sur le bouton générer, et le DCRE (date/heure courante) uniquement s’il est vide.
2. **Construction** : pour chaque segment :

* Injection automatique de MSH‑1 (|) et MSH‑2 (^~\\&) quand line === 'MSH'.
* Parcours de chaque numéro de champ r. Deux cas :
  + **Champ sans constantes** : gère les répétitions (selm) et composants (f, s) en fonction de la saisie réelle.
  + **Champ avec constantes** (extraAttributes) : construit une répétition (~) par définition, puis compose la chaîne HL7 via setComponent() pour combiner saisie utilisateur et constantes.
* Un segment est ajouté seulement si au moins un champ est renseigné.
* Les segments sont joints par CR

1. **Envoi Post** :



Le servlet relaie ensuite le flux vers le socket MLLP. Le message est également affiché dans le <textarea id="hl7Output"> pour inspection.

### 5. Servlet d’envoi : socketSenderSt.java

1. **Point d’entrée**  
   URL: POST /app-client/servlet/socketSender?zs:fnc=send  
   Corps: hl7msg=<message HL7 encodé>
2. **Encapsulation MLLP**
   * Récupère la chaîne hl7msg depuis request.getParameter("hl7msg").
   * Préfixe le flux par l’octet **0x0B** (VT) et suffixe par **0x1C 0x0D** (FS + CR).
   * Produit ainsi le paquet MLLP prêt à l’envoi sur une socket TCP.
3. **Envoi vers TunnelClient**
   * Ouvre un Socket vers localhost:2200.
   * Écrit le flux MLLP sur socket.getOutputStream().
   * Si configuré, lit la réponse (ACK) depuis socket.getInputStream() et la renvoie dans la réponse HTTP (body ou code de statut).
4. **Retour à l’utilisateur** 
   * En HTTP 200 en cas de succès, 500 sinon.
   * Log INFO/ERROR via java.util.logging dans le fichier logs\_app-client.log

* Packaging Docker :
* Le Dockerfile Dockerfile.app-client multi-stage basé sur maven:3.9-eclipse-temurin-8 pour compiler, puis tomcat:9.0-jdk8-temurin pour exécuter.
* Le conteneur expose **port 8080** (interface web) et se déploie automatiquement via docker-compose.

## 7.2 tunnel-client :

Le tunnel-client relaie localement des connexions TCP non chiffrées vers un serveur distant via TLS 1.2. Il se compose principalement de :

* **parameters.xml** : fichier de configuration des paramètres (ports, hôtes, keystore / truststore, logging).
* **ConfigLoader.java** : lecture/validation du fichier XML.
* **TunnelClient.java** : initialisation (logger, SSLContext) et gestion des connexions.

### 1. parameters.xml

Définit la configuration attendue par ConfigLoader :

* Les balises <localPort>, <remoteHost>, <remotePort> pilotent l’ouverture des connexions locale et distante.
* <truststoreFile> / <truststorePassword> servent à valider le serveur TLS, tandis que <keystoreFile> / <keystorePassword> (commentées) activent l’authentification mutuelle.
* Les paramètres de logging (<logDirectory>, <logFileName>, <logLevel>) sont lus pour configurer le logger de TunnelClient.

### 2. ConfigLoader.java

La classe **ConfigLoader** se charge d’ouvrir, de parser et de valider le fichier parameters.xml avant d’en exposer les paramètres sous forme de champs immuables.

* À la construction, on passe le File cfgFile pointant sur parameters.xml.
* **Parsing DOM** : on normalise l’arbre XML, on récupère les balises listées ci-dessus.
* **Validation & Mapping** :
  + Les balises requises (ports, hôtes, truststore) sont récupérées par getRequiredString/getRequiredInt. Toute absence ou valeur vide provoque une IllegalArgumentException.
  + Les balises optionnelles de mutual TLS sont lues par getOptionalString : si <keystoreFile> est défini, keystorePassword devient obligatoire, sinon les deux restent null.
  + Les paramètres de logging sont lus ou remplacés par leurs valeurs par défaut (./logs, logs\_tunnel-client.log, INFO), puis exposés sous forme de champs publics immuables (cfg.logDirectory, cfg.logLevel, etc.).

Chaque champ de ConfigLoader correspond directement à une balise de parameters.xml, garantissant que toute modification du XML se reflète immédiatement dans le comportement du tunnel-client.

### 3. TunnelClient.java

La classe **TunnelClient** constitue le point d’entrée, elle orchestre l’initialisation de la configuration, du logger et du contexte SSL, assurant le relais des données vers le serveur distant.

1. **Chargement de la configuration**

Dans main, on instancie un ConfigLoader en lui passant le chemin vers parameters.xml (par défaut ./tunnelConfig/parameters.xml ou modifiable via -DconfigDir). Si une balise requise manque ou contient une valeur invalide, ConfigLoader lance une exception qui provoque l’arrêt du programme (code 1).

1. **Logger**

Création du répertoire de logs (s’il n’existe pas déjà) : /var/log/tunnel-logs/logs\_tunnel-client.log, ajout d’un FileHandler en append, niveau et format simple.

1. **Initialisation du SSLContext**

La méthode createSSLContext charge le keystore (optionnel, si configuré) et le truststore (obligatoire) depuis le répertoire de config, puis instancie les KeyManager[] et TrustManager[], puis renvoie un SSLContext configuré pour TLS 1.2. Toute erreur d’I/O ou de mot de passe est loggée et interrompt l’exécution.

1. **Boucle d’écoute et relais des connexions**

Un ServerSocket écoute sur le port cfg.localPort. À chaque accept(), on crée un thread nommé client-handler-<port> qui invoque la méthode :

handleConnection(plainSocket, sslCtx, cfg.remoteHost,cfg.remotePort)

Dans handleConnection :

* + on établit une SSLSocket vers cfg.remoteHost:cfg.remotePort et on réalise le handshake TLS ;
  + on applique un timeout de 30 s ( = 30 000 ms) sur la socket cliente et sur la socket SSL, afin d’éviter tout blocage prolongé ;
  + deux threads relaient simultanément les flux plain→ssl et ssl→plain jusqu’à EOF :
    1. **plain → SSL** : lit en clair depuis le client local et écrit chiffré vers le tunnel server.
    2. **SSL → plain** : lit déchiffré l’ACK depuis le serveur et renvoie en clair.
  + **Clôture propre** : Dès qu’un flux se termine, on envoie un FIN TLS (via shutdownOutput()), on attend la fin de l’autre relais, puis on ferme proprement les deux sockets en journalisant en **WARNING** toute exception survenue lors de la fermeture.
* Packaging Docker :
* Le Dockerfile Dockerfile.tunnel-client est multi-stage basé sur maven:3.9-eclipse-temurin-8 pour compiler, puis tomcat:9.0-jdk8-temurin pour exécuter.
* ENV CONFIG\_DIR=/tunnelConfig (où est monté le truststore et le fichier parameters.xml).
* Le conteneur expose **port 2200**.

## 7.3 tunnel-server :

Le **tunnel-server** accepte des connexions TLS entrantes et les relaie vers le service final app-server en TCP non chiffré. Il se compose principalement de :

* **parameters.xml** : fichier de configuration des paramètres (ports, hôtes, keystore/truststore, logging).
* **ConfigLoader.java** : lecture et validation du XML.
* **TunnelServer.java** : initialisation (logger, SSLContext) et gestion des connexions SSL→plain.

### 1. parameters.xml

Définit la configuration attendue par ConfigLoader :

* Les balises <localPort>, <remoteHost>, <remotePort> pilotent l’ouverture des connexions locale et distante.
* <keystoreFile> / <keystorePassword> : fichier JKS et mot de passe pour le certificat et la clé privée du serveur, tandis que <truststoreFile> / <truststorePassword> (optionnels) activent l’authentification mutuelle.
* Les paramètres de logging (<logDirectory>, <logFileName>, <logLevel>) sont lus pour configurer le logger de TunnelServer.

### 2. ConfigLoader.java

Cette classe parse parameters.xml et expose tous les paramètres sous forme de champs public final :

* **Paramètres obligatoires**
  + localPort, remoteHost, remotePort pour l’écoute SSL et la connexion finale.
  + keystoreFile, keystorePassword pour charger le JKS du serveur.  
    Toute absence ou valeur vide génère une IllegalArgumentException.
* **Paramètres optionnels**
  + truststoreFile, truststorePassword lus uniquement si truststoreFile est défini ; autrement, mutual TLS reste désactivé.
* **Logging**
  + logDirectory, logFileName, logLevel lus ou remplacés par leurs valeurs par défaut, garantissant un comportement cohérent sans configuration explicite.

Chaque champ de **ConfigLoader** correspond directement à une balise de **parameters.xml**, assurant que toute modification du XML se répercute immédiatement dans TunnelServer.

### 3. TunnelServer.java

La classe TunnelServer orchestre : chargement de la config, configuration du logger, initialisation du SSLContext, puis boucle d’écoute et relais des connexions SSL→TCP.

1. **Chargement de la configuration**

Lecture de parameters.xml via **ConfigLoader** (répertoire ./tunnelConfig par défaut, modifiable via -DconfigDir). Toute erreur de parsing ou de valeur manquante arrête immédiatement le programme.

1. **Logger**

Création du répertoire cfg.logDirectory si nécessaire, et ajout d’un *FileHandler* en mode append vers <logDirectory>/<logFileName>, formaté en *SimpleFormatter*, niveau cfg.logLevel.

1. **Initialisation du SSLContext**

Appel à createSSLContext() pour charger le KeyStore (obligatoire) pour l’identité du serveur et le TrustStore (optionnel) pour l’authentification mutuelle, et construire un SSLContext configuré pour TLS 1.2.

Toute exception (fichier manquant, mot de passe erroné…) est loguée en *SEVERE* et interrompt l’exécution.

1. **Boucle d’écoute et relais des connexions**

Un SSLServerSocket s’ouvre sur cfg.localPort. À chaque accept(), un thread dédié (server-handler-<port>) est lancé pour :

handleConnection(sslClient, cfg.remoteHost, cfg.remotePort).

Dans handleConnection :

* **Handshake TLS** : mode serveur, startHandshake() pour sécuriser la connexion entrante.
* **Timeout** : 30 000 ms sur la socket SSL pour éviter les blocages.
* **Connexion finale** : ouverture d’une socket TCP non chiffrée vers cfg.remoteHost:cfg.remotePort, également avec timeout de 30 000 ms.
* **Relais bidirectionnel** : deux threads copient simultanément :
  + - 1. *SSL → plain* : lit les octets chiffrés depuis le client TLS, écrit en clair vers le service final.
      2. *plain → SSL* : lit les réponses non chiffrées du service, réécrit chiffré vers le client TLS.
    - **Demi-fermeture**: après fin du flux *SSL → plain*, envoi d’un FIN sur la connexion plain, attente du flux *plain → SSL*, puis log de la fin du tunnel.
    - **Fermeture propre** : dans un bloc finally, fermeture des deux sockets
* Packaging Docker :
* Le Dockerfile Dockerfile.tunnel-server est multi-stage basé sur maven:3.9-eclipse-temurin-8 pour compiler, puis eclipse-temurin:8-jre pour exécuter.
* ENV CONFIG\_DIR=/tunnelConfig (où est monté le keystore et le fichier parameters.xml).
* Expose **port 2201**.

## 7.4 app-server

Le serveur final **app-server** est déployé sous la forme d’un WAR avec comme servlet principal **zetaSockHl7, qui initialise** ensuite **socketSrv** et **Watcher**.

### 1. web.xml

* **Servlet zetaSockHl7**
  + **Classe** : zetasocket.zetaSockHl7
  + **Load-on-startup** : 5
  + **Mappings** : /servlet/zetaSockHl7
* **Init-params** (dans <servlet>) :
  + debug : niveau de verbosité console (0,1,2…)
  + port : port TCP d’écoute HL7 (ex. 2200)
  + folder : répertoire de stockage des messages (ex. C:/def/messageStore)
  + endrecord : séquence de fin de message
  + timer : intervalle (en s) de la supervision par **Watcher**

### 2. zetaSockHl7.java

Le servlet **zetaSockHl7** est le point d’entrée et coordonne les deux sous-systèmes :

1. **init()**
   * Récupère les init-params de web.xml, démarre socketSrv, qui se charge de la couche TCP HL7.
   * Lance un thread **Watcher** pour la surveillance et les appels périodiques.
2. **service(...)**
   * Interprète le paramètre HTTP zs:fnc : Si end → appelle la méthode d’arrêt de socketSrv et renvoie <reply><status>Restart</status></reply>.  
     Sinon → répond <reply><status>Nothing to do</status></reply>.
3. **destroy()**
   * Interrompt proprement socketSrv au déchargement du contexte servlet.

### 3. socketSrv.java (et doComms)

Le service **socketSrv**, lancé par le servlet, crée et pilote un thread **doComms** pour gérer la logique HL7:

1. **Initialisation**
   * Lit port, timeout, debug, folder, endrecord, et (optionnel) host ou logfile depuis la servlet.
   * Crée le dossier de stockage (folder) si nécessaire.
2. **doComms.run()**

Ouvre un **ServerSocket** sur port, avec timeout et restriction d’hôte éventuelle, et pour chaque connexion entrante :

* **Lecture HL7** : identifie le début (VT/MSH) et la fin (FS ou endrecord) du message.
* **Stockage** : écrit le message dans <folder>/<messageId>.hl7 et crée <messageId>.ok.
* **Réponse ACK** : renvoie un message HL7 MSA|AA|<messageId> encadré par les mêmes délimiteurs.
* Gère la **demi-fermeture** et ferme proprement le socket, journalisant les erreurs selon debug.

1. **Arrêt contrôlé** **:** stopMe() et stopWorker() permettent à la servlet d’interrompre la boucle et de libérer le port.

### 4. Watcher.java

Le thread **Watcher**, lancé par la servlet, assure :

1. **Restart automatique**  
   Toutes les 5 s, vérifie si le thread **doComms** est vivant. Si non et qu’aucune demande d’arrêt n’est en cours, relance **socketSrv**.
2. **Appels périodiques**  
   Toutes les timer/5 itérations, effectue un **HTTP GET** vers urlact (si défini), pour notifier ou déclencher une action externe.
3. **Arrêt**  
   Quitte dès que la servlet demande l’arrêt (stopThread) ou si urlact est absent.

* Packaging Docker :
* Le Dockerfile Dockerfile.app-server est multi-stage basé sur maven:3.9-eclipse-temurin-8 pour compiler, puis tomcat:9.0-jdk8-temurin pour exécuter.
* Conteneur expose **port 8080** (Tomcat) et **2200** (MLLP).
* Volume monté /var/messageStore pour stocker les messages HL7 côté hôte dans C:\def\messageStore  
  et fichier logs /var/log/tunnel-logs/logs\_tunnel-client.log relié côté hôte à C:\def\tunnel-logs