

Projet XML : Service REST & Client

Sébastien Guerrier, Léo Lovicourt

Mai 2023



Table des matières

1	Introduction	3
2	Service Rest	3
2.1	Pages d'accueil et d'aide	3
2.2	Persistance des données & annotations	4
2.3	Liste des STB	5
2.4	Détail d'une STB	5
2.5	Insertion d'une STB	5
2.6	Suppression d'une STB	6
2.7	Déploiement et captures d'écran	6
2.8	Tests	11
2.8.1	Tests unitaires	11
2.8.2	Tests Postman	11
2.9	Notice de déploiement de l'API	11
3	Client	12
3.1	Architecture du client	12
3.2	Notice d'utilisation du client	14
3.3	Client et captures d'écrans	15
4	Difficultés rencontrées	30
5	Conclusion	30

1 Introduction

L'objectif du projet est le déploiement d'un service REST permettant de gérer les documents STB23, conformes à la description établie lors du premier TP de XML. Il s'agit ensuite de créer un client permettant l'envoi et la consommation de documents STB23 avec le service REST.

Tous les échanges avec le service REST doivent utiliser des formats valides, c'est-à-dire du HTML ou XHTML dans le cas d'affichage, ou du XML pour les échanges d'informations. De plus, les flux reçus par le service devront être validés par le schéma XSD de description STB23. Cependant, en cas d'erreur de traitement ou de non conformité du flux XML, le processus de traitement sera arrêté, aucune modification de contenu ne sera stockée, et une réponse sera retournée avec les indications d'erreur adéquates.

Le client doit être une application autonome exécutée sur une machine locale, sans serveur. Il doit être "ergonomique" et permettre d'utiliser un flux STB23 conforme. Les fonctionnalités offertes doivent également être conformes à la description du service REST. La transmission d'une STB23 doit se faire selon au moins l'une des 2 méthodes suivantes :

- Saisie directe par l'utilisateur dans l'IHM de l'application
- Sélection d'un fichier XML stocké sur la machine locale

Enfin, après chaque échange avec le service REST, les informations retournées devront être affichées, et aucune validation de flux XML n'est imposée sur le client.

2 Service Rest

Le code source du service Rest est disponible au [lien suivant](http://app-4f5276a2-31f8-4d2a-9a9c-de10b52d5fd4.cleverapps.io), et il est possible d'accéder au service Rest grâce aux URL <http://app-4f5276a2-31f8-4d2a-9a9c-de10b52d5fd4.cleverapps.io> et <https://stb23-sedixed.cleverapps.io>. Si jamais les invitations au le dépôt distant ne sont pas accessibles par mail, vous pourrez les trouver au [lien suivant](#). Pour finir, la javadoc de l'API est disponible dans le répertoire `javadoc`.

Conformément à ce qui a été préalablement réalisé en TP, le service Rest est basé sur un projet Spring Boot. Nous avons donc réutilisé et adapté le code existant aux spécifications fournies dans le sujet de projet. De même, nous avons réutilisé (et raffinés lorsque nécessaires) les fichiers XSD et XSLT produits lors des séances de TP.

2.1 Pages d'accueil et d'aide

Le contrôleur responsable de ces 2 endpoints est `IndexController`. Il contient les 2 méthodes `index` et `help`, permettant respectivement de renvoyer le flux HTML de la page d'accueil et celui de la page d'aide.

Pour cela, nous avons créé la classe `FileLoader` dans le package `util`, fournissant une méthode statique permettant de renvoyer une chaîne de caractères à partir du contenu d'un fichier HTML situé dans le répertoire `resources/html`. Nous avons par conséquent créé les fichiers statiques `index.html` et `help.html` dans ce répertoire, ayant chacun le contenu spécifié dans l'énoncé (page d'accueil et page d'aide).

2.2 Persistance des données & annotations

Afin de mettre en place la persistance des données, nous avons créé une base de données MySQL dans laquelle stocker nos STB. Pour ce faire, nous définissons les variables adéquates dans le fichier `resources/application.properties` (datasource url, username, password). On y précise également le driver utilisé (JDBC) et la politique de mise à jour du schéma de la base de données. Nous définissons cette politique avec la variable `spring.jpa.hibernate.ddl-auto=update`, permettant la création des tables si nécessaire au lancement de l'application, ainsi que le rafraîchissement du schéma lorsque celui-ci est modifié. Il n'y a donc pas besoin de créer au préalable la base de données et ses tables.

Nous avons par la suite défini les entités pour lesquelles créer une table en base de données : celles-ci sont situées dans le package `model`. On y retrouve donc les entités `STB`, `Client`, `Person`, `Team`, `Member`, `Features` et `Feature` : nous avons défini une entité pour chaque élément "complexe" de la STB, c'est-à-dire chaque élément comportant des sous-éléments. Chacune est annotée avec l'annotation `@Entity` afin d'indiquer qu'il s'agit d'une entité à prendre en compte au démarrage de l'application, pour effectuer les opérations nécessaires dans la base de données.

Pour chacune de ces entités, nous définissons un ID (sa clé primaire en base de données) auto-généré. Celui-ci est défini de la manière suivante :

```
1 @Id
2 @GeneratedValue(strategy = GenerationType.IDENTITY)
3 @XmlTransient
4 private Integer id;
```

On remarque également l'annotation `@XmlTransient` : cette annotation sert à indiquer que cet attribut n'est pas utilisé lors du `marshalling/unmarshalling` entre un flux XML vers une entité. En effet, les flux XML n'ayant pas d'élément `<id>`, l'opération échouerait.

Nous utilisons aussi les annotations afin de spécifier quels attributs d'une entité sont des éléments XML : pour cela, nous utilisons l'annotation `@XmlElement` lorsqu'il s'agit d'un élément, ou `@XmlAttribute` lorsque c'est un attribut.

Les relations entre toutes ces entités sont également définies par les annotations éponymes `@OneToOne`, `@OneToMany`, `@ManyToOne` et `@ManyToMany`.

Pour finir, nous définissons l'interface `STBRepository` dans le package `repository` : celle-ci hérite de l'interface `CrudRepository<STB, String>` et permet de simplifier les opérations CRUD (Create, Read, Update, Delete) liées à la base de données en fournissant des méthodes simples à utiliser.

Nous avons donc fait le choix d'utiliser conjointement MySQL, JPA, JDBC et Hibernate pour la persistance des données. Nous avons choisi MySQL car c'est un SGBD que nous connaissons bien, et qui par conséquent n'a pas été difficile à utiliser rapidement. Nous avons ensuite choisi d'utiliser JDBC car nous l'avons également déjà manipulé, cependant ici nous avons fait le choix d'utiliser en plus de JDBC l'ORM Hibernate (implémentation de JPA) afin de simplifier l'écriture des opérations CRUD en base de données. De plus, Hibernate nous permet d'utiliser l'option `spring.jpa.hibernate.ddl-auto=update` dans le fichier `application.properties` : celle-ci permet de créer les tables automatiquement lorsqu'elles ne sont pas présentes, ou de les modifier si le schéma évolue au cours du développement du service. Ce choix nous a donc semblé pertinent puis-

qu'il nous a permis de mettre la base de données en place très rapidement, et d'avoir une persistance des données facile à maintenir et à faire évoluer.

2.3 Liste des STB

Les endpoints `/stb23/resume` et `/stb23/resume/xml` permettent respectivement de récupérer la liste des STB aux formats HTML et XML : ils sont définis dans le contrôleur `GetController`. Chacun d'eux fait appel au `STBService` que nous avons mis en place dans le package `service`. Il permet de centraliser les opérations effectuées autour des STB, en gardant un code concis dans les contrôleurs puisque la logique est déplacée dans le service.

Pour l'endpoint `/stb23/resume/xml`, nous avons défini la méthode `getXMLResume` qui permet de construire le flux XML voulu à partir des STB en bases de données. Pour l'endpoint `/stb23/resume`, nous utilisons le fichier XSLT `resources/xslt/stb23resume.xslt` afin de convertir le flux XML obtenu précédemment en flux HTML. Cette transformation est effectuée grâce à la méthode statique `xmlToHtmlStream` de notre classe `HtmlConverter`, qui utilise l'API de transformation XML (`javax.xml`).

2.4 Détail d'une STB

Les endpoints `/stb23/xml/{id}` et `/stb23/html/{id}` permettent respectivement de récupérer la STB d'identifiant `{id}` en base de données aux formats XML et HTML. Les méthodes `getXMLSpecification` et `getHTMLSpecification` se contentent également de faire appel au `STBService`, effectuant toute la logique nécessaire.

Pour l'endpoint `/stb23/xml/{id}`, la méthode du service utilisée est `getXMLFromStbId` : celle-ci tente de récupérer la STB d'identifiant donné en base de données. Si celle-ci n'existe pas, un message d'erreur au format XML est renvoyé. Sinon, on récupère le flux XML issu du `marshalling` de cette STB et on le renvoie.

Pour l'endpoint `/stb23/html/{id}`, nous utilisons le résultat obtenu par la méthode `getXMLFromStbId` pour le transformer grâce au fichier XSLT `resources/xslt/stb23.xslt`. Cependant, en cas d'erreur, nous utilisons le fichier XSLT `resources/xslt/stb23error.xslt` afin de convertir le flux XML d'erreur obtenu en flux HTML similaire.

2.5 Insertion d'une STB

L'endpoint `/stb23/insert` permet l'insertion d'une STB en base de données à partir du flux XML correspondant. Dans un premier temps, nous vérifions que celle-ci est conforme au schéma XSD défini en TP. Pour cela, nous disposons d'une classe `STBValidator` indiquant si le flux XML est conforme au schéma XSD `resources/xsd/stb23.xsd`. Ici, nous utilisons l'API de validation XML (`javax.xml`).

Si celle-ci n'est pas valide, le message d'erreur en XML est renvoyé. Sinon, on vérifie par la suite que cette STB n'est pas déjà présente (au sens de l'énoncé, c'est-à-dire titres, dates et versions identiques). Pour cela, on crée une entité STB à partir du flux XML (`unmarshalling`) et on utilise la méthode `isDuplicate` du service, permettant de déterminer si une telle STB existe déjà ou non.

Si cette STB existe déjà, le message d'erreur adéquat au format XML est renvoyé. Sinon, nous faisons appel à notre méthode `addSTB` du `STBService` qui réalise l'insertion de celle-ci en base de données, et renvoyons un message de succès pour l'insertion.

2.6 Suppression d'une STB

L'endpoint dédié à la suppression d'une STB est `/stb23/delete/{id}`. Il permet de supprimer la STB d'identifiant donné par `{id}`. La méthode réalisant cet endpoint est définie dans le contrôleur `DeleteController`. Il effectue un simple appel au `STBService`, en essayant de supprimer la STB d'identifiant `{id}`. Si une telle STB existe, un flux XML confirmant sa suppression est renvoyé, autrement c'est un flux XML d'erreur.

2.7 Déploiement et captures d'écran

Durant les séances de TP, nous avons déjà configuré CleverCloud afin de déployer le service REST à chaque push sur la branche main de notre dépôt. Dans la continuité de cette configuration, nous avons ajouté un add-on MySQL pour notre base de données de production. Nous avons ensuite défini les variables d'environnement nécessaires dans notre application (datasource url, username, password) à partir de ceux générés dans l'add-on.

ATTENTION : Nous avons remarqué qu'il y avait parfois des soucis avec l'add-on MySQL de Clever Cloud, qui faisait que l'API ne pouvait pas s'y connecter après un déploiement. Nous nous sommes assuré que cela n'arrive pas, jusqu'au dernier déploiement. Néanmoins, si cela se reproduisait tout de même pour on ne sait quelle raison (une erreur mentionnant "hibernate" dans la stacktrace de l'exécution de spring-boot pour l'API), simplement redémarrer le service Clever Cloud devrait le résoudre, n'hésitez pas à nous contacter dans ce cas.

Vous pouvez voir ci-dessous une succession de captures d'écran pour vous montrer ce que donne l'affichage de chaque route sur la version déployée du projet. Mais vous pouvez passer à la section suivante dans le sommaire si vous préférez voir tout ceci par vous-même.

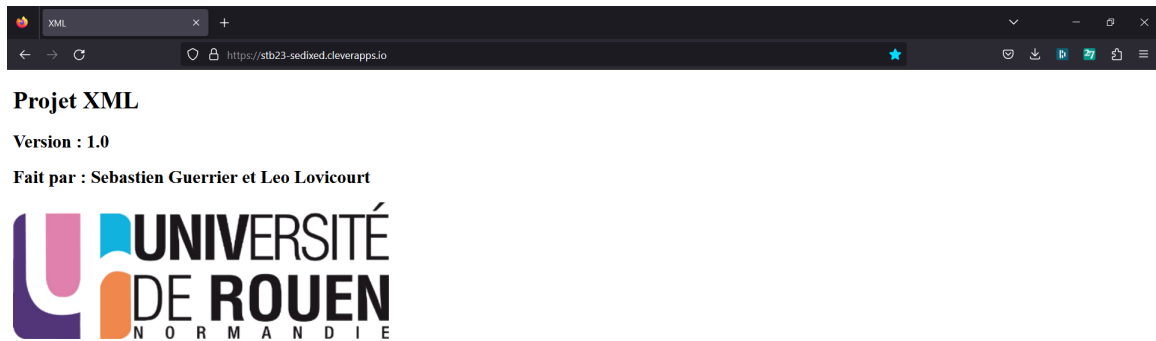


FIGURE 1 – Image de la route d'index

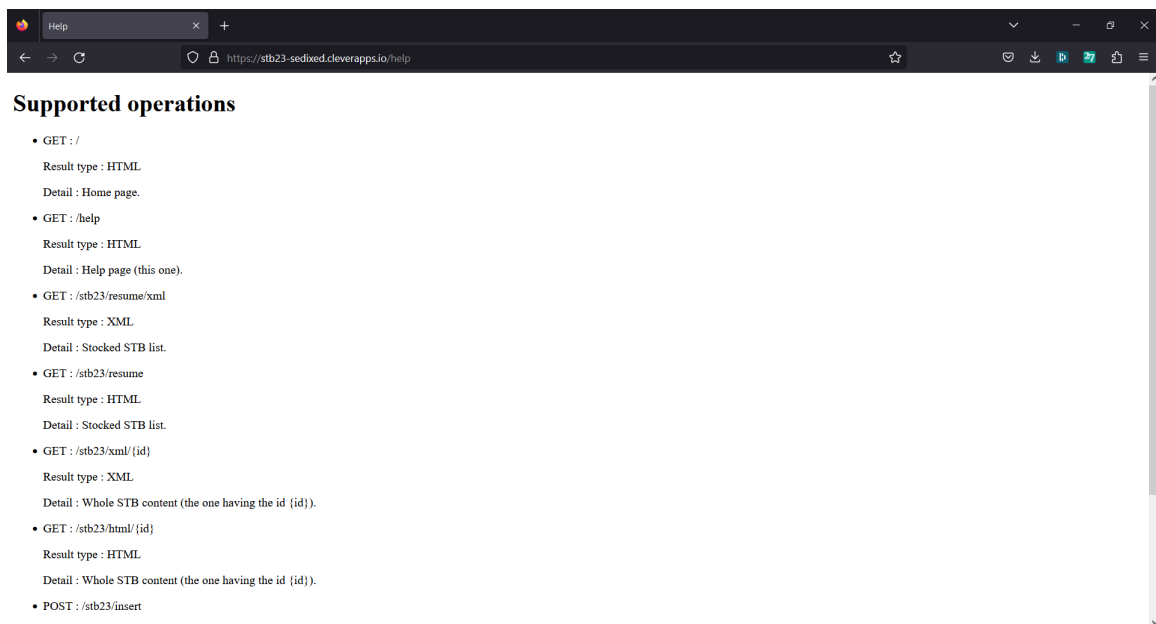


FIGURE 2 – Image de la route /help

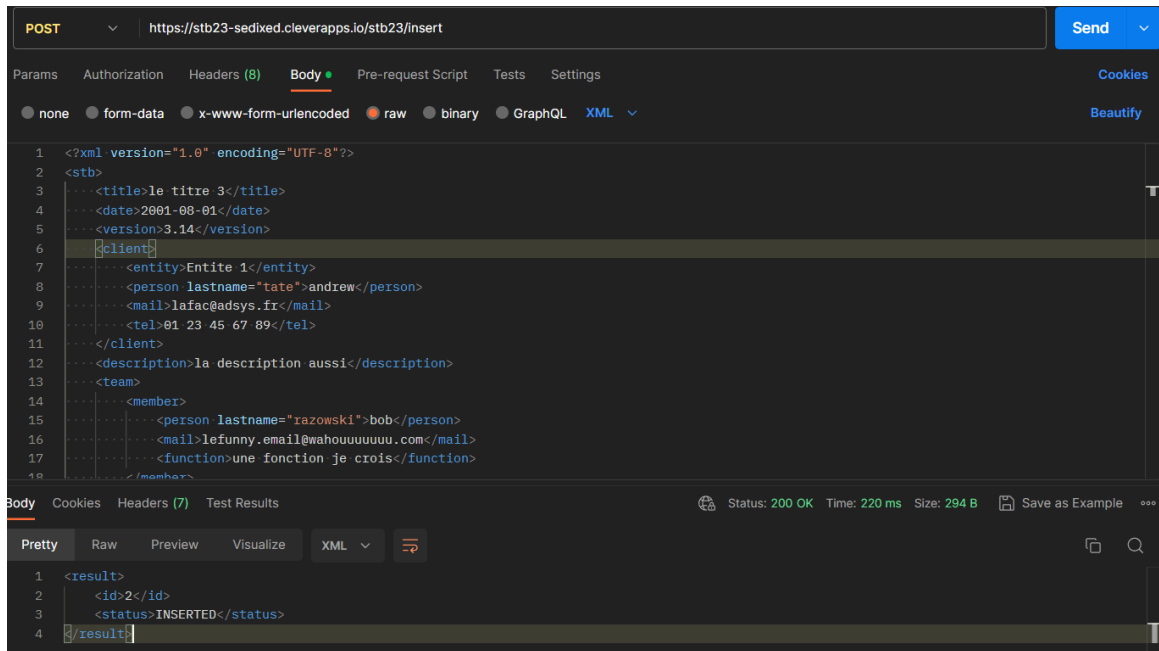


FIGURE 3 – Image de la route `/stb23/insert` (testé sur postman)



FIGURE 4 – Image de la route `/stb23/resume`

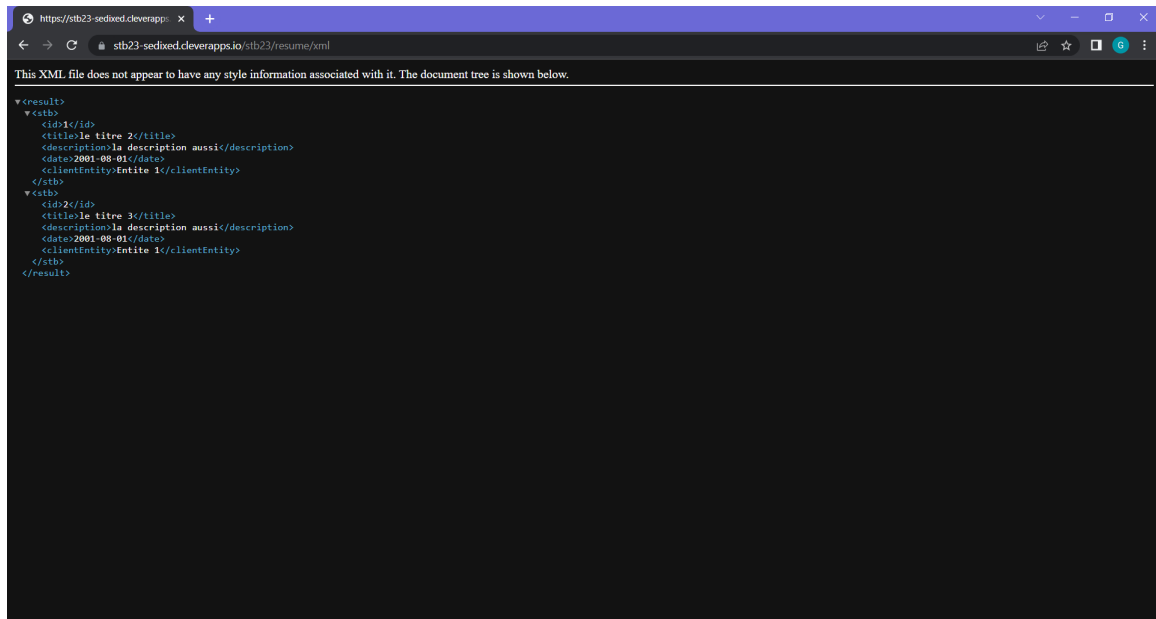


FIGURE 5 – Image de la route /stb23/resume/xml

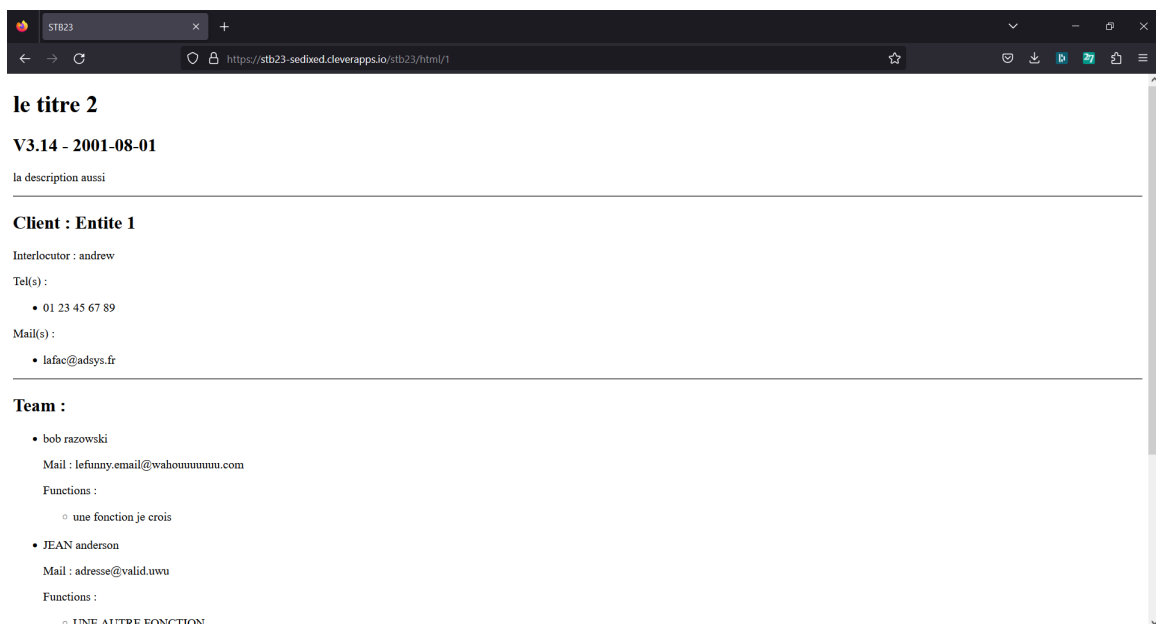


FIGURE 6 – Image de la route /stb23/html/1

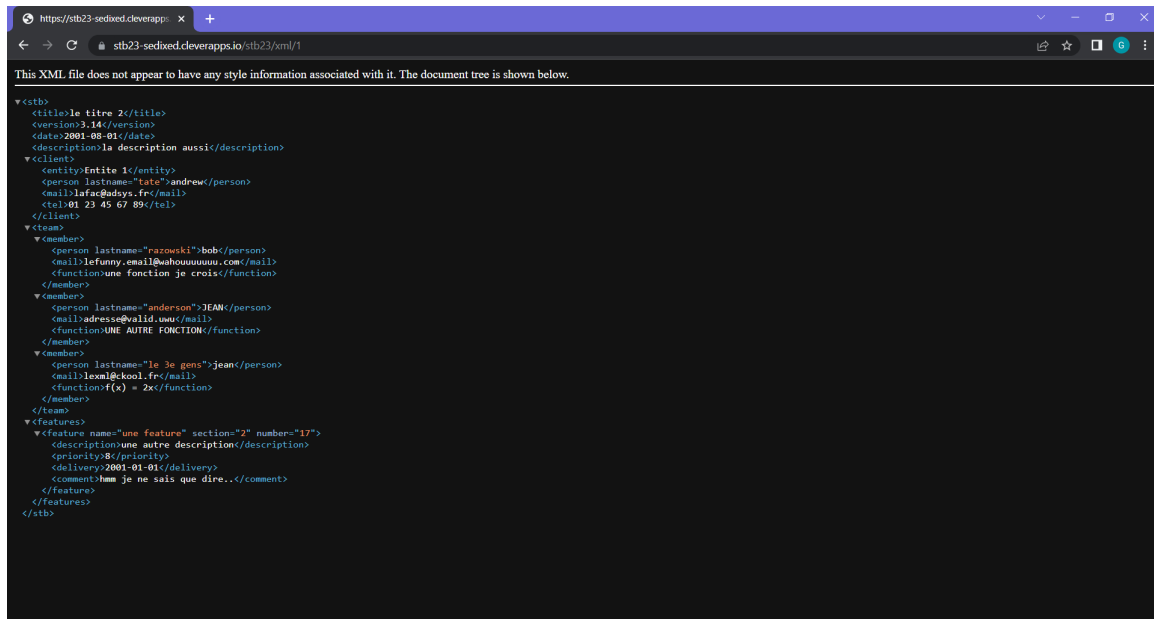


FIGURE 7 – Image de la route /stb23/xml/1

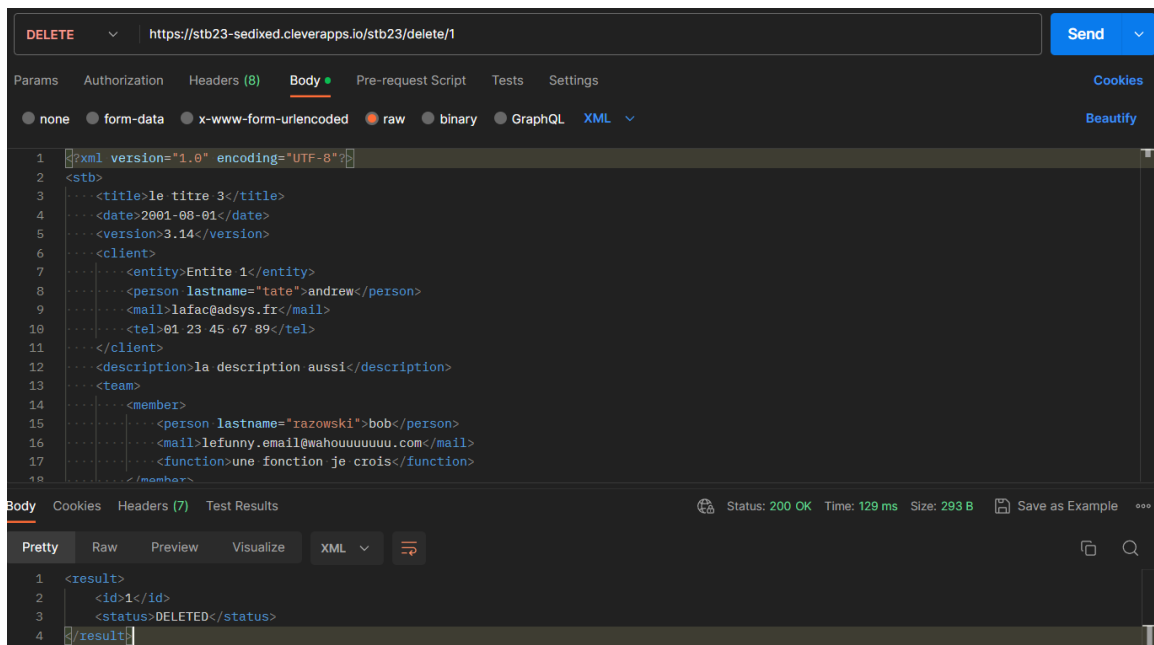


FIGURE 8 – Image de la route /stb23/delete/1 (testé sur postman)

2.8 Tests

2.8.1 Tests unitaires

Dans les commentaires liés à la correction des projets des années précédentes, vous avez mentionné la réalisation de tests unitaires lors de la phase de développement (JUnit). Cela aurait été un plaisir à faire sachant que nous avons beaucoup aimé les TP de tests logiciels sur le sujet, mais nous n'avons malheureusement pas eu le temps d'en réaliser

2.8.2 Tests Postman

Comme demandé dans le sujet, nous avons réalisé un jeu de requêtes Postman qui comporte toutes les requêtes. Le fichier JSON qui permet de récupérer la collection de ces requêtes de tests Postman se trouve dans le dossier **resources** du dépôt de l'API et à la racine de l'archive fourni sur Universitice, dans laquelle vous avez trouvé ce rapport.

2.9 Notice de déploiement de l'API

Pour déployer l'API en ligne, vous pouvez utiliser un service comme CleverCloud comme nous l'avons vu en TP mais également la démarrer en local : nous allons donc expliquer comment faire tourner l'API en local sur votre machine.

La première étape est de modifier le fichier `resources/application.properties` pour permettre à l'API d'utiliser votre propre SGBD. Pour ce faire, vous devez dans un premier temps modifier l'url, le nom d'utilisateur et le mot de passe, il suffira de remplir correctement ces informations (à condition d'avoir bien sûr votre SGBD qui est accessible), à l'endroit encadré sur la capture d'écran ci-dessous. Il faudra ensuite commenter les 3 lignes au-dessus qui définissent l'url, username et password pour CleverCloud, afin de pouvoir réutiliser sa base de données si besoin. Vous n'avez donc pas besoin de script de création de base de données comme nous l'avons expliqué dans la partie 2.2.

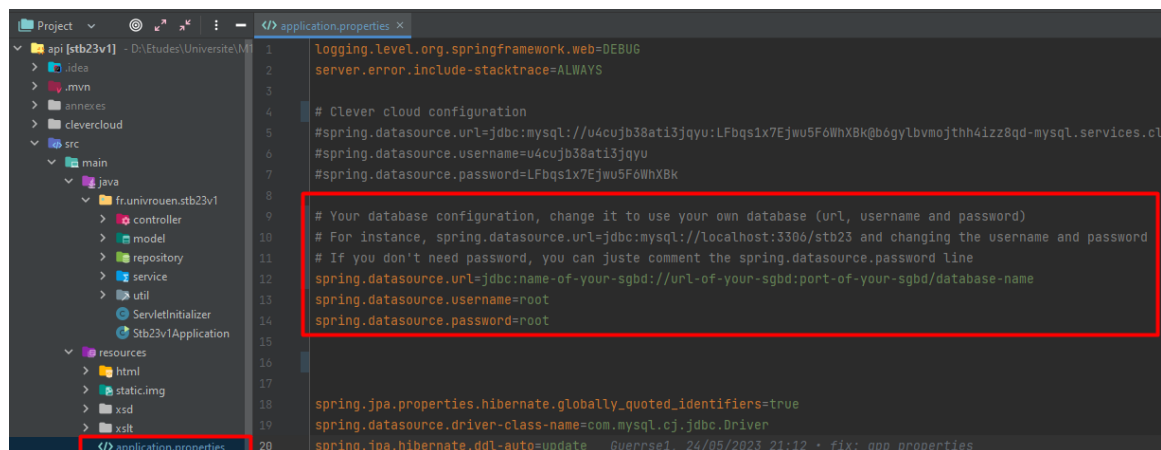


FIGURE 9 – Image du fichier `application.properties`

Ensuite, vous devez vous assurer de pouvoir utiliser Maven dans votre terminal ou un IDE comme IntelliJ et d'avoir installé sur votre machine ou un IDE comme IntelliJ [le SDK 17 de Java](#), comme nous l'avons fait en TP pour commencer le projet en utilisant Spring Boot.

Il vous suffira alors de vous rendre dans le dossier racine du code source de l'API après avoir cloné ou téléchargé le code source du dépôt, d'y ouvrir un terminal et de lancer les deux commandes suivantes (ne pas copier-coller car vous aurez des problèmes d'espacement) :

```
1 $: mvn install
2 $: mvn spring-boot:run
```

Vous devriez désormais avoir l'API démarrée en local, et pouvez lancer le client pour l'utiliser.

3 Client

Le code source du client est disponible au [lien suivant](#). Si jamais les invitations au le dépôt distant ne sont pas accessibles par mail, vous pourrez les trouver au lien [suivant](#). La javadoc du client est disponible dans le répertoire `javadoc`.

Comme aucune spécification n'était fournie dans le sujet à propos de celui-ci (hormis qu'il doit être exécuté sur une machine locale, sans serveur), nous avons choisi de réaliser une application Java en utilisant JavaFX (qui est une plateforme de développement d'interfaces graphiques (GUI) moderne et puissante pour les applications Java) pour l'interface de celle-ci. Nous avons également utilisé Maven pour gérer les dépendances du client et faciliter sa compilation avant utilisation, comme nous l'avons fait pour l'API avec Spring Boot en TP et durant ce projet.

Le choix d'utiliser JavaFX découle du peu de temps que notre groupe a pu accorder à la réalisation du projet, afin de s'assurer de pouvoir finir le client dans les temps. En effet, bien que nous ayons eu 3 semaines pour le réaliser, nous étions très pris par les révisions et les autres projets, mais nous avons heureusement réalisé une bonne partie du travail en TP. Nous avons donc décidé d'utiliser une technologie permettant de réaliser une interface flexible et éditée rapidement et simplement, pour faire un client fonctionnel et ergonomique dans les temps, tout en terminant l'API. Nous avons également envisagé de réaliser le client avec de simples fichiers HTML, CSS et JS comme certains camarades pour avoir un client rapidement, qui aurait vraisemblablement convenu aux exigences formulés pour celui-ci, mais nous n'étions pas sûr que cela que vous conviendrait.

3.1 Architecture du client

L'architecture mise en place pour le client est assez simpliste. Nous avons des fichiers FXML utilisés pour la vue du projet : ceux-ci sont mappés sur des contrôleurs (dans le package `controller`). C'est dans ces contrôleurs que nous définissons les actions associées à chaque élément de la vue (bouton, textarea, etc... comme dans la librairie Swing de Java) et que nous effectuons nos requêtes.

Nous avons géré la connexion au service Rest avec notre singleton `ConnectionSingleton` (dans le package `util`). Il permet de définir l'URI qui sera utilisée lors de l'envoi de la requête lorsqu'on valide la connexion à l'API dans l'onglet de configuration de l'application.

Nous avons également créé la classe `XmlFormater` permettant de formater un flux XML. En effet,

le flux XML reçu étant brut (tout sur une ligne), nous avons créé la méthode `formatXml` permettant de formater le XML reçu, en ajoutant des indentations où cela est nécessaire.

3.2 Notice d'utilisation du client

Exécuter le client se déroule presque de la même manière que l'exécution de l'API (voir 2.9). Vous allez devoir ouvrir un terminal dans le dossier racine du code source que vous aurez cloné ou téléchargé depuis le dépôt au préalable, puis lancer les commandes suivantes (ne pas copier-coller car vous aurez des problèmes d'espacement) :

```
1 $: mvn install
2 $: mvn javafx:run
```

Vous verrez apparaître une fenêtre d'application avec plusieurs onglets. Le premier onglet est dédié à la configuration de l'url et le port pour se connecter à l'API. Si par exemple vous voulez vous connecter à l'API que vous faites tourner en local, l'url que vous écrirez ressemblera à "http ://localhost" et le port "8080". Il faudra ensuite cliquer sur le bouton "Valider et tester" pour valider l'url qui prefixera les routes de l'API pour la suite de l'utilisation du client. Si vous ne configurez pas correctement l'url et le port, cela ne sert à rien d'essayer les autres fonctionnalités du client, comme sur l'image ci-dessous :

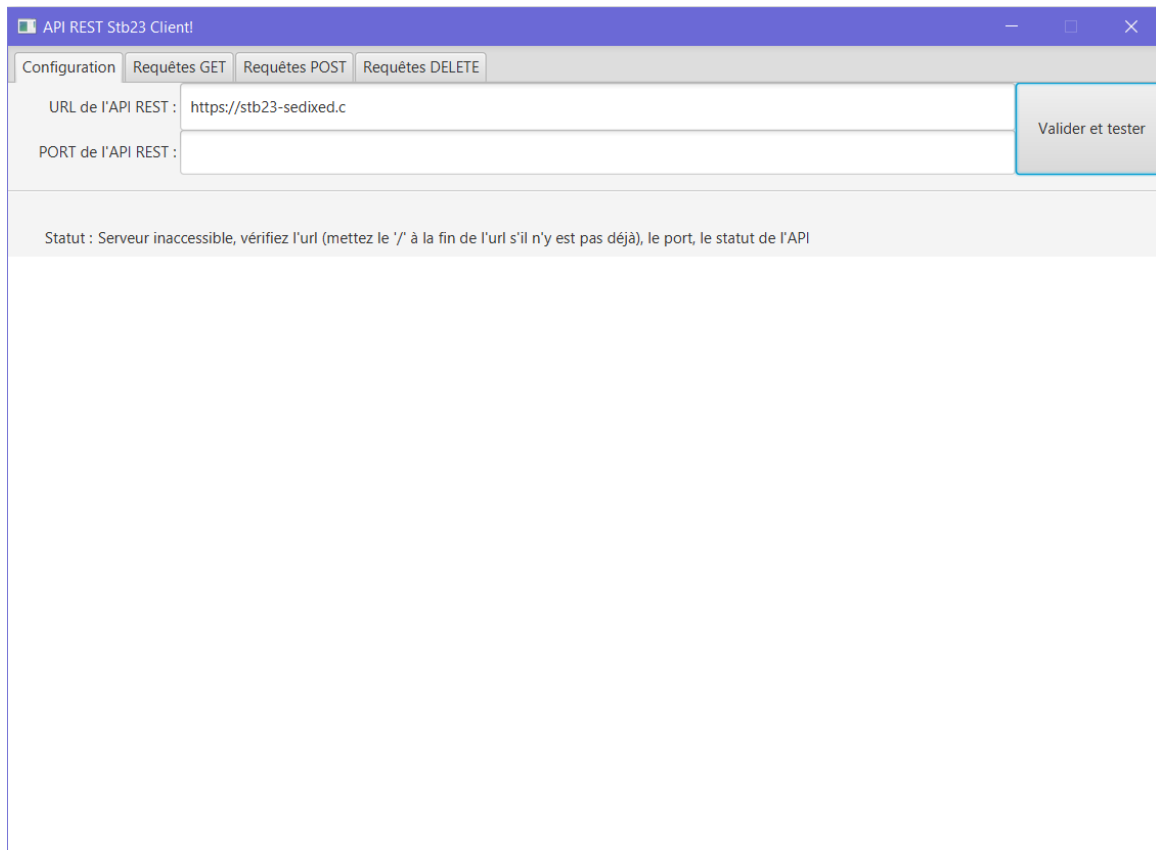


FIGURE 10 – Image d'une tentative de connexion à l'API ratée à cause d'une mauvaise combinaison d'url et de port

Assurez-vous que l'API soit accessible, que vous entrez la bonne url et le bon port (dans le cas

de notre API déployée sur CleverCloud, l'url sera "https://stb23-sedixed.cleverapps.io/" et le port vide). Vous verrez alors apparaître la page d'accueil de l'API en cas de succès comme ci-dessous :

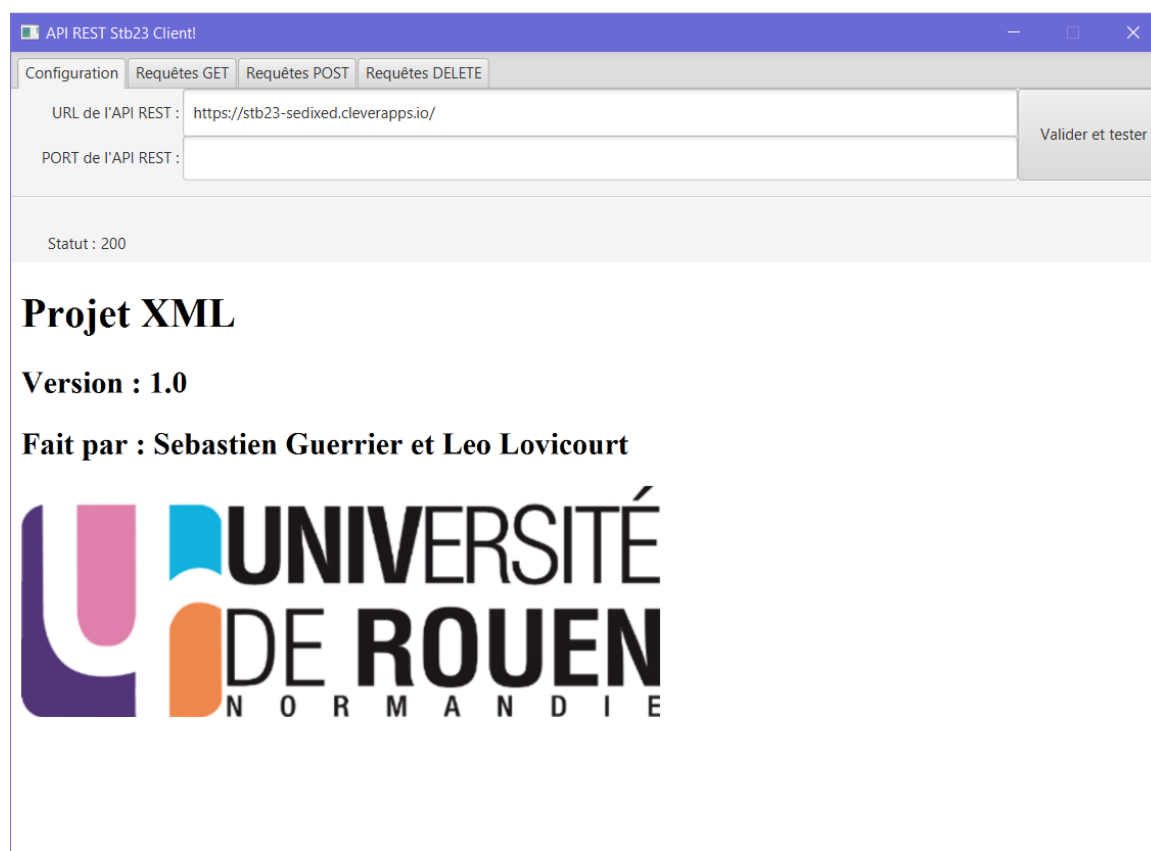


FIGURE 11 – Image d'une tentative de connexion à l'API réussie (on voit alors s'afficher la page d'accueil de l'API)

Vous pouvez désormais utiliser les différentes routes de l'API, qui sont séparés dans des onglets (un onglet pour les routes des requêtes GET, celles des requêtes POST et celle de requête DELETE). Sur chaque onglet, vous pouvez cliquer sur les boutons de nom des routes et vous verrez la réponse de l'API à la requête, dans une zone en bas de la fenêtre dédiée, avec le statut de la réponse juste au-dessus, ou un message d'erreur personnalisé en cas de soucis, pour vous aiguiller. Pour certaines routes, vous devez préciser un identifiant dans une zone de texte (les routes avec un {id} à spécifier) et pour la requête POST (/stb23/insert), vous pouvez charger un fichier XML avec le bouton adéquat pour envoyer ce fichier ou écrire directement le flux XML dans la zone de texte dédiée et envoyer ce flux.

3.3 Client et captures d'écrans

Voici une successions de captures d'écran pour montrer tous les différents cas d'utilisation du client. Mais vous pouvez passer à la section suivante dans le sommaire si vous préférez voir tout ceci par

vous-même.

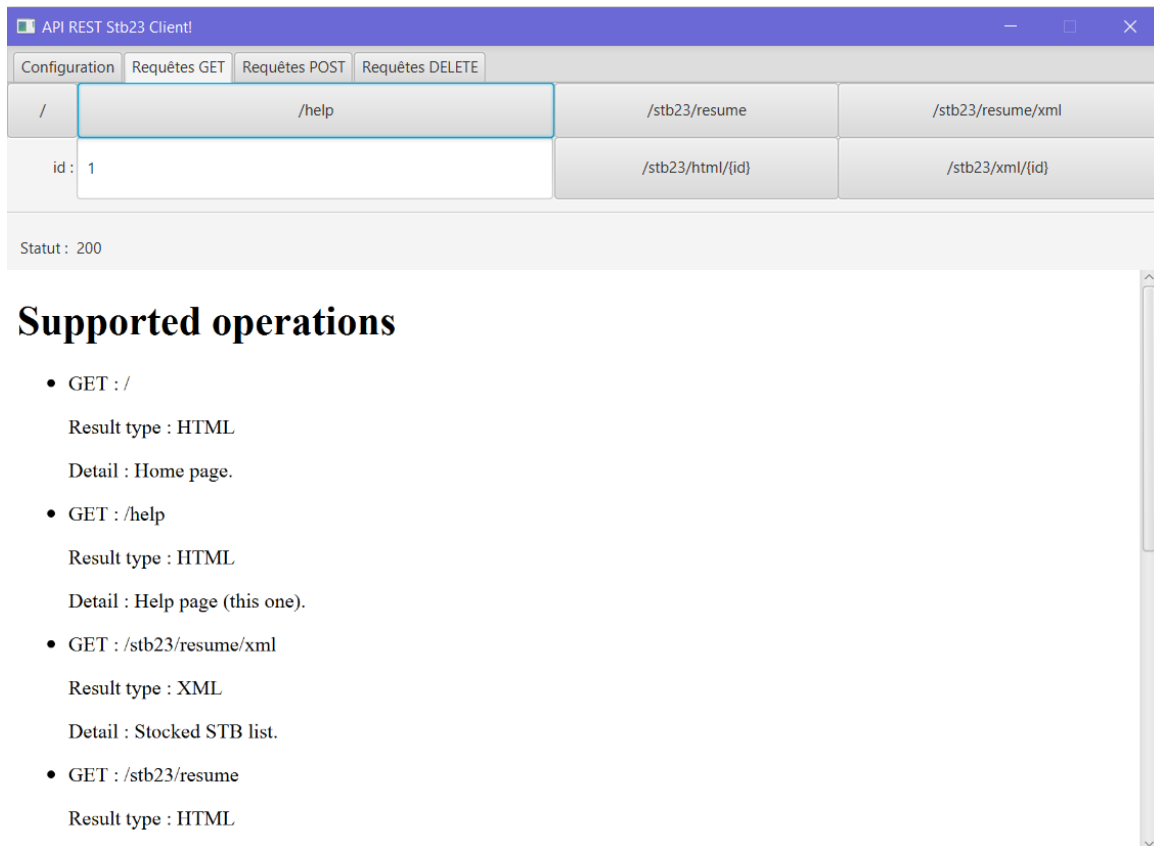


FIGURE 12 – Image de la route /help sur le client

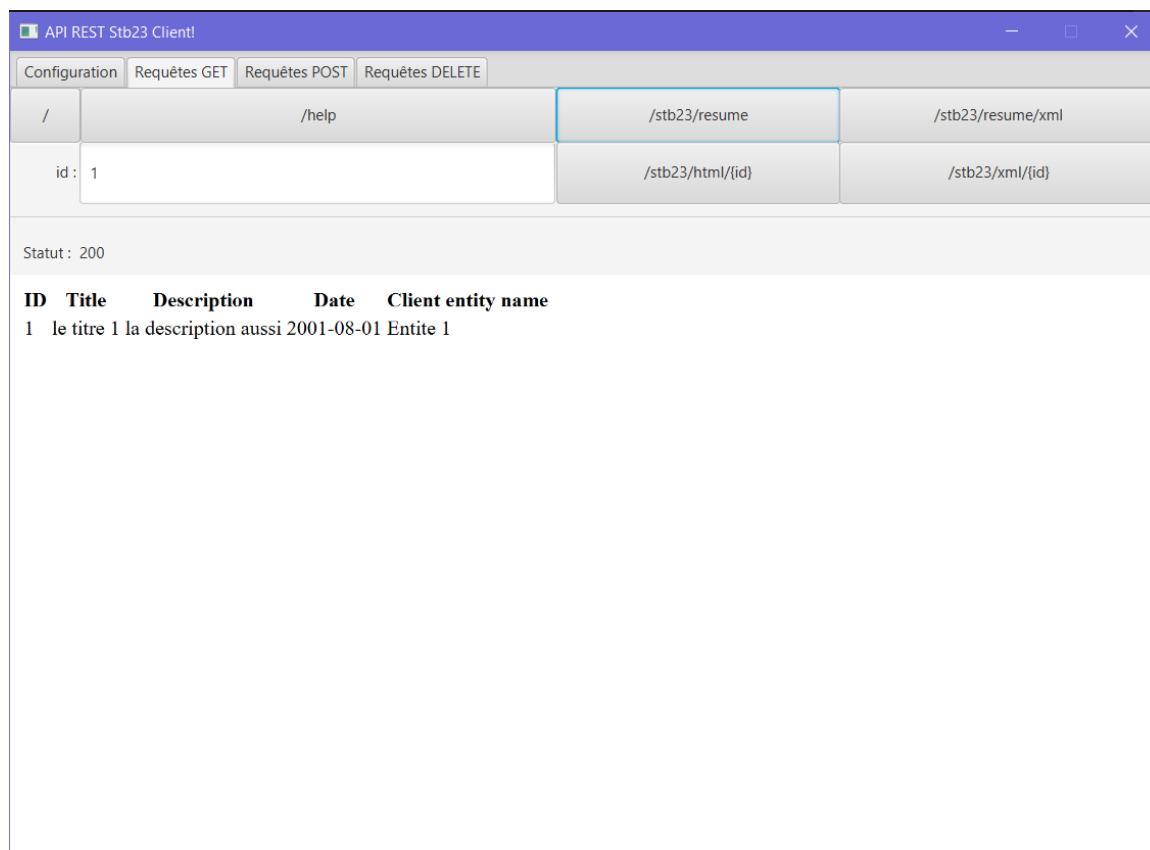


FIGURE 13 – Image de la route /stb23/resume sur le client

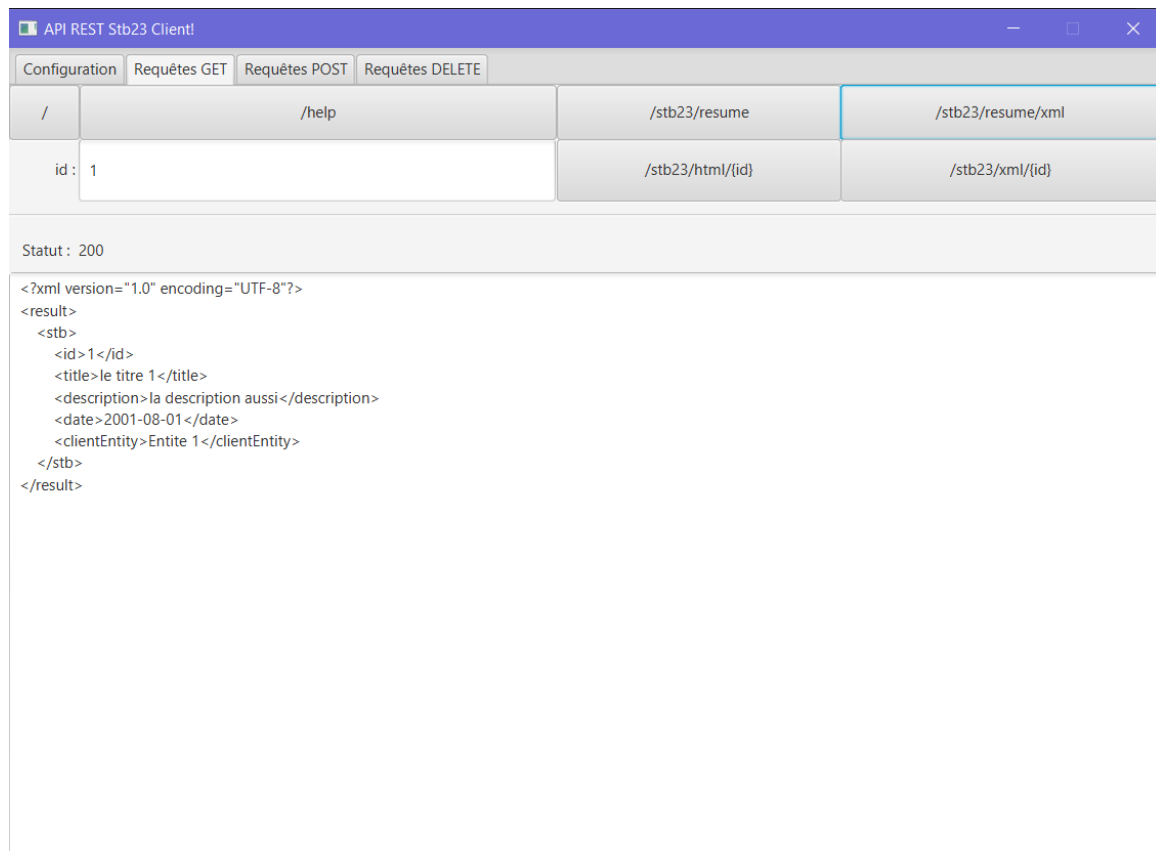


FIGURE 14 – Image de la route `/stb23/resume/xml` sur le client

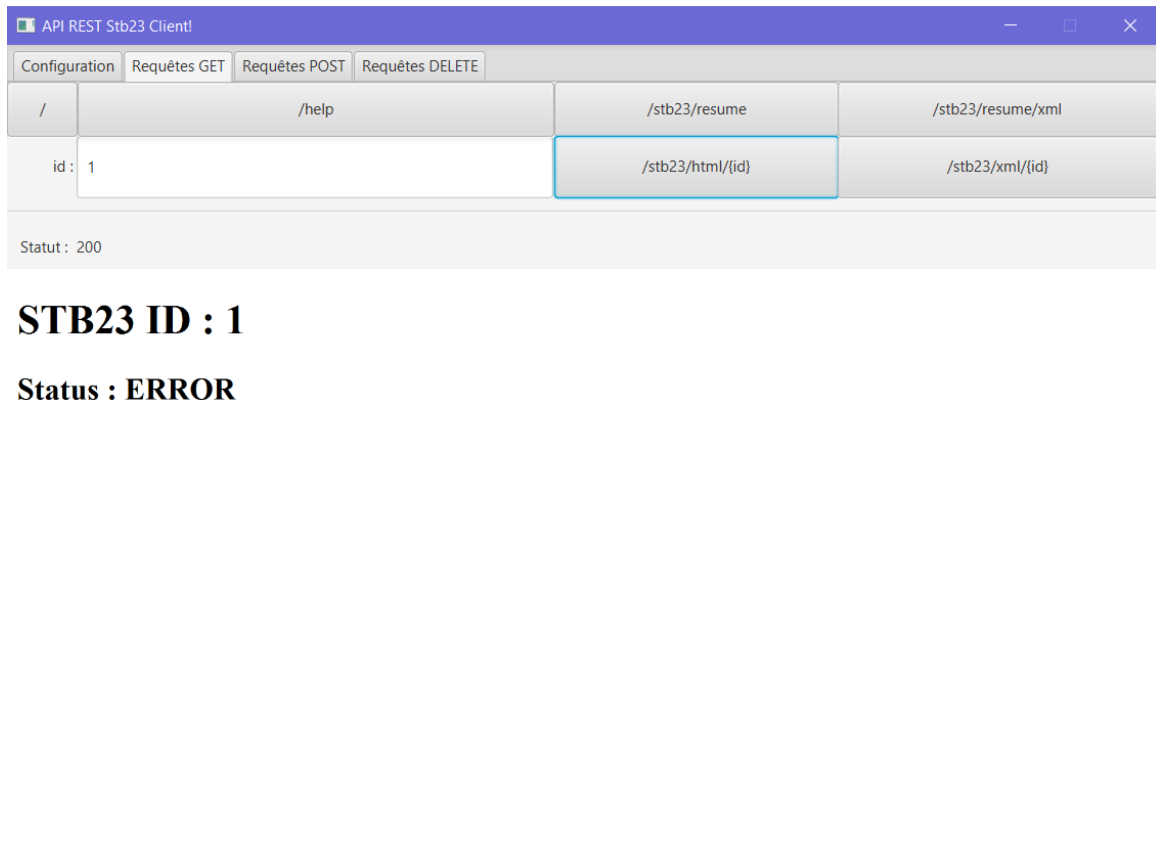


FIGURE 15 – Image de la route `/stb23/html/{id}` sur le client avec un identifiant invalide

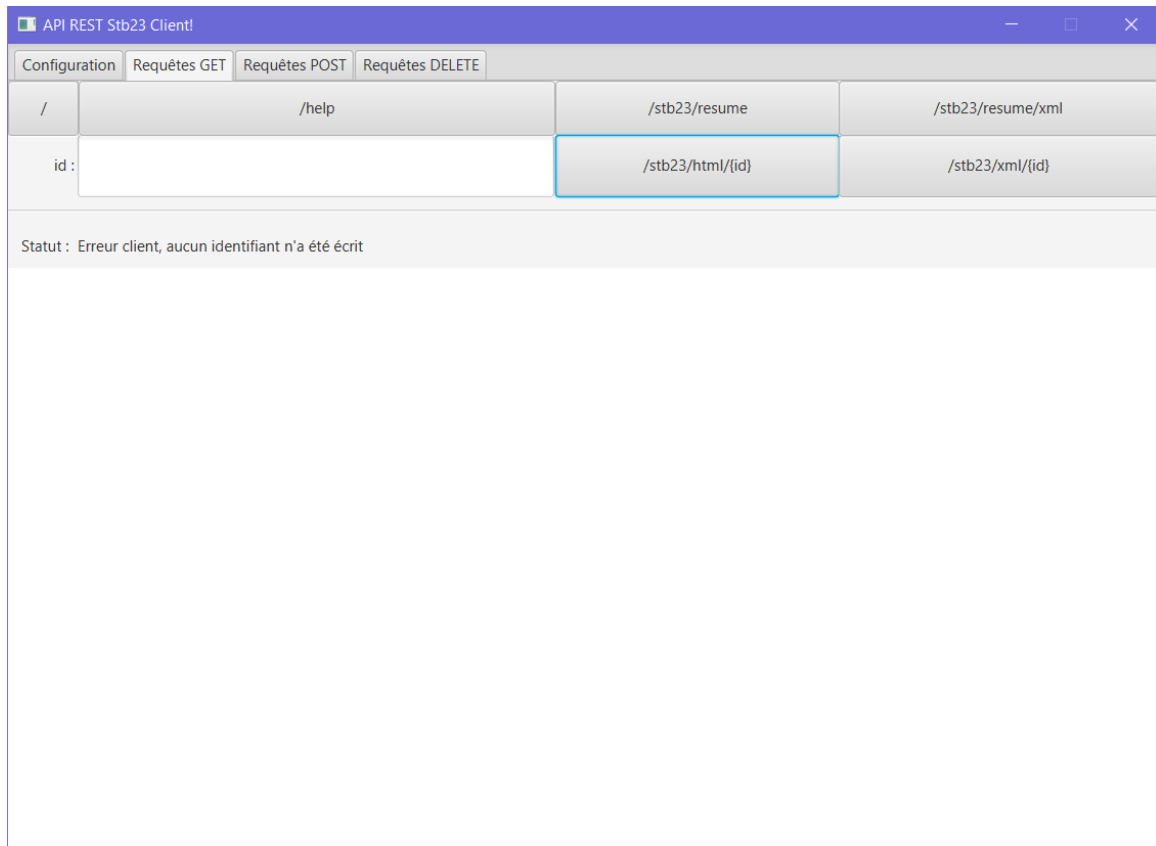


FIGURE 16 – Image de la route `/stb23/html/{id}` sur le client sans spécifier d'identifiant

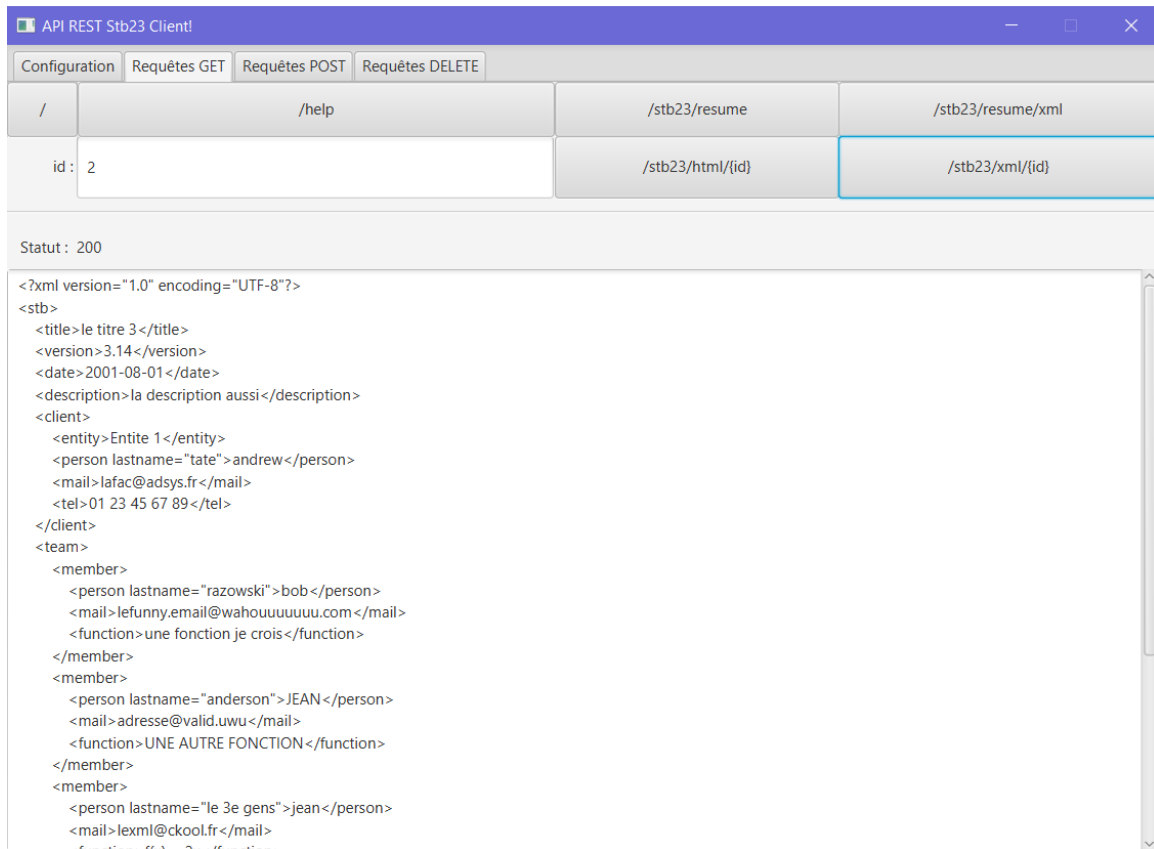


FIGURE 17 – Image de la route `/stb23/xml/{id}` sur le client avec un identifiant valide

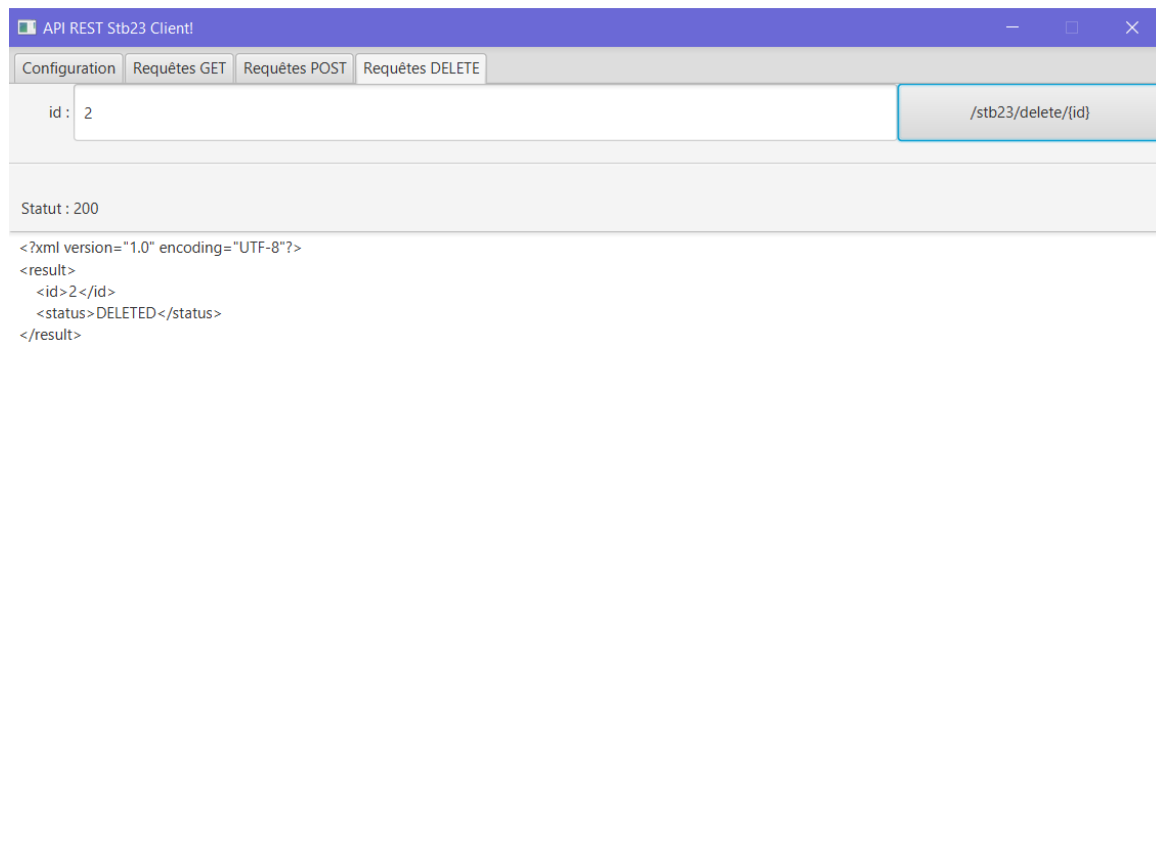


FIGURE 18 – Image de la route `/stb23/delete/{id}` sur le client avec un identifiant valide

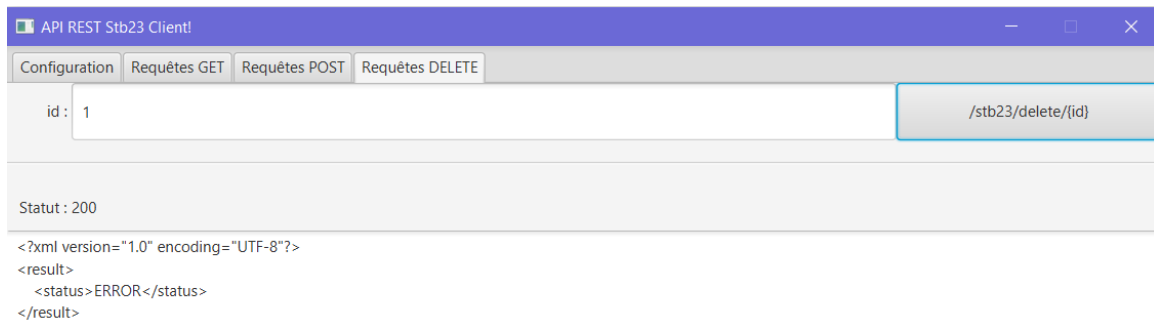


FIGURE 19 – Image de la route `/stb23/delete/{id}` sur le client avec un identifiant invalide

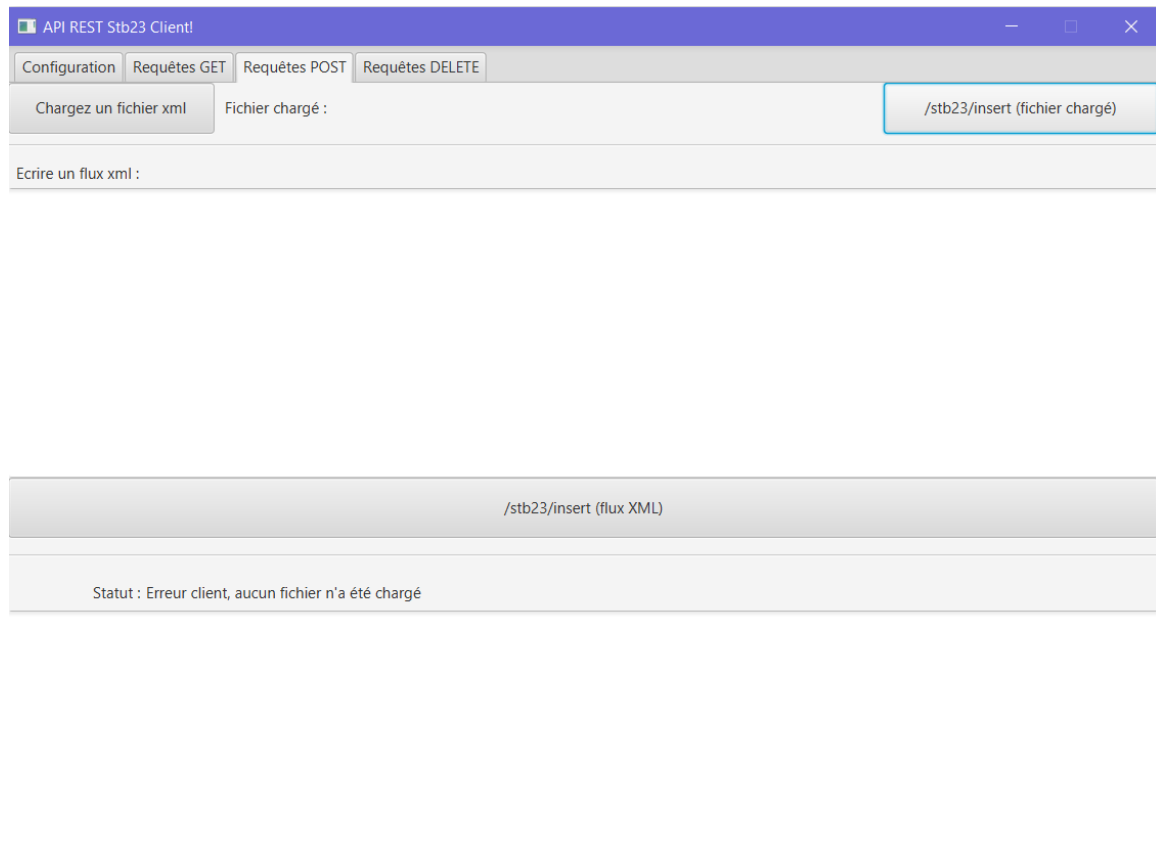


FIGURE 20 – Image de la route /stb23/insert sur le client sans charger de fichier xml

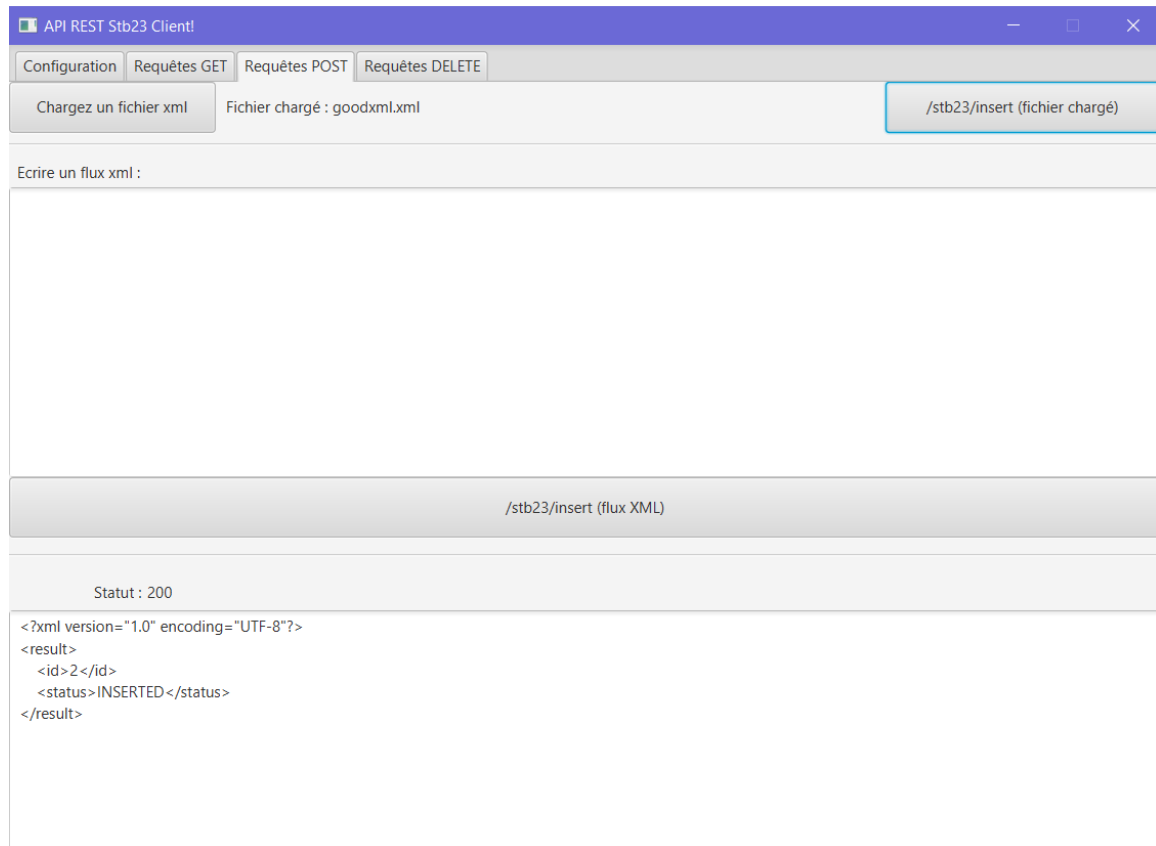


FIGURE 21 – Image de la route /stb23/insert sur le client avec un fichier xml d'une stb qui n'a pas déjà été insérée et valide

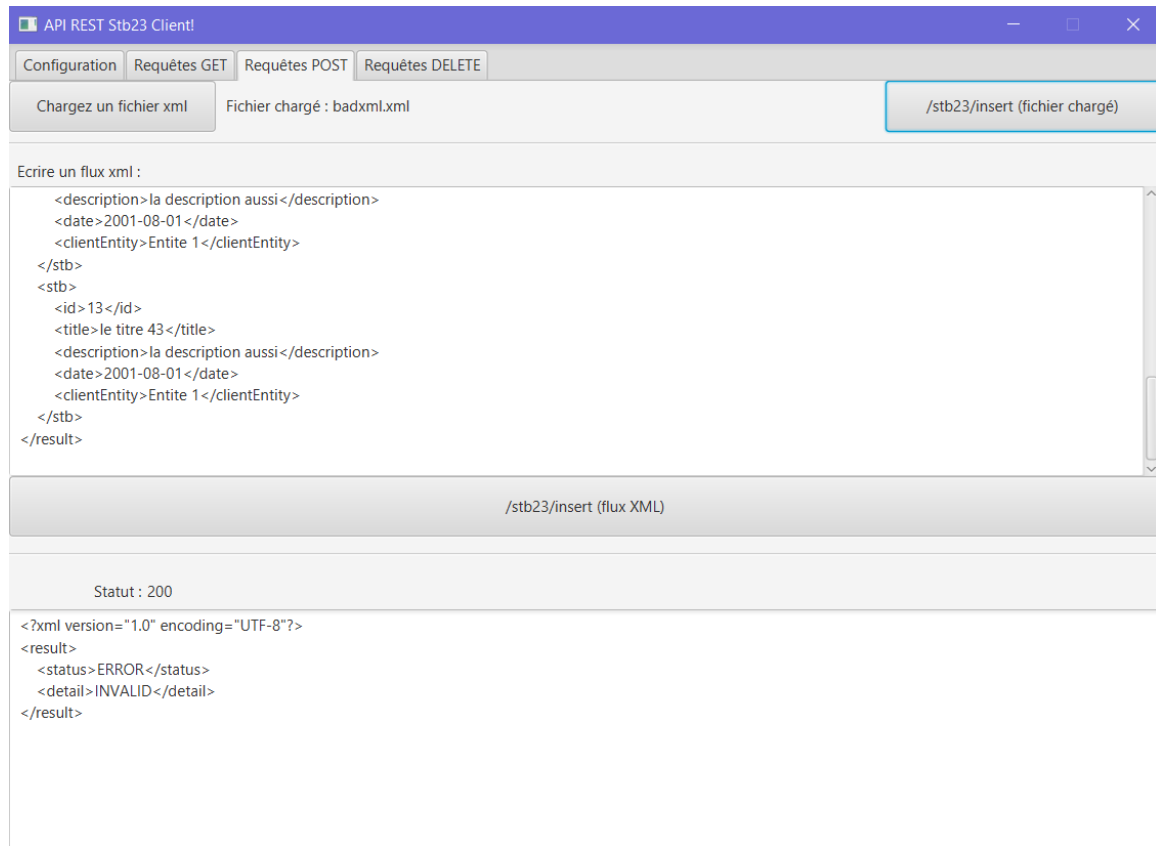


FIGURE 22 – Image de la route /stb23/insert sur le client avec un fichier xml d'une stb invalide

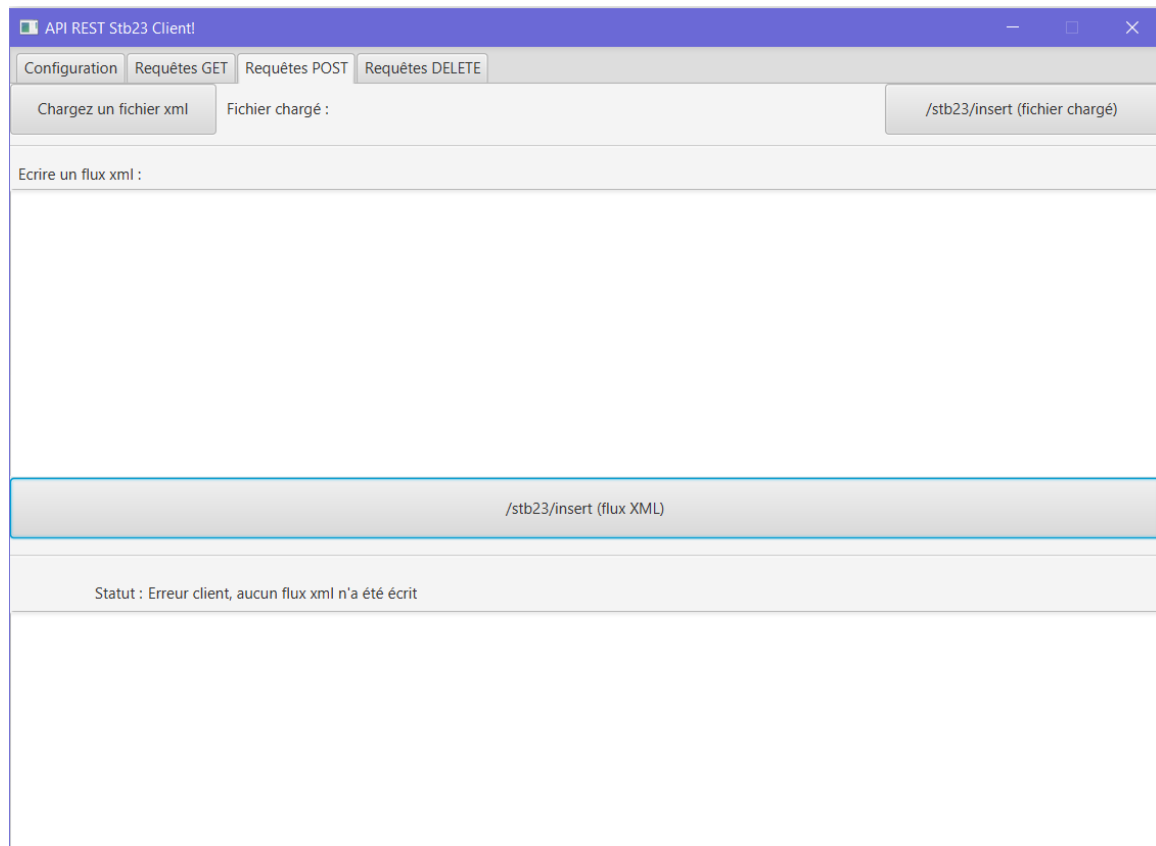


FIGURE 23 – Image de la route `/stb23/insert` sur le client sans écrire de flux xml

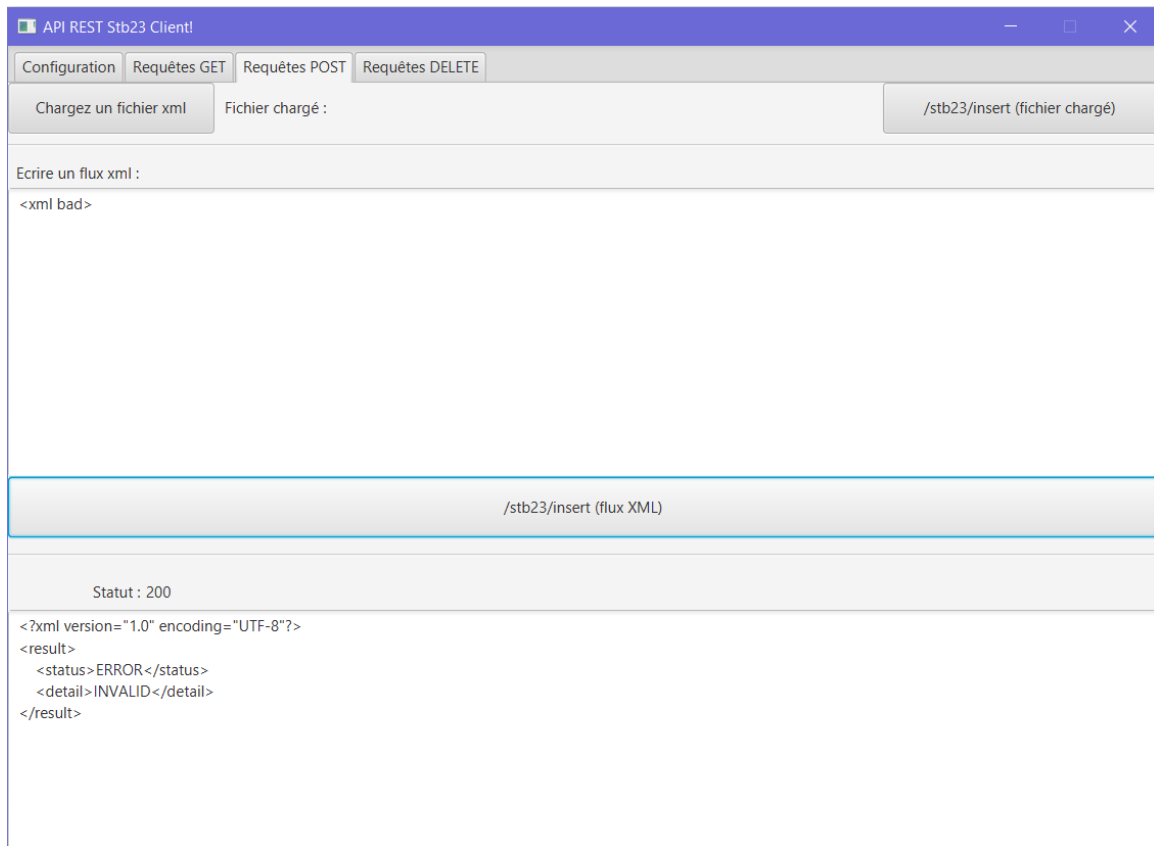


FIGURE 24 – Image de la route /stb23/insert sur le client avec un flux xml d'une stb invalide

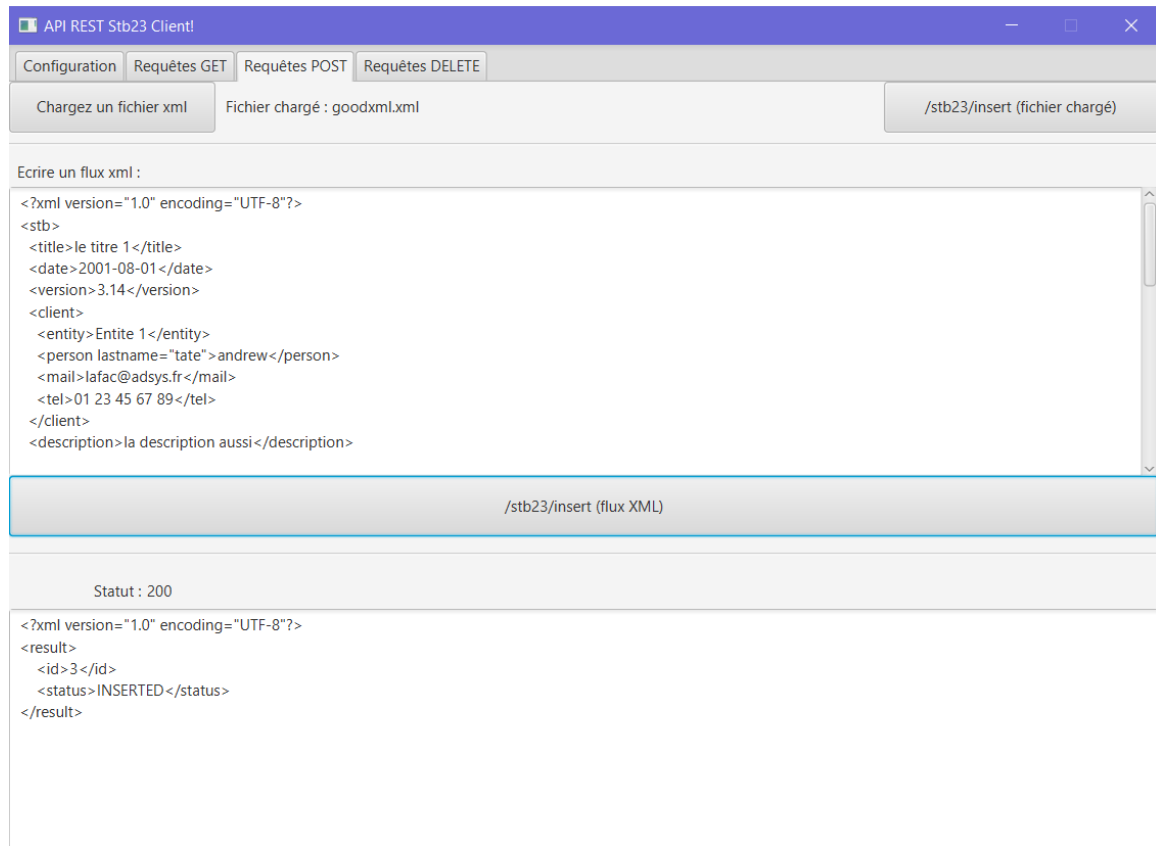


FIGURE 25 – Image de la route `/stb23/insert` sur le client avec un flux xml d'une stb qui n'a pas déjà été inséré et valide

4 Difficultés rencontrées

Nous avons rencontré plusieurs difficultés lors de la réalisation de ce projet. Premièrement, nous n'avons pas réellement compris ce qui était attendu pour le client (application Java, application web, simple fichier HTML effectuant des requêtes en JS), et ce même suite au mail envoyé afin de répondre aux questions posées par les étudiants. Nous avons donc décidé de réaliser une application Java respectant les quelques contraintes spécifiées.

Une autre difficulté a été l'utilisation des annotations fournies pour la persistance des données. En effet, nous avons eu du mal à comprendre comment "relier" les entités entre elles avec les annotations `@OneToOne`, `@OneToMany`, `@ManyToOne`, ou encore `@JoinColumn`. En plus de ces annotations, les paramètres de celles-ci étaient assez confus, ce qui nous a demandé un temps considérable.

5 Conclusion

Pour conclure, nous avons réalisé un service REST fonctionnel avec le cadriciel Spring ainsi qu'un client Java en utilisant la technologie JavaFX, pouvant envoyer toutes les requêtes traitées par le service à celui-ci, et avoir un résultat cohérent et lisible.