Rapport XML : STB23 V1

Jordan Elie

Mai 2023

Table des matières

1 Introduction

Le but de ce projet était de créer un service REST permettant de gérer les STB (spécification technique des besoins) de projet. Ce service devait donc implémenter des opérations d'insertion, de suppression, ou encore de visualisation de STB. Afin de pouvoir utiliser ce service, un client devait également être développé.

Afin de réaliser le service REST, nous devions utiliser le framework Java Spring Boot, celui-ci devait être ensuite être déployé sur Clever Cloud.

2 API REST

Le service REST se décompose en différents composants résumés sur ce schéma :

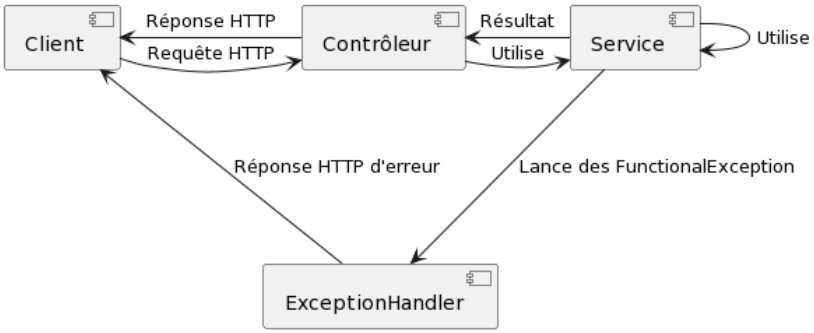

FIGURE 1 – Diagramme reliant les différents composants du service REST.

Ce schéma montre ici le déroulé des opérations effectuées lorsqu'un client envoie une requête sur le service REST :

- La requête arrive au contrôleur associé.
- Ce contrôleur fait appel à différents services afin d'exporter au plus possible la logique associée au traitement de la requête.
- Ces services peuvent également utiliser d'autres services (de plus bas niveau généralement) afin de réaliser leurs opérations.
- En cas d'exception, le service va la capturer et renvoyer une `FunctionalException` à la place.
- Ce type d'exception sera capturé par le `ExceptionHandler` (`FunctionalExceptionHandler`).
- La réponse sera ensuite envoyé au client, soit par le contrôleur, soit par l'`ExceptionHandler`. Il est important de noter qu'au sein de l'application que les réponses sont modélisées par des `DTO` (Data Transfer Object) pour plus de clarté.

Par la suite nous allons étudier plus en détail ces différents composants.

2.1 Contrôleurs

Au sein de l'application on retrouve deux classes contenant les différents contrôleurs de l'application :

- `HomeController` : Cette classe contient les deux endpoints `/` et `/help`, ceux-ci étant des endpoints racines il a été choisi de les regrouper dans la même classe.
- `STBController` : Cette classe contient tous les contrôleurs liés à la manipulation de STB. On y retrouve donc les 4 contrôleurs `GET`, le contrôleur `POST` ainsi que le contrôleur `DELETE`.

2.2 Services

À des fins évolutives, les différents services implémentent une interface. Celles-ci permettent, en cas de différentes implémentation du service, de pouvoir choisir celui que l'on souhaite utiliser. On pourrait par exemple imaginer un service effectuant des opérations sur les STB (ISTBService), avec deux implémentations possibles : La première utilisant une base de données (DatabaseSTBService), la seconde utilisant du xQuery (xQuerySTBService).

L'application utilise actuellement 3 services, possédant chacun une unique implémentation :

- IHomeService : Le service regroupant les opérations que peuvent effectuer les contrôleurs de la classe HomeController.
- ISTBService : Ce service permet de manipuler les STBs au travers de différentes telles que `getAllSTBs`, `getSTBAsHTML` ou encore `insertSTBFromString`.
- IXMLService : Ce service contient toutes les opérations qu'il peut être utile d'appliquer sur des fichiers XML, on pourrait par exemple citer la méthode `isXMLValid` qui permet de valider un flux XML en utilisant le fichier XSD donné.

2.3 DTO & Adapteurs

`DTO` signifie Data Transfer Object, il s'agit d'une classe similaire à une entité, qui sera donc transférée à un client. L'application comporte plusieurs `DTO`, comme par exemple la classe `STBStatusDTO` qui est renvoyée lors de l'insertion d'une STB. Voici un extrait de ce `DTO` :

```java
@XmlRootElement(name = "stb-status")
@XmlAccessorType(XmlAccessType.FIELD)
public class STBStatusDTO implements DTO {

    /**
     * The ID of the STB.
     */
    @XmlElement(name = "id")
    private Integer id;

    /**
     * The STB status.
     */
    @XmlElement(name = "status")
    private STBStatus status;

    // ...

}
```

Le réel avantage d'utiliser ces classes par rapport à une entité est que celles-ci ne sont justement par présentes en base de données, et peuvent également servir de filtre à une réelle entité. Par exemple, le `DTO STBSummaryDTO` permet de filtrer l'entité `STB` afin d'en garder les informations résumées. Afin de convertir une entité en un DTO nous utilisons un adapteur (ici : `STBAdapter`), qui permettent de faire le lien entre une entité et une DTO, ou encore une liste d'entités et un DTO etc...

2.4 FunctionalException & FunctionalExceptionHandler

Spring permet de créer une classe particulière appelée `ControllerAdvice` qui va se charger de gérer toutes les exceptions non traitées par l'application. Il a donc été choisi de créer un type d'exception prenant un `DTO` en paramètre, celui-ci sera donc la réponse à envoyer au client. Cette exception est appelée `FunctionalException` et le gestionnaire d'erreur associé est `FunctionalExceptionHandler`. Voici un exemple d'utilisation de ces deux composants :

```
@Override
public STB getSTB(Integer id) throws FunctionalException {
    // Get the STB and throw an exception if its null
    STB stb = this.stbRepository.findById(id).orElse(null);
    if (stb == null) {
        throw new FunctionalException(
            STB23Error.STB_NOT_FOUND.getErrorMessage(),
            new STBStatusDTO(id, STBStatus.ERROR), // Reponse qui sera renvoyee au client
            STB23Error.STB_NOT_FOUND.getHttpStatus()
        );
    }

    return stb;
}
```

```
@ControllerAdvice
public class FunctionalExceptionHandler extends ResponseEntityExceptionHandler {

    /**
     * Exception handler called when a FunctionalException is thrown.
     *
     * @param  err The FunctionalException thrown.
     * @param  req The user request.
     * @return     The FunctionalException error content.
     */
    @ExceptionHandler(value = { FunctionalException.class })
    public ResponseEntity<DTO> functionalError(FunctionalException err, WebRequest req) {
        // L'exception arrivera ici et sera traitee afin d'envoyer la reponse d'erreur au client.
    }

}
```

2.5 Résultat de ces composants

En mettant tous ces composants bout à bout, comme présenté sur le schéma au début de la première partie, nous arrivons donc à une architecture modulaire ou chacune des opérations effectuées est à sa place. Cela permet également d'avoir le moins de logique possible dans les contrôleurs, par exemple le contrôleur d'insertion d'une STB se résume à ceci :

```java
/**
 * Insert the given STB into the database.
 *
 * @param  stbString           The XML STB to insert.
 * @return                     The current status of the given STB.
 * @throws FunctionalException Exception thrown if the STB is invalid, or a duplicate.
 */
@PostMapping(
    value = "/stb23/insert",
    consumes = MediaType.APPLICATION_XML_VALUE,
    produces = MediaType.APPLICATION_XML_VALUE
)
public @ResponseBody STBStatusDTO insertSTB(@RequestBody String stbString) throws FunctionalException {
    STB stb = this.stbService.insertSTBFromString(stbString);
    return new STBStatusDTO(stb.getId(), STBStatus.INSERTED);
}
```

En effet, il n'est pas nécessaire de gérer les erreurs comme tout se fait au niveau de la classe `FunctionalExceptionHandler`.

2.6 Utilisation du service REST

Il existe deux manières d'utiliser le service REST :

1. Utiliser la version de production déployée sur Clever Cloud.
2. Utiliser les sources données sur le repository GitHub et les compiler.

2.6.1 Service en ligne

Afin d'utiliser la version de production du service REST, rendez-vous au bout de ce lien. Vous pourrez alors utiliser les différentes routes de l'API que vous pourrez trouver sur l'endpoint `/help`.

2.6.2 Exécution depuis les sources

Afin de compiler les sources :

1. Rendez-vous sur le repository GitHub afin de télécharger les sources.
2. Ouvrez un terminal dans le dossier racine et installez les dépendances Maven en exécutant la commande suivante : `mvn install -f pom.xml`.
3. Lancez ensuite l'application en exécutant la commande : `mvn spring-boot:run -f pom.xml`

3 Client

Le client a été réalisé en utilisant du HTML et du Javascript, il s'agit d'un client très basique sans design superflu. Le choix de ces technologies vient de leur simplicité d'utilisation et l'aisance que nous avions avec celles-ci, ainsi que leurs possibilitées (onglets, drag and drop...). De plus, ces technologies ne nécessitent aucune dépendance sur la machine de la personne souhaitant l'utiliser, un simple navigateur suffit.

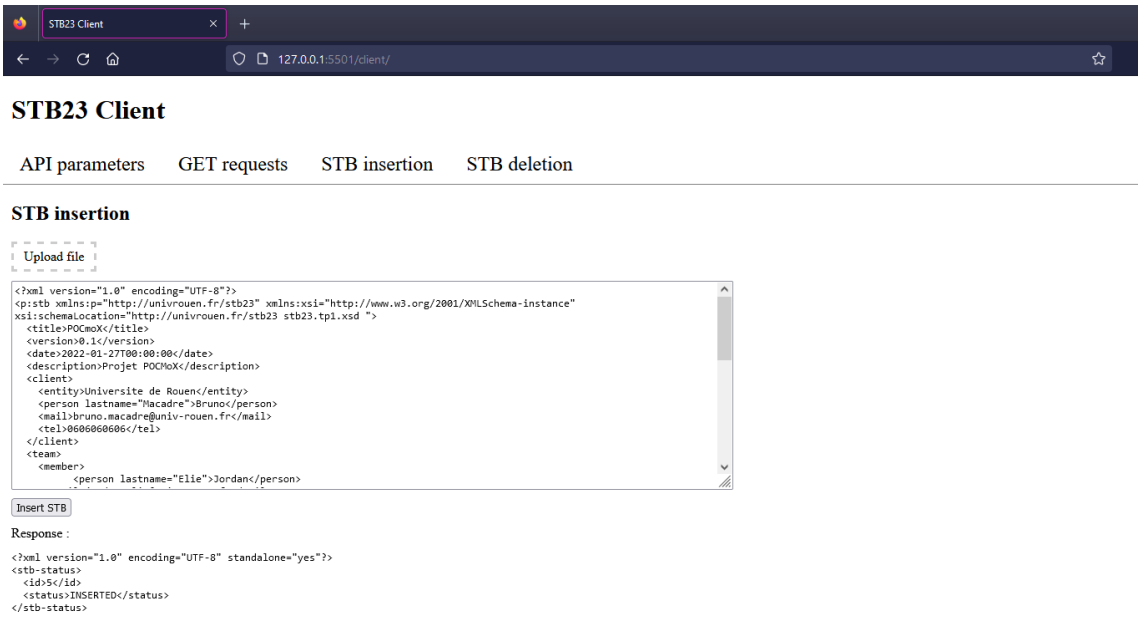

FIGURE 2 – Illustration d'une insertion de STB depuis le client.

3.1 Composants

Le client utilise divers composants (classes) afin de fonctionner d'une manière modulaire :

- `XmlBeautify` : Il s'agit d'une classe que nous n'avons pas développée (récupérée sur NPM), qui permet de styliser un flux XML (Indentation, retour à la ligne...).
- `TabList` : Cette classe permet de créer des onglets à partir des éléments donnés.
- `STBApiRequester` : Ce module a pour objectif de simplifier l'envoi de requêtes à l'API. Pour cela il se charge lui-même de récupérer les paramètres de connexion et d'effectuer les requêtes.
- `DragAndDropFileInput` : Cette classe permet d'ajouter des évènements à un élément afin que celui-ci puisse gérer le Drag and Drop de fichier.

3.2 Utilisation

Afin d'utiliser le client, il vous suffit de double cliquer sur le fichier `index.html` contenu dans le répertoire `client`. Le client devrait alors s'ouvrir dans votre navigateur.

4 Conclusion

Pour conclure, ce projet nous a permis d'apprendre à créer un service REST en utilisant Spring Boot, nous avons pu en profiter pour mettre en pratique ce que nous avons vu en cours et TP tels que la transformation XSLT ou encore la valdiation XSD. Cependant ce projet ne c'est pas déroulé sans difficulté, et n'est pas encore parfait, de nombreuses fonctionnalités peuvent encore être implémentées.

4.1 Difficultés rencontrées

Tout au long du projet nous avons pu rencontrer différentes difficultés, cependant parmi toutes celles-ci nous pouvons en citer deux majeures :

— La mise en place de l'architecture globale du projet : En effet, celle-ci a dû en premier lieu être réfléchie pour être la plus modulaire possible. Mais également en profiter pour utiliser au mieux les fonctionnalités de Spring.
— La gestion du Cross-Origins. En premier lieu les requêtes Cross-Origins ne nous ont pas posé problème comme nous développions en local, cependant à la mise en production nous nous sommes rendus compte qu'il fallait créer une classe de configuration permettant de permettre les requêtes Cross-Origins. La seconde difficulté fut de comprendre qu'il fallait le permettre pour les différentes méthodes de requêtes HTTP...

Cependant, le projet aurait pu être beaucoup plus complexe si nous avions dû le réaliser depuis zéro, heureusement la plupart des points qui aurait pu être bloquant ont été réalisés en TP (Validation XSD, transformation XSLT...).

4.2 Perspectives d'évolution

Ce projet pourrait être encore plus abouti avec de nouvelles fonctionnalités, voici quelques exemples :
— La possibilité de rechercher des STB par critères.
— Une meilleure gestion des logs avec par exemple plus de détails.
— La possibilité d'éditer une STB.