

RAPPORT
Architecture Logicielle
Projet Bibliothèque de manipulation d'expressions
symboliques
M1 Informatique GIL

Vallée Mathieu
Merre Alexandre

26 mai 2023

Table des matières

1	Introduction	4
2	Tâches à accomplir	4
2.1	Présentation du rapport	4
3	Service REST	5
3.1	Architecture globale	5
3.2	Controllers	5
3.3	Services	6
3.3.1	HomeService	6
3.3.2	STBHTMLServices	7
3.3.3	STBService	8
3.3.4	STBXMLService	10
3.4	Entities	12
3.5	Repository	12
3.6	Utils	12
3.6.1	MarshalMethods	13
3.6.2	ResourceReader	13
3.6.3	StatusGenerator	13
3.6.4	XMLStyleSheetManager	14
3.7	Addendum	14
4	Client	15
4.1	Architecture globale	15
4.2	Controller	15
4.2.1	http ://localhost :8100/	15
4.2.2	http ://localhost :8100/help	15
4.2.3	http ://localhost :8100/stb23/list	15
4.2.4	http ://localhost :8100/stb23/{id}	16
4.2.5	http ://localhost :8100/stb23/insert GET	16
4.2.6	http ://localhost :8100/stb23/insert POST	16
4.2.7	http ://localhost :8100/stb23/delete/{id}	16
4.3	Model.API	16
4.4	Model.services	17
4.5	Webapp.WEB-INF.jsp	17
4.5.1	Page d'accueil	17
4.5.2	Page d'aide	18
4.5.3	Liste des STB	19
4.5.4	Détail d'une STB	20
4.5.5	Insérer une STB	20
5	Conclusion	21
5.1	Pistes d'améliorations	21
5.2	Conclusion	21
6	Mode d'emploi	22
6.1	API	22
6.2	Client	22
6.3	Dépôt GitHub	22

6.4	URL Déploiement CleverCloud	22
-----	---------------------------------------	----

1 Introduction

Ce projet a pour objectif de mettre en pratique les concepts étudiés dans le cadre de l'UE XML/Langage Web 2. Nous nous concentrons sur le développement d'une API REST en utilisant le framework Spring et sa gestion de base de données. De plus, nous devons également créer un client autonome qui interagira avec cette API.

2 Tâches à accomplir

- Concevoir et développer l'API REST en utilisant le framework Spring.
- Configurer une base de données pour stocker les données de l'API. Nous choisirons une solution de base de données appropriée et mettrons en place les tables et les relations nécessaires pour gérer les informations de manière efficace.
- Mettre en oeuvre un système permettant de vérifier, sérialiser et dé-sérialiser du flux XML et les objet Java correspondant. Cette opération est essentielle pour le stockage en base de données.
- Créer le schéma XSD permettant la validation des XML susmentionnés ainsi que des feuilles XSLT permettant de convertir pour le client ce flux XML en flux HTML agréable à lire.
- Développer un client autonome qui interagira avec l'API REST. Ce client sera conçu pour offrir une interface conviviale et permettre aux utilisateurs de manipuler les données de l'API de manière intuitive.
- Documenter le projet en fournissant des instructions d'installation, des exemples d'utilisation et des explications sur la structure du code.

2.1 Présentation du rapport

En somme, ce projet nous permettra de consolider nos connaissances en développement d'applications web et en manipulation de données en mettant en pratique les concepts étudiés en cours. Grâce à la réalisation de cette API REST avec Spring et de son client autonome, nous développerons nos compétences techniques et notre capacité à concevoir des systèmes logiciels fonctionnels.

Dans ce rapport, nous nous efforcerons de fournir une analyse approfondie de nos compréhensions et des implémentations réalisées dans le cadre de ce projet. Nous commencerons par une section dédiée au service REST, où nous présenterons en détail son architecture, les technologies utilisées et son fonctionnement. Ensuite, nous aborderons le choix de notre client autonome, en détaillant son architecture ainsi que les différentes technologies employées. Nous poursuivrons en explorant les fonctionnalités bonus que nous avons ajoutées à notre application. Enfin, nous dresserons une liste de pistes d'améliorations envisageables pour notre projet, et conclurons en récapitulant les principaux points abordés.

3 Service REST

3.1 Architecture globale

Logiquement, étant donné que notre service est élaboré avec Spring Boot, celui-ci respecte le design pattern MVC. L'idée est de séparer les préoccupations efficacement et de faciliter la maintenance du code. Cela constitue aussi une bonne pratique pour notre future vie professionnelle. Notre service REST s'articule autour de 5 dossiers :

- **Controller** : Contenant la liste des controllers contenant les routes de notre API. Les différentes routes sont séparées dans des fichiers différents en fonctions des services appelés par les-dites routes.
- **Entities** : Contenant les objets java utilisés pour la persistance des données dans la base de données, la récupération des informations à persister se fera au moyen de Marshalling et Unmarshalling sur des flux XML, la bibliothèque utilisée est Jakarta.
- **Repositories** : Servant à effectuer le CRUD. Celui-ci est géré par Spring en intégralité.
- **Services** : Contenant la liste des services effectuant les opérations demandées par les controllers. On retrouve par exemple les insertions dans la base de données ou l'unmarshalling des fichiers XML.
- **Utils** : Contenant des classes utilitaires utilisées par les services.

De plus, nous avons également un bean, chargé de la configuration de la connexion à la base de données et un dossier **Resources** contenant les fichiers utiles à l'API pour la validation et transformation des XML.

3.2 Controllers

Rentrons plus en détails dans chacun des fichiers du package **Controller** :

- **IndexController** : Le fichier 'IndexController' est un contrôleur qui gère les requêtes pour les routes "/" et "/help". Il est annoté avec `@RestController`, ce qui indique qu'il retourne directement du contenu (dans notre cas, du HTML). Le contrôleur est responsable de l'injection de dépendance de **HomeService** et de l'appel des méthodes `getHTMLHomePage()` et `getHTMLHelpPage()` pour renvoyer respectivement la page d'accueil et la page d'aide au format HTML.
- **STBController** : Le fichier 'STBController' est un contrôleur qui gère les requêtes pour les routes `"/stb23/insert"` et `"/stb23/delete/id"`. Il est annoté avec `@RestController`, ce qui indique qu'il retourne directement du contenu (dans notre cas, du XML). Le contrôleur est responsable de l'injection de dépendance de **STBService** et de l'appel des méthodes `insert()` et `delete()` en fonction des requêtes reçues. La méthode `insert()` accepte une requête POST avec un corps XML (`@RequestBody`) et renvoie le résultat du traitement effectué par `stbService.insert(xml)`. La méthode `delete()` accepte une requête DELETE avec un paramètre d'URL (`@PathVariable`) et renvoie le résultat du traitement effectué par `stbService.delete(id)`. Les réponses produites sont au format XML (`MediaType.APPLICATION_XML_VALUE`).
- **STBHTMLController** : Le fichier 'STBHTMLController' est un contrôleur qui gère les requêtes pour les routes `"/stb23/resume"` et `"/stb23/html/id"`. Il est annoté avec `@RestController`, ce qui indique qu'il retourne directement du contenu (dans notre cas, du HTML). Le contrôleur est responsable de l'injection de dépendance de **STBHTMLService** et de l'appel des méthodes `getList()` et `getById()` en fonction des requêtes reçues. La

méthode `list()` accepte une requête GET et renvoie le résultat du traitement effectué par `stbhtmlService.getList()`. La méthode `get()` accepte une requête GET avec un paramètre d'URL (`@PathVariable`) et renvoie le résultat du traitement effectué par `stbhtmlService.getId(id)`. Les réponses produites sont au format HTML.

- **STBXMLController** : Le fichier 'STBXMLController' est un contrôleur qui gère les requêtes pour les routes `"/stb23/resume/xml"` et `"/stb23/xml/id"`. Il est annoté avec `@RestController`, ce qui indique qu'il retourne directement du contenu (dans notre cas, du XML). Le contrôleur est responsable de l'injection de dépendance de **STBXMLService** et de l'appel des méthodes `getList()` et `getId(id)` en fonction des requêtes reçues. La méthode `list()` accepte une requête GET et renvoie le résultat du traitement effectué par `stbxmlService.getList()`. La méthode `get()` accepte une requête GET avec un paramètre d'URL (`@PathVariable`) et renvoie le résultat du traitement effectué par `stbxmlService.getId(id)`. Les réponses produites sont au format XML.

Nous avons donc ici uniquement paramétré les différentes routes et fait les appels correspondant aux différents services. Nous avons également assuré le bon format attendu des réponses et entrée de chacune des routes.

3.3 Services

Rentrons plus en détails dans chacun des fichiers du package **Services** :

3.3.1 HomeService

```
package fr.univrouen.stb23v1.services;

import org.springframework.stereotype.Service;

import java.util.ArrayList;
import java.util.List;

@Service
public class HomeService {

    /**
     * Get in an HTML format the home page.
     * @return the home page in a String.
     */
    public String getHTMLHomePage() {
        // Implementation code...
    }

    /**
     * Get in an HTML format the help page.
     * @return the help page in a String.
     */
    public String getHTMLHelpPage() {
        // Implementation code...
    }
}
```

```

    }
}

```

Le fichier 'HomeService' est un service qui fournit des méthodes pour générer les pages HTML de la page d'accueil et de la page d'aide. Il est annoté avec @Service pour être identifié en tant que composant de service par le framework Spring.

La méthode `getHTMLHomePage()` génère la page d'accueil au format HTML. Elle utilise un objet `StringBuilder` pour construire progressivement le contenu de la page en ajoutant des balises HTML et des éléments tels qu'un titre, un en-tête, des informations de version, la liste des développeurs et un logo. La page finale est renvoyée sous forme de chaîne de caractères.

La méthode `getHTMLHelpPage()` génère la page d'aide au format HTML. Elle utilise des listes (`serviceName`, `urlList`, `methodsList`, `operationResumeList`) pour stocker les différentes informations sur les opérations proposées par le service REST. Elle itère ensuite sur ces listes pour construire dynamiquement le contenu de la page d'aide en utilisant des balises HTML appropriées. La page finale est renvoyée sous forme de chaîne de caractères.

3.3.2 STBHTMLServices

```

package fr.univrouen.stb23v1.services;

import fr.univrouen.stb23v1.utils.StatusGenerator;
import fr.univrouen.stb23v1.utils.XMLStylesheetManager;
import jakarta.xml.bind.JAXBException;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import javax.xml.transform.TransformerException;
import java.io.IOException;

@Service
public class STBHTMLService {

    @Autowired
    private STBXMLService stbxmlService;

    public String getList() throws JAXBException,
        IOException, TransformerException {
        return XMLStylesheetManager
            .getStbResumeHtml(stbxmlService.getList());
    }

    public String getById(String id) {
        try {
            return XMLStylesheetManager
                .getStbHtml(stbxmlService.getById(id));
        } catch (JAXBException |
            TransformerException | IOException e) {
            return StatusGenerator
                .generateStatusXML(id, "ERROR");
        }
    }
}

```

```

    }
}

```

Le service `STBHTMLService` est annoté avec `@Service` et utilise l'injection de dépendances via l'annotation `@Autowired` pour obtenir une instance de la classe `STBXMLService`.

Il fournit les fonctionnalités suivantes :

1. La méthode `getList()` permet d'obtenir une liste de spécifications STB au format HTML. Elle utilise la méthode `getList()` de `stbxmlService` pour récupérer la liste des spécifications au format XML, puis applique une feuille de style XSLT gérée par `XMLStylesheetManager` pour générer la version HTML de la liste.
2. La méthode `getById(String id)` permet d'obtenir une spécification STB spécifique au format HTML en utilisant son identifiant. Elle utilise la méthode `getById(id)` de `stbxmlService` pour récupérer la spécification au format XML, puis applique une feuille de style XSLT pour générer la version HTML correspondante. En cas d'erreur lors de la récupération ou de la transformation, une réponse d'erreur XML est générée à l'aide de `StatusGenerator.generateStatusXML()`.

Les exceptions `JAXBException`, `TransformerException` et `IOException` sont gérées lors de l'appel à `getById()`, et une réponse d'erreur est renvoyée en cas d'exception.

Le service `STBHTMLService` facilite la conversion des spécifications STB du format XML vers le format HTML en utilisant les fonctionnalités de transformation XML/XSLT fournies par `XMLStylesheetManager` et les fonctionnalités de récupération des données XML de `STBXMLService`.

3.3.3 STBService

```

package fr.univrouen.stb23v1.services;

import fr.univrouen.stb23v1.entities.STB;
import fr.univrouen.stb23v1.repositories.STBRepository;
import fr.univrouen.stb23v1.utils.StatusGenerator;
import fr.univrouen.stb23v1.utils.MarshalMethods;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class STBService {

    @Autowired
    private STBRepository stbRepository;

    /**
     * Insert a new STB specification.
     * @param xml the XML representation of the STB specification.
     * @return the status of the insert operation in XML format.
     */
    public String insert(String xml) {

```



```

STB stb;
try {
    stb = MarshalMethods.deserializeXml(xml);
} catch (Exception e) {
    return StatusGenerator
        .generateStatusInsertXML("ERROR", "INVALID");
}

Iterable<STB> stbList = stbRepository.findAll();
for (STB stbInList : stbList) {
    if (stbInList.getTitle().equals(stb.getTitle())
        && stbInList.getVersion().equals(stb.getVersion())
        && stbInList.getDate().equals(stb.getDate())) {
        return StatusGenerator
            .generateStatusInsertXML("ERROR", "DUPLICATED");
    }
}

try {
    stbRepository.save(stb);
} catch (Exception e) {
    return StatusGenerator
        .generateStatusInsertXML("ERROR", "INVALID");
}

return StatusGenerator
    .generateStatusXML(stb.getId().toString(), "INSERTED");
}

/**
 * Delete a specific STB specification by ID.
 * @param id the ID of the STB specification to delete.
 * @return the status of the delete operation in XML format.
 */
public String delete(String id) {
    try {
        stbRepository.deleteById(id);
    } catch (Exception e) {
        return StatusGenerator.generateStatusXML(id, "ERROR");
    }

    return StatusGenerator.generateStatusXML(id, "DELETED");
}
}

```

Le service `STBService` est annoté avec `@Service` et utilise l'injection de dépendances via l'annotation `@Autowired` pour obtenir une instance de la classe `STBRepository`.

Il propose les fonctionnalités suivantes :

1. La méthode `insert(String xml)` permet d'insérer une nouvelle spécification STB. Elle

prend en paramètre une représentation XML de la spécification et la désérise en un objet STB à l'aide de la méthode `MarshalMethods.deserializeXml()`. Si la désérialisation échoue, une réponse d'erreur XML est générée à l'aide de `StatusGenerator.generateStatusInsertXML()` avec le statut "ERROR" et la raison "INVALID".

Ensuite, la méthode vérifie si une spécification STB avec le même titre, version et date existe déjà dans la base de données en parcourant la liste des spécifications récupérées via `stbRepository.findAll()`. Si une correspondance est trouvée, une réponse d'erreur XML est générée avec le statut "ERROR" et la raison "DUPLICATED".

Enfin, la spécification STB est sauvegardée dans la base de données à l'aide de `stbRepository.save()`. Si une exception se produit lors de la sauvegarde, une réponse d'erreur XML est renvoyée avec le statut "ERROR" et la raison "INVALID". Sinon, une réponse XML est générée avec l'ID de la spécification nouvellement insérée et le statut "INSERTED".

2. La méthode `delete(String id)` permet de supprimer une spécification STB spécifique en utilisant son ID. Elle supprime la spécification à l'aide de `stbRepository.deleteById()`. Si une exception se produit lors de la suppression, une réponse d'erreur XML est renvoyée avec l'ID de la spécification et le statut "ERROR". Sinon, une réponse XML est générée avec l'ID de la spécification supprimée et le statut "DELETED".

3.3.4 STBXMLService

```
package fr.univrouen.stb23v1.services;

import fr.univrouen.stb23v1.entities.STB;
import fr.univrouen.stb23v1.repositories.STBRepository;
import fr.univrouen.stb23v1.utils.StatusGenerator;
import jakarta.xml.bind.JAXBException;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import javax.xml.transform.TransformerException;
import java.io.IOException;

@Service
public class STBXMLService {

    @Autowired
    private STBRepository stbRepository;

    /**
     * Get the list of all STB specifications in XML format.
     * @return the list of STB specifications in XML format.
     */
    public String getList() {
        Iterable<STB> stbList = stbRepository.findAll();

        StringBuilder result = new StringBuilder("<result>\n");
        for (STB stb : stbList) {
```

```

        result
        .append("<stb_id=\"")
        .append(stb.getId())
        .append("\">\n");

        result.append("\t<title>")
        .append(stb.getTitle())
        .append("</title>\n");

        result.append("\t<description>")
        .append(stb.getDescription())
        .append("</description>\n");

        result.append("\t<date>")
        .append(stb.getDate())
        .append("</date>\n");

        result.append("\t<client-entity>")
        .append(stb.getClient().getEntity())
        .append("</client-entity>\n");

        result.append("</stb>\n");
    }
    result.append("</result>");

    return result.toString();
}

/**
 * Get a specific STB specification by ID.
 * @param id the ID of the STB specification.
 * @return the STB specification with the specified ID.
 */
public STB getById(String id) {
    return stbRepository.findById(id).orElse(null);
}
}

```

La classe **STBXMLService** est annoté avec **@Service** et utilise l'injection de dépendances via l'annotation **@Autowired** pour obtenir une instance de la classe **STBRepository**. Elle est responsable de la manipulation des spécifications STB au format XML.

Attributs :

stbRepository : un attribut de type **STBRepository** annoté avec **@Autowired**. Il est utilisé pour accéder aux données des spécifications STB.

Méthodes :

getList() : une méthode publique qui retourne une chaîne de caractères représentant une liste de spécifications STB au format XML. Elle récupère toutes les spécifications STB en uti-

lisant `stbRepository.findAll()`, puis les concatène dans une chaîne de caractères au format XML.

`getId(String id)` : une méthode publique qui prend en paramètre un identifiant de spécification STB et retourne l'objet STB correspondant. Elle utilise `stbRepository.findById(id)` pour rechercher la spécification STB dans la base de données.

La classe `STBXMLService` est un service essentiel de l'application, fournissant des fonctionnalités liées à la manipulation et à l'accès aux spécifications STB au format XML.

3.4 Entities

On retrouve dans ce package, l'intégralité des objets Java servant à la dé-sérialisation des flux XML dans le but de les stocker dans notre base de donnée. Nous ne rentrerons pas dans le détail des différents fichiers, les attributs et annotations de ces classes sont conformes aux spécifications données lors de la première séance de TP.

Les annotations sont ici utilisées également pour noter la multiplicité de chacun des attributs. Ils servent notamment pour créer les tables de jointure automatiquement.

La partie dé-sérialisation est assurée par le package `jakarta.xml.bind.annotation.*`, avec les différentes annotations `@XML[...]` et la partie persistance des données est assurée par le package `jakarta.persistence.*` notamment avec les annotations `@Entity`, `@OneToOne`,

3.5 Repository

On retrouve dans ce package une unique interface, `STBRepository`, qui étend `CrudRepository` avec le type générique STB et l'identifiant de type String.

Cette interface sert de contrat pour la manipulation des données de la classe STB au sein de la couche d'accès aux données (Data Access Layer) de l'application. Elle utilise les fonctionnalités fournies par `CrudRepository` pour effectuer les opérations CRUD (Create, Read, Update, Delete) sur les objets STB dans la base de données.

En étendant `CrudRepository`, l'interface `STBRepository` hérite de méthodes prédéfinies telles que `save`, `findById`, `findAll`, `delete`, etc., qui permettent de manipuler les enregistrements de la table correspondant à l'entité STB dans la base de données. Les méthodes héritées peuvent être utilisées directement sans avoir besoin de les implémenter explicitement dans cette interface.

Ainsi, ce fichier joue un rôle important en fournissant une abstraction pour l'accès aux données des spécifications STB via des opérations de base, simplifiant ainsi leur utilisation et leur gestion dans le reste de l'application.

3.6 Utils

Rentrons plus en détails dans chacun des fichiers du package `Utils`

3.6.1 MarshalMethods

La classe utilitaire nommée `MarshalMethods` qui fournit des méthodes pour la sérialisation et la désérialisation d'objets STB au format XML à l'aide de JAXB.

Voici une explication détaillée des méthodes présentes dans ce fichier :

deserializeXml : Cette méthode prend en paramètre une chaîne de caractères représentant un document XML et renvoie un objet STB désérialisé à partir de cette chaîne. Elle utilise JAXB pour effectuer la désérialisation. Voici les étapes principales de la méthode :

- Elle crée un contexte JAXB en utilisant la classe `JAXBContext`.

- Elle crée un désérialiseur (`Unmarshaller`) à partir du contexte JAXB.

- Elle crée un objet `SchemaFactory` pour valider le document XML selon un schéma.

- Elle récupère le fichier du schéma XML en utilisant `DefaultResourceLoader` pour charger le fichier depuis le classpath.

- Elle crée un objet `Schema` à partir du schéma XML.

- Elle lit la chaîne de caractères XML avec `StringReader`.

- Elle utilise le désérialiseur pour convertir la chaîne XML en un objet STB, en prenant en compte la validation par rapport au schéma XML.

- Elle renvoie l'objet STB désérialisé.

serializeXml : Cette méthode prend en paramètre un objet STB et renvoie une chaîne de caractères représentant la sérialisation de cet objet en XML. Elle utilise JAXB pour effectuer la sérialisation. Voici les étapes principales de la méthode :

- Elle crée un contexte JAXB en utilisant la classe `JAXBContext`.

- Elle crée un sérialiseur (`Marshaller`) à partir du contexte JAXB.

- Elle crée une chaîne de caractères (`StringWriter`) pour stocker le résultat de la sérialisation.

- Elle utilise le sérialiseur pour convertir l'objet STB en XML en utilisant la méthode `marshal`.

- Elle renvoie la chaîne de caractères représentant la sérialisation XML.

En résumé, ce fichier fournit des méthodes pour convertir des objets STB en XML et vice versa en utilisant JAXB. Ces méthodes sont utilisées pour la manipulation des données STB au format XML, notamment dans le contexte de la couche de services de l'application.

3.6.2 ResourceReader

La classe utilitaire `ResourceReader` qui offre une méthode statique `asString` pour lire le contenu d'une ressource en tant que chaîne de caractères.

La méthode `asString` prend en paramètre un objet `Resource` provenant du framework Spring. Elle ouvre un flux de lecture (`Reader`) à partir de la ressource en utilisant l'encodage UTF-8. Ensuite, elle utilise la méthode `FileCopyUtils.copyToString` du framework Spring pour copier le contenu du flux de lecture dans une chaîne de caractères. En cas d'erreur de lecture (`IOException`), une exception `UncheckedIOException` est levée.

L'utilisation de cette classe facilite la lecture du contenu d'une ressource, telle qu'un fichier, en tant que chaîne de caractères dans le code Java.

3.6.3 StatusGenerator

La classe utilitaire `StatusGenerator` qui fournit des méthodes statiques pour générer du contenu XML représentant différents statuts d'erreurs.

La méthode `generateStatusXML` génère une chaîne de caractères représentant un élément XML contenant un identifiant (`id`) et un statut (`status`). La méthode utilise un objet `StringBuilder` pour construire la chaîne de caractères en ajoutant les balises XML correspondantes.

La méthode `generateStatusInsertXML` génère également une chaîne de caractères représentant un élément XML, mais cette fois-ci avec un statut (`status`) et un détail (`detail`). La construction de la chaîne de caractères est similaire à la méthode précédente, en utilisant un objet `StringBuilder`.

Ces méthodes permettent de générer facilement du contenu XML représentant des statuts ou des résultats dans le code Java. Notamment pour les retours d'erreur dans le format demandé dans l'énoncé.

3.6.4 XMLStyleSheetManager

La classe utilitaire `XMLStylesheetManager` qui fournit des méthodes statiques pour appliquer des feuilles de style XSLT à des documents XML.

La méthode `getStbHtml` permet de générer une représentation HTML d'un objet STB en utilisant une feuille de style XSLT. La méthode charge la feuille de style à partir d'un fichier XSLT, puis crée un objet Transformer à partir de la feuille de style. Ensuite, elle utilise un objet `JAXBContext` pour créer une source JAXB à partir de l'objet STB. La méthode effectue ensuite la transformation en utilisant l'objet Transformer et stocke le résultat dans une chaîne de caractères. Il s'agit de la feuille produite en TP pour représenter des STB sous forme de flux HTML.

La méthode `getStbResumeHtml` génère une représentation HTML résumée d'un document XML en utilisant une feuille de style XSLT. La méthode charge la feuille de style à partir d'un fichier XSLT, puis crée un objet Transformer à partir de la feuille de style. Ensuite, elle utilise un objet `StreamSource` pour représenter le flux XML en tant que chaîne de caractères. La méthode effectue la transformation en utilisant l'objet Transformer et stocke le résultat dans une chaîne de caractères. Cette méthode est utilisée pour afficher les STB sous forme de tableau HTML conformément à l'énoncé.

Ces méthodes permettent d'appliquer des feuilles de style XSLT à des documents XML pour générer des représentations HTML spécifiques.

3.7 Addendum

Il est aussi bon de noter la présence du fichier `application.properties` dans le dossier `resources` servant à paramétrer les credentials d'accès à la base de donnée ainsi que les paramètres de notre ORM, Hibernate, concernant la manière de générer la base de donnée. Notamment s'il doit créer ou pas les tables nécessaires lorsqu'elles sont absentes.

```
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.globally_quoted_identifiers=true

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://uxg0jycf8h1bznul:v05oAJ0rZqv30Wxamoyf@
bigbtyyj9og64pvbu1zp-mysql.services.clever-cloud.com:3306/bigbtyyj9og64pvbu1zp
spring.datasource.username=*****
spring.datasource.password=*****
```

4 Client

4.1 Architecture globale

Similairement à l'API, notre client est élaboré avec Spring Boot, la vue se fera au moyen de JSP donc, et notre organisation des responsabilités s'effectue aussi en suivant le design pattern MVC.

On retrouve 6 dossiers importants :

- **Controllers** : Contenant les classes chargées de configurer les routes du client pour appeler les services appropriés à chaque route.
- **Model.API** : Contenant les classes envoyant la requête à l'API et récupérant le résultat sous forme de chaîne.
- **Model.services** : Contenant les classes récupérant les réponses de l'API et effectuant le traitement de ces données. Généralement il s'agira de sérialisation/désérialisation de données pour que le controller puisse manipuler des objets Java.
- **Model.utils** : Contenant des classes utilitaires (Marshalling/Unmarshalling).
- **Model.entities** : Contenant les objets Java utilisés pour la désérialisation des flux XML renvoyés par l'API. Grandement similaire au dossier portant le même nom dans l'API.
- **webapp.WEB-INF.jsp** : Contenant les différentes vues de notre client.

Nous passerons plus vite sur certains dossier car certains des concepts sont les mêmes que pour le service. Voire même pas du tout pour les dossiers **Model.entities** et **Model.utils**, pour le premier, c'est l'exact même avec deux fichiers en plus, pour désérialiser les réponses de l'API pour les routes `/stb23/resume/xml` et `/stb23/xml/{id}`, pour le second en revanche, c'est exactement le même, minus les fichiers non pertinent pour le client.

4.2 Controller

Dans le service REST, nous utilisons l'annotation `@RestController`, pour dire que c'était un contrôleur REST. Ici nous utilisons simplement `@Controller`. La différence étant dans notre cas que nous pouvons manipuler les JSP, dont nous parlerons dans une prochaine partie.

Dans cette partie, nous allons énumérer les différentes routes des contrôleurs et détailler leur cheminement.

4.2.1 `http://localhost:8100/`

Cette route renvoie tout simplement à la page d'accueil de notre client. Lorsque que ce contrôleur est appelé, le service associé est appelé, qui appelle à son le service de requête à l'API, pour récupérer directement la page d'accueil HTML du service REST. Nous traitons cette page en lui ajoutant un menu de navigation, permettant de naviguer dans tout le client. Enfin, le contrôleur affiche la JSP avec le contenu HTML reçu. La méthode est GET.

4.2.2 `http://localhost:8100/help`

Cette route renvoie à la page d'aide de notre client. Le procédé est exactement le même que pour la page d'accueil, la méthode également.

4.2.3 `http://localhost:8100/stb23/list`

Cette route renvoie à la page pour afficher la liste des STB. Ici, le procédé est différent. Lorsque le contrôleur est appelé, le service associé est appelé qui appelle à son tour le service de requête à l'API. Le résultat de la requête est stocké sous forme d'une variable de type `ResultSTBList`, étant une nouvelle entité permettant de stocker le résultat de cette requête

spécifique. Le résultat est ensuite renvoyé au contrôleur pour être affiché via les JSP. La méthode utilisée est GET.

4.2.4 `http://localhost:8100/stb23/{id}`

Cette route renvoie à la page pour afficher le détail d'une STB. Ici, le procédé est très similaire à la liste des STB, et la méthode également.

4.2.5 `http://localhost:8100/stb23/insert` GET

Cette route renvoie juste vers la page JSP étant le formulaire de création pour pouvoir insérer une STB. Nous n'appelons donc ni le service, ni le service de requête à l'API mais uniquement la page JSP.

4.2.6 `http://localhost:8100/stb23/insert` POST

Cette route, a la même URL que la route précédemment citée, mais la méthode (POST) est différente. Cela permet au formulaire de création de STB d'envoyer des données POST. Ici une chaîne de caractère correspondant à un XML est envoyé. Le service est appelé, appelant à son tour le service de requête à l'API. Lorsque la requête d'insertion est faite, la réponse renvoyée est une chaîne de caractère de type XML. Nous avons créé une nouvelle entité permettant de gérer ces résultats : `ResultRequest`. Cela permet de vérifier si la requête a bien été insérée, si tel est le cas, nous affichons la page de détail de la STB créée, sinon nous renvoyons au formulaire de création.

4.2.7 `http://localhost:8100/stb23/delete/{id}`

Enfin la dernière route est très similaire à la précédente. La différence étant que l'ID de la STB à supprimer se trouve dans l'URL. Le reste du chemin est exactement pareil que celle pour insérer une STB.

4.3 Model.API

On retrouve dans ce dossier 3 fichiers :

- `APIConstant.java` : Contenant le chemin vers l'API, de manière à pouvoir simplement changer entre notre localhost et le vrai service déployé.
- `HomeRequest.java` : `HomeRequest` va essentiellement servir à faire les requêtes HTTP au service REST, permettant d'avoir les données des pages d'accueil et des pages d'aide.
- `STBRequest.java` : `STBRequest` va essentiellement servir à faire les requêtes HTTP au service REST, permettant d'avoir les données des pages concernant les STB. Par exemple, nous pouvons avoir la liste des STB, avoir le détail d'une STB, supprimer ou créer des STB.

Il est important de noter que nous ne stockons pas de données dans le client, celui-ci sert uniquement à l'affichage. Toutes les données sont donc récupérées depuis le service REST et manipulées pour être exploitable dans la vue.

Aussi, nous n'échangeons avec l'API qu'à travers du XML, qui était une contrainte du sujet, il faut donc appliquer du traitement à nos données XML que l'on envoie ou reçoit.

4.4 Model.services

Ces classes du modèle servent juste de facade pour faire des requêtes à l'API, et permettent par conséquent de relier les contrôleurs avec les requêtes API, tout en y appliquant des fonctions auxiliaires.

4.5 Webapp.WEB-INF.jsp

Ces sous dossiers contiennent des pages JSP. Nous avons en effet choisi JSP pour la simplicité d'installation, ainsi que par le manque de temps et de réflexion. C'est finalement un outil qui s'est avéré très efficace pour générer des pages HTML depuis Spring Boot. Dans les JSP, nous utilisons également les Expression Language (Spring EL). Cela permet d'afficher des variables de manière très simple et efficace.

Malheureusement, nous n'avons pas réussi à faire fonctionner TagLib, notre version de Spring étant trop récente d'après ce que l'on voit sur les retours en ligne. TagLib nous aurait permis de simplement boucler sur des variables de type List par exemple. Pour remédier à ce problème, nous avons utilisé du code Java dans les JSP, qui est tout aussi efficace que TagLib bien que moins facile à utiliser.

Durant cette partie, nous allons mettre des images des rendus graphiques de notre client, sur toutes les pages. Nous n'allons cependant pas tout détailler, étant donné que cela reste très similaire, mais nous allons détailler l'exemple le plus complexe, la page qui nous a donné le plus de mal.

Il faut noter que les pages sont très minimalistes, celles-ci n'ont pas de CSS ou de script par exemple.

4.5.1 Page d'accueil

Cette page disponible à la racine de l'URL ("http ://localhost :8100/") affiche la page d'accueil.

-
- [Accueil](#)
 - [Aide](#)
 - [Liste des STBs](#)
 - [Ajouter une STB](#)

Service REST & Client

Version 1.0

Développeurs :

- MERRE Alexandre
- VALLEE Mathieu



4.5.2 Page d'aide

Cette page disponible à l'URL "http://localhost:8100/help" affiche la page d'aide.

- [Accueil](#)
- [Aide](#)
- [Liste des STBs](#)
- [Ajouter une STB](#)

Page d'aide

Liste des opérations proposées par le service REST :

Page d'accueil

URL : /

Méthode attendue : GET

Résumé de l'opération (format attendu, format de retour, ...) : Format HTML ou XHTML valide. Affiche la page d'accueil.

Aide

URL : /help

Méthode attendue : GET

Résumé de l'opération (format attendu, format de retour, ...) : Format HTML ou XHTML valide. Affiche la page contenant les informations d'aide.

Liste des STB - Format XML

4.5.3 Liste des STB

Cette page disponible à l'URL "http://localhost:8100/stb23/list" affiche la page de la liste des STB.

- [Accueil](#)
- [Aide](#)
- [Liste des STBs](#)
- [Ajouter une STB](#)

Liste des STBs

Projet Alpha

Id : 13

Description : Description du projet Alpha

Date : 2008-09-29T03:49:45

Client Entity : Entreprise ABC

[Detail](#)

Supprimer

Cette page nous paraît être la plus pertinente à détailler. En effet, nous avons besoin de boucler sur la liste des STB pour toutes les afficher. Nous l'avons donc fait grâce au code suivant :

```
<% for (STBResume stb : (List<STBResume>) request.getAttribute("stbList")) {%>
    <h2><% out.print(stb.getTitle()); %></h2>
    <p>Id : <% out.print(stb.getId()); %></p>
    <p>Description : <% out.print(stb.getDescription()); %></p>
    <p>Date : <% out.print(stb.getDate()); %></p>
    <p>Client Entity : <% out.print(stb.getClientEntity()); %></p>
    <p><a href="<% out.print("/stb23/" + stb.getId()); %>">Detail</a></p>
    <form action="<% out.print("/stb23/delete/" + stb.getId()); %>" method="post">
        <button type="submit" class="btn-link">Supprimer</button>
    </form>
<% } %>
```

Cela permet d'avoir accès à tous les éléments que l'on a besoin, comme le titre, l'Id, etc qui sont utiles à l'affichage de la liste des STB. On peut voir que les expressions entre balise `<% %>` utilisent du code Java.

Il faut également noter que le tableau a été passé en argument via le contrôleur. En effet, les instance de type Model permettent de passer des arguments aux JSP.

4.5.4 Détail d'une STB

Cette page disponible à l'URL "http ://localhost :8100/stb23/id" affiche la page de détail d'une STB.

Celle-ci n'est pas disponible dans le menu nav, mais dans la liste des STB.

Projet Alpha

Description du projet Alpha

Version 1000.0

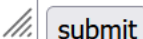
Date : 2008-09-29T03:49:45

4.5.5 Insérer une STB

Cette page disponible à l'URL "http ://localhost :8100/stb23/insert" affiche la page pour ajouter une STB. Afin de pouvoir insérer une STB, l'utilisateur doit rentrer un fichier XML dans la zone dédiée.

- [Aide](#)
- [Liste des STBs](#)
- [Ajouter une STB](#)

Entrez une STB en XML :



5 Conclusion

5.1 Pistes d'améliorations

L'application a sûrement beaucoup de points sur lesquels elle pourrait s'améliorer. Les principaux auxquels nous pensons sont :

- Nous avons beaucoup utilisé JAXB, permettant de rendre la gestion des données plus simple et plus flexible, pour transformer du XML en objet Java ou inversement. Nous pensons être sur la bonne voie quant à l'utilisation de cette librairie, cependant il est possible que nos éléments XML soient mal construits, et par conséquent cela peut fausser certaines requêtes.
- Notre Schéma XML nous paraît bon par rapport au TP1 et TP2, cependant il est également possible que celui-ci soit erroné, et nous pouvons l'améliorer davantage, notamment au niveau de sa lisibilité.
- L'utilisation des TagLibs dans les JSP aurait pu permettre une maintenabilité du client meilleure.
- Les configurations du service sont également très peu présentes dans notre projet, il y a sûrement moyen de configurer mieux Spring Boot.

5.2 Conclusion

Pour conclure, nous pensons que notre projet remplit les exigences du projet. Il y a bien entendu quelques défauts perceptibles. Mais le plus important est que ce projet nous a apporté beaucoup de connaissances quant-à l'utilisation du framework Spring Boot, mais également toutes les technologies XML. Nous y avons également vu quelques concepts intéressants, tels que celui d'avoir un service REST et un client interdépendant, ainsi que la partie sur le déploiement du service REST.

Un regret que nous avons eu est l'arrivée tardive du sujet, nous laissant peu de temps pour la réalisation de celui-ci, par rapport au contenu demandé, bien que nous étions initiés au principe de base de Spring Boot. Nous aurions aimé par exemple avoir plus de temps afin d'améliorer la qualité de notre projet, et faire plus d'expérimentations avec les différentes librairies et technologies de Spring Boot. Nous aurions également aimé travailler sur le framework Angular pour le client, afin de varier les technologies, mais c'est un choix que nous n'avons pu faire à cause du manque de temps.

6 Mode d'emploi

6.1 API

Le but de ce mode d'emploi est de pouvoir utiliser le client avec notre API. Pour que le client marche, il est donc nécessaire que l'API soit déployée sur CleverCloud. Si toutefois vous souhaitez la lancer en local, vous pouvez grâce aux commandes suivantes :

```
mvn install
mvn spring-boot :run
```

Cela lancera alors sur l'URL "http ://localhost :8080/" l'API. Les URL demandées ont été implémentées et sont donc connues.

6.2 Client

Le client a pour but d'être lancé avec l'API Déployé. L'URL de l'API est précisée dans le fichier "src/main/java/fr.univrouen.stb23v1/model/api/APIConstant.java".

Pour lancer le client, il faut faire les deux commandes suivantes :

```
mvn install
mvn spring-boot :run
```

Le client est alors démarré à l'URL suivante : "http ://localhost :8100/". Il y a normalement un menu de navigation sur chaque page, permettant de naviguer librement à travers celles-ci.

6.3 Dépôt GitHub

Le dépôt est disponible en public à l'URL suivante :

Dans la première branche "main", nous avons uniquement le service REST, qui sert à être déployé sur CleverCloud. Dans cette branche à sa racine, vous pouvez y retrouver le dossier "resources".

La seconde branche "client" contient le client de l'application, avec également le dossier "resources".

6.4 URL Déploiement CleverCloud

L'API déployée est disponible à l'adresse suivante : "https ://stb23-merre-vallee-rest-service.cleverapps.io/".

Biensur, elle n'est pas déployée tout le temps, mais le sera lorsque l'on sera prévenu de la correction.