

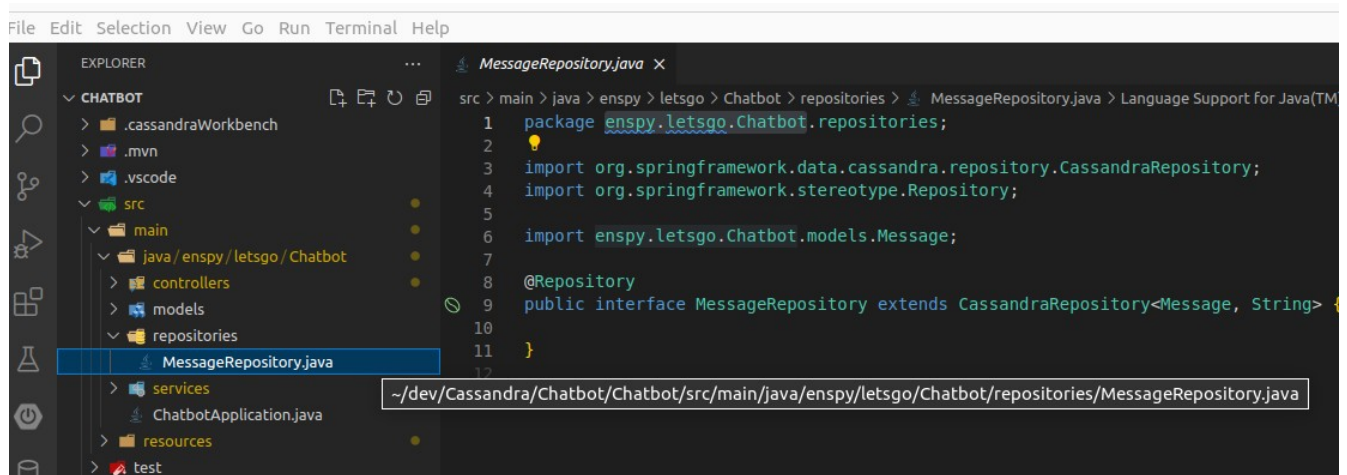
C'est une classe non persistante, elle sert pour le mappage de l'objet json {username, password} provenant du coté client afin de l'authentifier pour qu'il puisse visualiser l'historique des messages et des réponses.

## 2. Les interfaces d'accès aux données du package Repositories

Pour permettre l'accès à la base de données.

### → a) L'interface *MessageRepository*

Elle étend la classe `CassandraRepository` de la librairie `org.springframework.data.cassandra.repository` et permet d'implémenter l'accès à la base de données lorsqu'un service a besoin d'accéder à la BD pour effectuer une opération.



## II. La Couche de service

Elle contient tous les services de l'application implémentés à travers les méthodes de ses classes

### 1. Les Classes de service

#### → a) La classe *MessageService* du package *Services*

Accessible à travers le chemin `src/main/java/enspy/letsgo/chatbot/models/MessageService.java`, elle contient tous les services à travers les méthodes qui y sont implémentés, ainsi qu'une injection de dépendance de la classe `MessageRepository` pour accéder à la base de données. Parmi les services on a :

- `saveMessage()`

Permet de sauvegarder un objet message incluant sa réponse et sa catégorie, ainsi que l'heure et la date correspondantes

- `categorize()`

Permet de catégoriser les messages afin d'y apporter la réponse appropriée. C'est le noeud central de la logique métier de cette application

- getAllMessage()

Permet de recuperer L'historique de tous les messages pour la vue administrateur, ce qui permettra d'optimiser le fichiers de categorisation afin de mieux entrainer le modele plutard en vue de le rendre davantage performant.

```

11
12 @Service
13 public class MessageService {
14
15     @Autowired
16     private MessageRepository messageRepository;
17
18
19
20     public String categorize(Message message) {
21
22         BotResponse botResponse = new BotResponse();
23
24         String category = "";
25
26         try {
27             category = botResponse.findCategory(message.getMessageBody());
28         } catch (Exception e) {
29             // TODO Auto-generated catch block
30             e.printStackTrace();
31         }
32         return category;
33     }
34
35
36
37     public Message saveMessage(Message message, String responsebody){
38
39         message.setMessageId(UUID.randomUUID().toString().split("-")[0]);
40         message.setMessageBody(message.getMessageBody());
41         message.setEvent_time(LocalDate.now());
42         //message.setCategory(category);
43         message.setSenderUsername(message.getSenderUsername());
44         message.setResponseBody(responsebody);
45         messageRepository.save(message);
46         System.out.println("Message categorized and saved!!");
47
48         return message;
49     }
50
51
52
53     public List<Message> getAllMessage(){
54
55         return messageRepository.findAll();
56

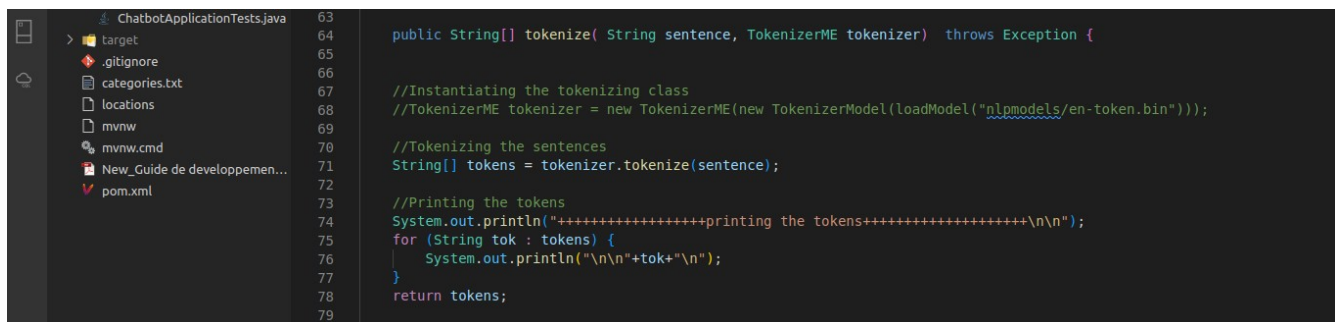
```

## b) Les classe utilitaire BotResponse

Elle est disponible dans le package Services>chatbotUtils. Elle contient les methodes du pipeline de categorisation d'apache openNLP. À savoir:

- tokenize()

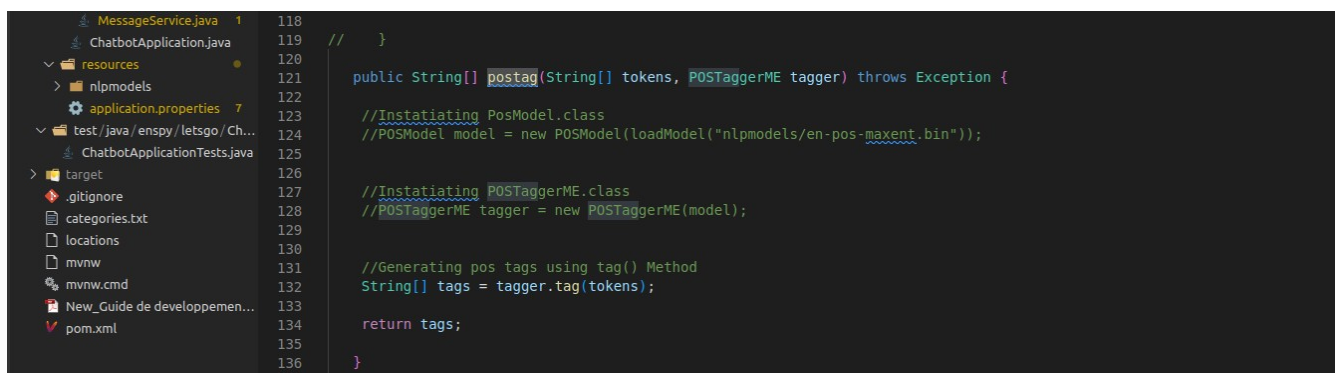
C'est la methode qui constitue la premiere étape de notre pipeline. Elle utilise des librairie d'apache openNLP pour tokenizer le message reçu et le decomposer en token afin de pouvoir le traiter.



```
63
64
65
66
67 //Instantiating the tokenizing class
68 //TokenizerME tokenizer = new TokenizerME(new TokenizerModel(loadModel("nlpmodels/en-token.bin")));
69
70 //Tokenizing the sentences
71 String[] tokens = tokenizer.tokenize(sentence);
72
73 //Printing the tokens
74 System.out.println("++++++printing the tokens++++++\n\n");
75 for (String tok : tokens) {
76     System.out.println("\n\n"+tok+"\n");
77 }
78 return tokens;
79
```

- Postag()

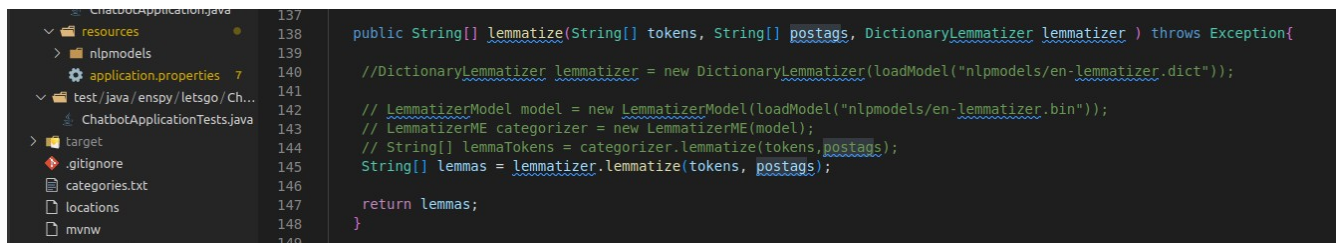
C'est le part of speech tagging qui constitue la seconde étape de notre pipeline. Elle utilise les librairies d'apache opennlp pour catégoriser chaque mot dans sa catégorie grammaticale appropriée, telle que nom, verbe, adjectif, adverbe, pronom, préposition, conjonction, etc. le part of speech tagging est une étape essentielle dans les tâches de traitement automatique du langage naturel (TALN), car il fournit des informations précieuses sur la structure syntaxique et les relations grammaticales au sein d'une phrase.



```
118
119 // }
120
121 public String[] postag(String[] tokens, POSTaggerME tagger) throws Exception {
122
123     //Instantiating PosModel.class
124     //POSModel model = new POSModel(loadModel("nlpmodels/en-pos-maxent.bin"));
125
126
127     //Instantiating POSTaggerME.class
128     //POSTaggerME tagger = new POSTaggerME(model);
129
130
131     //Generating pos tags using tag() Method
132     String[] tags = tagger.tag(tokens);
133
134     return tags;
135
136 }
```

- Lemmatize()

Elle constitue troisième étape de notre pipeline, elle utilise des bibliothèques d'Apache OpenNLP pour effectuer un processus de réduction d'un mot à sa forme de base ou de dictionnaire, appelée "lemme". Cela implique de normaliser les mots en les ramenant à leur forme canonique, ce qui facilite leur analyse et leur traitement ultérieur.



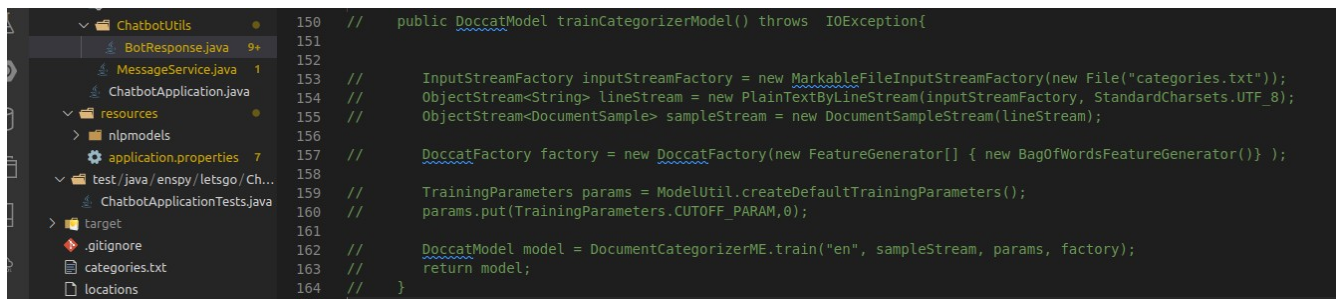
```

137
138
139
140
141
142
143
144
145
146
147
148
149
public String[] lemmatize(String[] tokens, String[] postags, DictionaryLemmatizer lemmatizer ) throws Exception{
    //DictionaryLemmatizer lemmatizer = new DictionaryLemmatizer(loadModel("nlpmodels/en-lemmatizer.dict"));
    //LemmatizerModel model = new LemmatizerModel(loadModel("nlpmodels/en-lemmatizer.bin"));
    //LemmatizerME categorizer = new LemmatizerME(model);
    //String[] lemmaTokens = categorizer.lemmatize(tokens, postags);
    String[] lemmas = lemmatizer.lemmatize(tokens, postags);
    return lemmas;
}

```

- Traincategorizer()

C'est la quatrième étape de notre pipeline (entièrement commentée parce qu'elle a été factorisée pour améliorer le temps de réponse de l'application) elle permet d'entraîner le modèle de catégorisation d'Apache OpenNLP à partir du fichier categories.txt qui se trouve à la racine du projet.



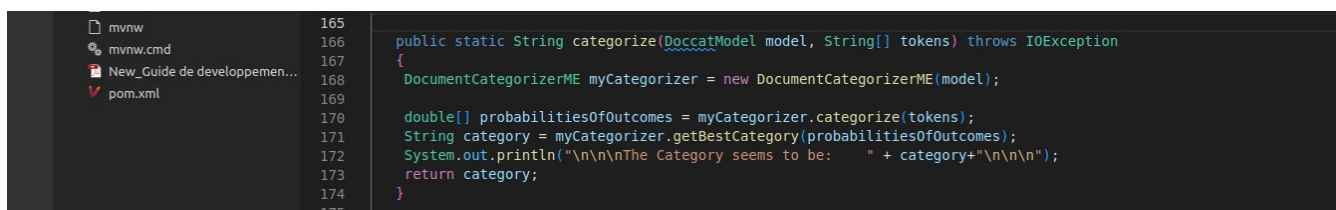
```

150 // public DoccatModel trainCategorizerModel() throws IOException{
151
152
153 //     InputStreamFactory inputStreamFactory = new MarkableFileInputStreamFactory(new File("categories.txt"));
154 //     ObjectStream<String> lineStream = new PlainTextByLineStream(inputStreamFactory, StandardCharsets.UTF_8);
155 //     ObjectStream<DocumentSample> sampleStream = new DocumentSampleStream(lineStream);
156
157 //     DoccatFactory factory = new DoccatFactory(new FeatureGenerator[] { new BagOfWordsFeatureGenerator() });
158
159 //     TrainingParameters params = ModelUtil.createDefaultTrainingParameters();
160 //     params.put(TrainingParameters.CUTOFF_PARAM, 0);
161
162 //     DoccatModel model = DocumentCategorizerME.train("en", sampleStream, params, factory);
163 //     return model;
164 // }
165

```

- Categorize()

Elle constitue la cinquième et dernière étape du pipeline, cela implique d'utiliser le modèle entraîné à la quatrième étape pour catégoriser les messages reçus afin de générer une réponse appropriée:



```

165
166
167
168
169
170
171
172
173
174
175
public static String categorize(DoccatModel model, String[] tokens) throws IOException
{
    DocumentCategorizerME myCategorizer = new DocumentCategorizerME(model);
    double[] probabilitiesOfOutcomes = myCategorizer.categorize(tokens);
    String category = myCategorizer.getBestCategory(probabilitiesOfOutcomes);
    System.out.println("\n\nThe Category seems to be: " + category + "\n\n");
    return category;
}

```



### III. La couche de controle

Elle gere l'interaction entre la vue et le modele, permettant ainsi de declencher les services en fonction de la requete Http entrante.

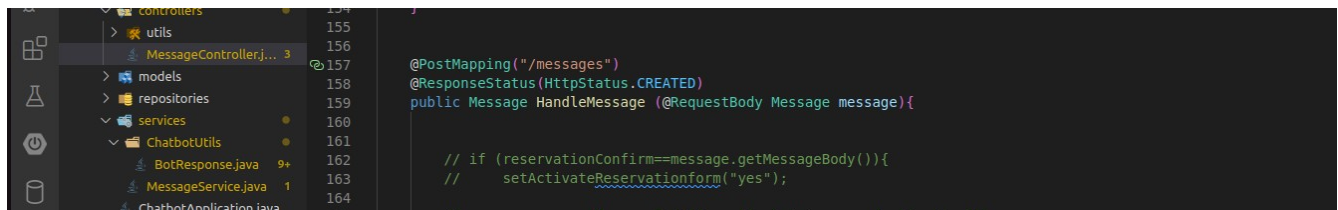
#### 1. Les classes de controle( package controller)

##### ➔ a) La classe MessageController

Elle se trouve a l'adresse src/main/java/enspy/letsgo/chatbot/controller/MessageController.java, elle contient les methodes de declenchement des services suivantes:

- HandleMessage()

Permet de recevoir et traiter tous message entrant sur le endpoint **“/message”** et retourne un objet JSON de type message contenant le champ Responsebody qui inclut la reponse et category qui inclue la categorie determiner par notre module d'intelligence artificielle.



```
154  
155  
156  
157 @PostMapping("/messages")  
158 @ResponseStatus(HttpStatus.CREATED)  
159 public Message HandleMessage (@RequestBody Message message){  
160  
161 // if (reservationConfirm==message.getMessageBody()){  
162 //     setActivateReservationform("yes");  
163  
164
```

- getAllMessage()

Permet de retourner tous les messages contenus dans la base de donnée



```
153  
154  
155  
156 @PostMapping("/api/login")  
157 public String login(@RequestBody Admin admin){  
158  
159 if (admin.getAdminUsername().equals(adminUsername) && admin.getAdminPassword().equals(adminPassword)){  
160 return "OK";  
161 }  
162 return "Bad login";  
163  
164
```

- login()

retourne OK si les informations de connexions contenues dans le payload sont correctes et “bad login” si l’une de ces informations est erronée

## ➔ ***Le package utilitaire utils***

Qui contient les classes utilitaires locationChecker.java et locationUtils.java qui permettent de vérifier si la destination entrée par l’usager est connue de l’application et en cas d’erreur, il lui fait des suggestions de destination dont l’orthographe se rapproche au maximum de la destination entrée par le client. L’approche pour déterminer cette destination proposées est basée sur la distance de Levenshtein

.